Stellaris® ARM® Cortex™-M3 Lab Manual

Getting Started with the Stellaris® Guru Evaluation Kit

Dhananjay V Gadre | Rohit Dureja | Shanjit S Jajmann

Netaji Subhas Institute of Technology (NSIT), New Delhi

Universities Press

Texas Instruments

Universities Press (India) Private Limited

Registered Office
3-6-747/1/A & 3-6-754/1, Himayatnagar, Hyderabad 500 029 (A.P.), India
email: info@universitiespress.com; www.universitiespress.com

Distributed by
Orient Blackswan Private Limited

Registered Office
3-6-752 Himayatnagar, Hyderabad 500 029 (A.P.), India

Other Offices
Bengaluru / Bhopal / Chennai / Ernakulam / Guwahati / Hyderabad / Jaipur / Kolkata / Lucknow / Mumbai / New Delhi / Noida / Patna

© Dhananjay V Gadre 2013

All rights reserved. No part of this document may be reproduced, stored in retrieval systems, or transmitted in any form or by any means, electronic, mechanical, photocopying or scanning, without the written permission of Dhananjay V Gadre.

Limits of liability: While the publisher and the authors have used their best efforts in preparing this book, they make no representation or warranties with respect to the accuracy or completeness of the contents of this book, and specifically disclaim any implied warranties of merchantability or fitness for any particular purpose. There are no warranties which extend beyond the description contained in this paragraph. No warranty may be created or extended by sales representatives or written sales materials. The accuracy and completeness of the information provided herein and the opinions stated herein are not guaranteed or warranted to produce any particular results, and the advice and strategies contained herein may not be suitable for every individual. Neither Universities Press nor the authors shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

Disclaimer: The contents of this book have been checked for accuracy. Since deviations cannot be precluded entirely, Universities Press or the authors cannot guarantee full agreement. As the book is intended for educational purpose, Universities Press and the authors shall not be responsible for any errors, omissions or damages arising out of the use of the information contained in the book. This publication is designed to provide accurate and authoritative information with regard to the subject matter covered. It is sold on the understanding that the Publisher is not rendering professional services.

Trademarks: All brand names and product names used in this book are trademarks, registered trademarks, or trade names of their respective holders. Universities Press is not associated with any product or vendor mentioned in this book.

ISBN 978 81 7371 881 6

For sale in India, China, Pakistan, Bangladesh, Sri Lanka, Nepal, Bhutan, Indonesia and Malaysia only

Cover and book design
© Universities Press (India) Private Limited 2013

Typeset in Life BT 11pt by
OSDATA, Hyderabad 500 049

Printed in India by

Published by
Universities Press (India) Private Limited
3-6-747/1/A & 3-6-754/1, Himayatnagar, Hyderabad 500 029 (A.P.), India

To my teachers

Professors Vijender Sharma, Ashok Dhall and Rajesh Mohan
and
Dr Ashutosh Singh (VU2IF)

–Dhananjay V Gadre

To my parents, my brother
and
my grandfather, who is my inspiration

–Rohit Dureja

To my parents and my brother

–Shanjit S Jajmann

Contents

Preface

The development of the Guru evaluation board and the subsequent writing of this lab manual marks the culmination of a long standing desire to create an ARM®-based platform for pedagogical applications. I designed an ARM-based system in 2006, but it used a time-limited Keil C compiler. As a teacher, I am not in favour of buying software tools for educational applications and time-limited or code-size-limited professional tools (such as Keil) never interested me. More recently, however, GNU ports for ARM microcontrollers started becoming available, and I thought the time was ripe to create an ARM platform for educational applications. Two years ago, I met Dr C P Ravikumar in rather fortuitous circumstances and our discussions led to the idea of Texas Instruments (India) sponsoring the development of a TI's Cortex™-M3-microcontroller-based evaluation kit. We started work on the development of Guru in December 2011 and had the first prototype ready and tested in May 2012. A first batch of mass produced Guru kits were successfully used in an Advanced Faculty Training Programme titled 'Hardware and Firmware Design for ARM-based Embedded Systems', conducted by Centre for Development of Advanced Computing (CDAC), Hyderabad, in June 2012. An initial user manual accompanied the Guru kit, and it was decided to upgrade the manual with additional details and experiments to a full-fledged lab manual, You, reader, are holding the result in your hands.

This lab manual is supported by a website (http://www.armcontroller.in), where you can find software examples and other materials. The manual has 12 chapters. Chapters 1 through 4 deal with the features and design of the Guru kit, TIŠs ARM Cortex™-M3 architecture in brief and GNU ARM® software toolchain installation. Chapter 5 deals with C programming for ARM. Chapters 6 through 12 deal with the various peripherals of the ARM microcontroller. Each chapter illustrates one particular peripheral in detail through a set of experiments. In all, the lab manual discusses 26 experiments that can be performed on the Guru kit apart from the several suggested experiments; users are encouraged to try their hand at performing these experiments.

The Guru kit can also be used as a useful resource for full-fledged projects including the mandatory final-year BTech project. We would love to hear about your exploits with it.

Acknowledgments

I acknowledge the following people for their help in various forms in completing this book: Dr C P Ravikumar, Director of University Relations at Texas Instruments India, without whose help this book would not have been possible, and Sagar Juneja, also from Texas Instruments India, for his efficiency in providing us chip samples and evaluation kits. Thanks are also due to several students at the Centre for Electronics Design and Technology (CEDT) at NSIT, who helped in various ways throughout the development of this manuscript and before that, during the designing of the Guru kit. I also thank Dr Sarat Babu, Executive Director, CDAC, Bangalore, for his constant reminder of the need for suitable teaching material on ARM microcontrollers, and Dr Sadanand Gulwadi, University Program Manager at ARM Embedded Technologies Private Ltd, for useful discussions and help with the initial prototype of the Guru Kit.

Thanks are also due to Harshit Jain and Anup Rajput, CEDT alumni, for their support and help, and to Vaishnavi Sundararajan, another CEDT alumnus, currently a PhD student of Computer Science at Chennai Mathematical Institute, for editing the book at our end, pro bono. This publication by Universities Press was facilitated by Sreelatha Menon, Managing Editor, whose faith in our capabilities was a big support to us all through the writing of the book.

And most of all, thanks are due to my wife Sangeeta and son Chaitanya for their love, care and understanding without which I could not have embarked upon or completed this project.

Dhananjay V Gadre
New Delhi, India

1 Introduction

This lab manual allows the user to get acquainted with the ARM® Stellaris® Cortex™-M3 microcontroller family through a hands-on approach by performing experiments on a hardware evaluation kit.

The experiments proposed in the manual are of progressively increasing level of difficulty. It exposes the readers to the various on-chip hardware features of the microcontroller based on an open source hardware kit which we designed and named 'Guru'. It is possible to perform several experiments using just Guru. However, more versatility in the range of experiments can be built in using the expansion options Guru offers through its onboard 'Arduino'[1] connector that connects to external interface circuits called 'shields'. Besides, by choosing a proper combination of a shield and appropriate software, one can even create several academic projects.

This manual, apart from describing the (we will use the term 'Guru' and 'Stellaris Guru' interchangeably in this manual), also describes several shields that allows various interfaces and sensors to be connected to the microcontroller. Since the Guru kit offers the Arduino interface, users can also design their own shields as well.

1.1 Meet the Guru Kit

The kit contains the following items:

1. The Stellaris® Guru circuit board
2. USB cable

The items can be seen in Figure 1.1.

Use the USB cable and connect the Stellaris® Guru kit to the USB port of your laptop or desktop computer (to which we will henceforth refer to as the host computer). A few indicator LEDs on the Guru kit should turn ON/flash, indicating that the board is in a ready-to-use state. To identify which LEDs would blink and which would turn on, refer to the annotated picture of the Stellaris® Guru kit in Figure **??**. You will observe that

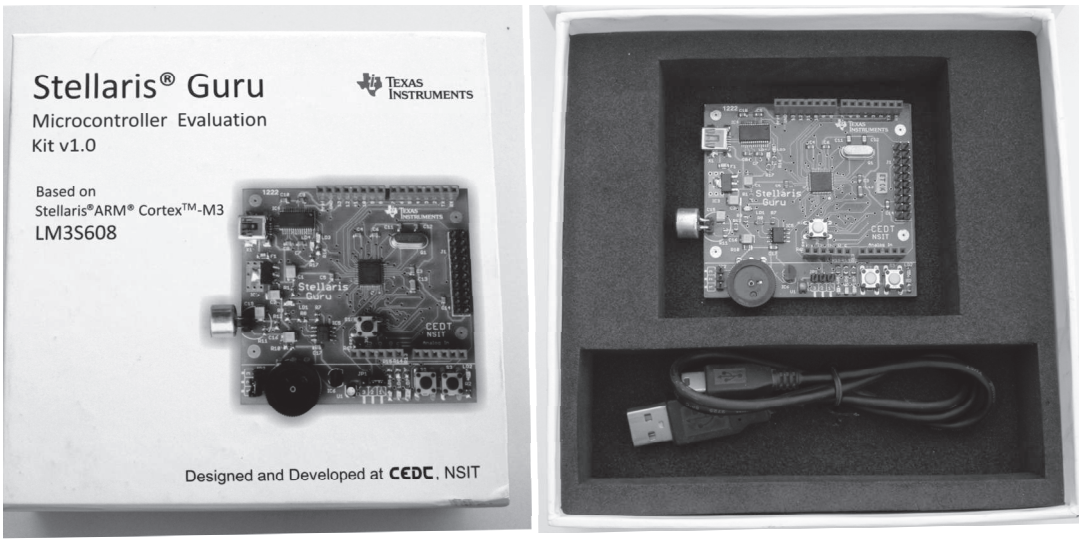

Figure 1.1 *Contents of the kit box*

- the power indicator LED, LD1, would turn on and remain on continuously to indicate that the Guru kit is receiving power from the host computer through the USB port.
- LEDs, LD3 and LD4, would flash momentarily, confirming that the Guru kit has established serial communication with the computer.

1.2 Step by Step!

Once you get the indication that your kit is working properly, take time off to review the contents of this lab manual.

Chapter 2 deals with the architecture of the Cortex™-M3 processor[2] and provides only an overview of the processor. For further details of the working of the processor, the reader is referred to the datasheet of the processor architecture from ARM®. This chapter also provides an overview of the system-on-chip details of Texas Instruments Stellaris® Cortex™-M3 family.

Chapter 3 provides the details of the Guru kit including the circuit diagram and its operation. It dwells in great detail on the choice of various components of the kit and on its power supply design.

Chapter 4 deals with the software tools that are necessary to program the Guru evaluation kit. The tools described include download links for the Sourcery CodeBench Lite Edition, which contains the GNU C/C++ Compliers, GNU Assembler, Debugger and other libraries,

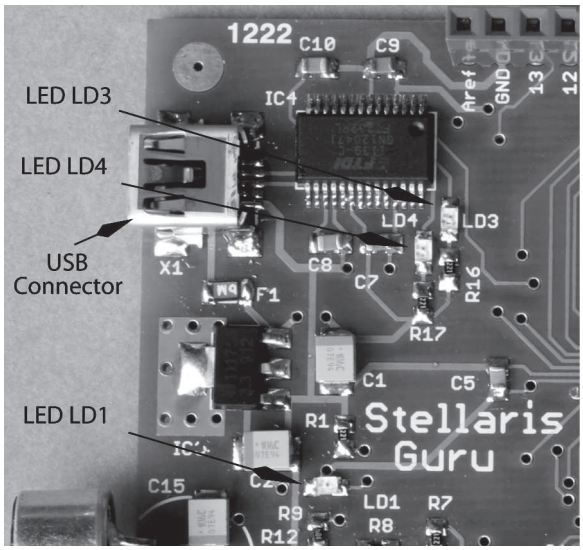

Figure 1.2 *A section of the Guru kit showing the important LEDs on the board*

Eclipse IDE and the LM Flash programmer. These represent a complete end-to-end solution for editing, compiling and debugging a C program and to download the resultant object code file to the Guru kit. These tools can also be used to write, assemble and debug an assembly language program as well, but this manual only illustrates programs and projects in C alone.

Chapter 5 deals with the important and necessary sections of a C program for the Stellaris® ARM® Cortex™-M3 controller, and the impact of each section on the final executable file.

Chapter 6 deals with the Stellaris® driver library available from TI, which simplifies the programming of the microcontroller by use of a common and simplified software interface for all the peripheral devices present on the Stellaris® microcontroller. While the published library from TI has hundreds of functions, this manual uses only a small subset of these library functions that are most commonly used. These functions are described in this chapter.

Chapter 7 deals with the digital input and output ports on the Stellaris® microcontroller. It also tells you how to program the common input and output devices such as LEDs and switches and to connect them to the input and output port pins.

Chapter 8 deals with the interrupts offered by the Stellaris® microcontroller. The Stellaris® Cortex™-M3 has a rich interrupt structure, with each on-chip peripheral equipped with several interrupt sources. Also, each digital input pin acts as an interrupt

source. Issues related to creating interrupt service routines (ISR) that function in tandem with the main program are described here.

Chapter 9 deals with the several timers available in the Stellaris® microcontroller and describes the various modes in which these timers can be operated in and how they can be used to generate time delays, interrupts, etc.

Chapter 10 describes the universal asynchronous receiver/transmitter links (UART) available on the Stellaris® microcontroller for communication with a host computer or with other devices (such as another Guru kit). Many of the Stellaris® microcontrollers have multiple UART channels. The LM3S608 Stellaris® microcontroller on the Guru kit has two UART channels.

Chapter 11 deals with the on-chip multi-channel ADC (analog-to-digital convertor) and its various applications and the uses of the ADC peripheral. As most of the Stellaris® microcontrollers do not have an on-chip DAC (digital-to-analog converter), the PWM function together with an external low pass filter can be used to create slow rate DACs. This chapter shows how to do that.

Chapter 12 illustrates the various operating modes of the microcontroller—active mode, power saving mode and sleep mode—and how to invoke them.

In the next chapter, Chapter 2, we briefly look at the ARM® Cortex™-M3 architecture. But before that, here is a list of suggested experiments that can be done using Guru.

1.3 Suggested List of Experiments

The experiments listed below can all be implemented on the Guru kit. You will be guided on what experiments to perform at various stages as you progress through the chapters. This will help to ensure that you have a gentle learning curve.

1. Blink an LED using the Register Access Model.
2. Control an LED through a switch by polling method using the Register Access Model.
3. Blink an LED using functions from StellarisWare API.
4. Control an LED through a switch by polling method using functions from StellarisWare API.
5. Blink LEDs in a controlled pattern.
6. Control intensity of an LED using PWM implemented in software.
7. Mimic light intensity sensed by the light sensor by varying the blinking rate of an LED.
8. Control an LED through a switch by interrupt method and flash the LED once in every five switch presses.

9. Blink an LED using delays generated using the SysTick timer.
10. Control intensity of an LED using PWM implemented in hardware.
11. System reset using the Watchdog timer in case something goes wrong.
12. UART Echo Test.
13. Control intensity of an LED on parameters received over UART.
14. Plot light intensity sensed by the light sensor on PC.
15. Generate a real time clock using timers and output time over UART.
16. Take analog readings on rotation of a rotary potentiometer connected to an ADC channel.
17. Temperature indication on an RGB LED.
18. Display temperature on PC by sending values over UART.
19. Sample sound using a microphone and display sound levels on LEDs.
20. Sample sound and plot amplitude vs. time on PC.
21. Evaluate the various sleep modes by putting core into sleep and deep-sleep modes.
22. System clock real time alteration using the PLL modules.
23. Control intensity of an RGB LED to generate composite colours using PWM implemented in software.
24. Control intensity of an RGB LED to generate composite colours using PWM implemented in hardware.
25. Filter the sound input using three filters of high, medium and low frequencies and show output on LEDs, one for each filter.
26. Sample sound and analyse its spectrum using FFT and plot amplitude vs. frequency results on a PC.

2 ARM® Cortex™-M3 Core and Stellaris® Peripherals

The ARM® Cortex™ family comprises processors based on three distinct *profiles* of the ARMv7 architecture, viz.,

1. the *A profile* for sophisticated, high-end applications running on open and complex operating systems,
2. the *R profile* for real-time systems, and
3. the *M profile* optimised for cost-sensitive and microcontroller applications.

The architecture of the ARM® Cortex™-M family of processors supports the performance requirements of diverse applications in a wide range of domains including automotive systems, networking and industrial applications. Cortex™-M3 is the first ARM® processor based on the ARMv7-M core architecture, specifically designed to achieve high performance in power- and cost-sensitive applications.
This high performance 32-bit processor offers the following significant benefits:

- Outstanding processing performance combined with fast interrupt handling
- Rapid application execution through Harvard architecture characterised by separate instruction and data buses
- Enhanced system-debug capability with extensive breakpoint and trace capabilities
- Efficient processor core, system and memories
- Ultra-low power consumption with integrated-sleep and deep-sleep modes
- Thumb instruction set which combines high code density with 32-bit performance
- Optional memory protection unit (MPU) for safety-critical applications

2.1 Overview of ARM® Cortex™-M3 Architecture

The ARM® Cortex™-M3 processor is built on a high-performance processor core with Harvard architecture and a 3-stage pipeline, ideal for embedded applications. It delivers

outstanding power efficiency through a combination of highly optimised instruction set and a superior design enabled by a high-end processing hardware including single-cycle 32 bits $\times$32 bits multiplication and hardware division operations.

The Cortex™-M3 implementation allows the system components to be tightly coupled, thereby reducing processing overheads and processor area while significantly improving the interrupt handling and debug capabilities of the processor. The Cortex™-M3 uses a version of the Thumb instruction set based on Thumb-2 technology[2] that ensures high code density and reduced program memory requirements. The instruction set provides the exceptional performance expected of the 32-bit processors with the high code density of 8-bit and 16-bit processors. Figure 2.1 shows the architecture of the ARM® Cortex™-M3.

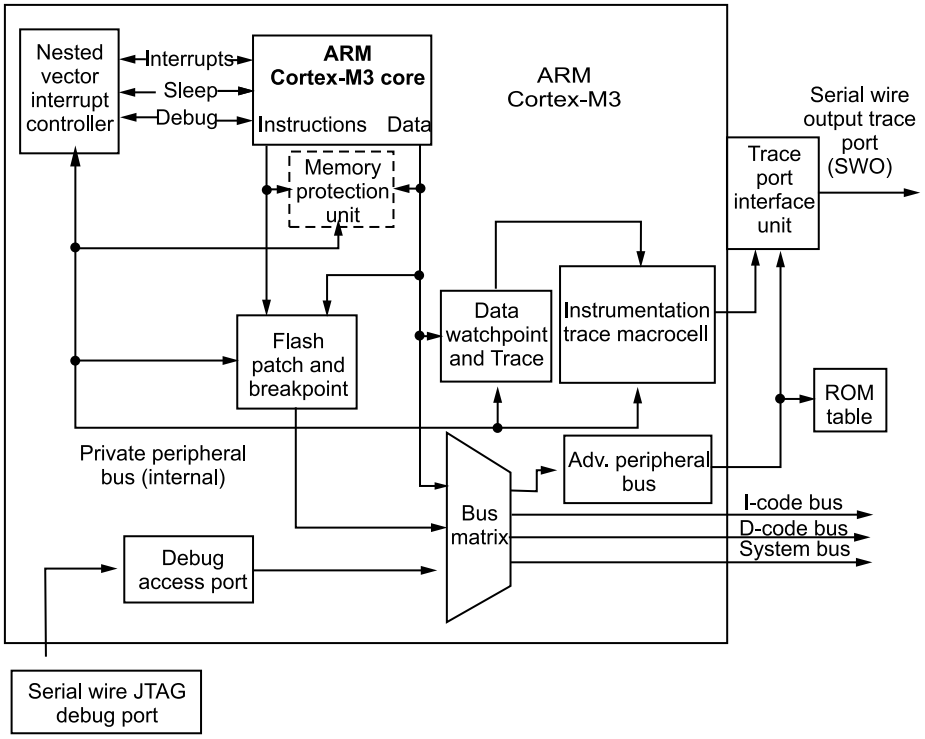

Figure 2.1 *Architecture diagram (ARM® Cortex™-M3 Technical References Manual, Revision r2p1)*

The salient features of the Cortex™-M3 are summarised below; these will be discussed in the subsequent sections.

- 32-bit processor with 32-bit data path, 32-bit register bank and 32-bit memory interfaces
- Harvard architecture with separate instruction and data buses
- 3-stage pipeline comprising Instruction fetch, Instruction decode and Instruction execute, with branch prediction

- The instruction fetch operation involves 32-bits. Up to two instructions can be fetched in one cycle, resulting in more bandwidth being available for data transfer.
- ALU with hardware divide and single-cycle multiply operations
- Nested vector interrupt controller (NVIC) configurable to a maximum of 240 external interrupts
- An optional memory protection unit (MPU) that permits control of individual regions in the memory
- Configurable processor modes and privilege levels giving more control over application execution and security

2.2 Cortex™-M3 Core Peripherals

The Cortex™-M3 includes the following system components:

- **SysTick** - The system timer is a 24-bit countdown timer that can be used as a real-time operating system (RTOS) tick timer or a simple counter.
- **Nested Vector Interrupt Controller (NVIC)** - The NVIC is an embedded interrupt controller that supports low latency interrupt handling and processing.
- **System Control Block (SCB)** - The SCB is the programmer's model interface to the processor. It provides system implementation information, control configurations, system control and reporting of system exceptions.
- **Memory Protection Unit (MPU)** - The MPU improves system reliability by defining memory attributes for different memory regions. It provides up to 8 distinct regions and an optional predefined background region.

2.3 Programmer's Model

The Cortex™-M3 processor is based on the ARMv7-M architecture, which includes the entire 16-bit Thumb instruction set and the base Thumb-2 32-bit Instruction Set Architecture (ISA). Thumb-2 is a major enhancement to the Thumb instruction set architecture, enabling higher code density than Thumb and higher performance with 16/32-bit instructions.

This section describes the Cortex™-M3 programmer's model, where individual registers, processor modes and access levels are dealt with.

2.3.1 Core Registers

Cortex™-M3 has the following 32-bit registers:

- 13 General-purpose registers, R0 to R12
- Stack pointer comprising two banked registers, main stack pointer (MSP) and program stack pointer (PSP)

- Link register, R14
- Program counter, R15
- Special registers, like program status register (PSR), exception mask registers and control register

The register map of the Cortex™-M3 is shown in Figure 2.2.

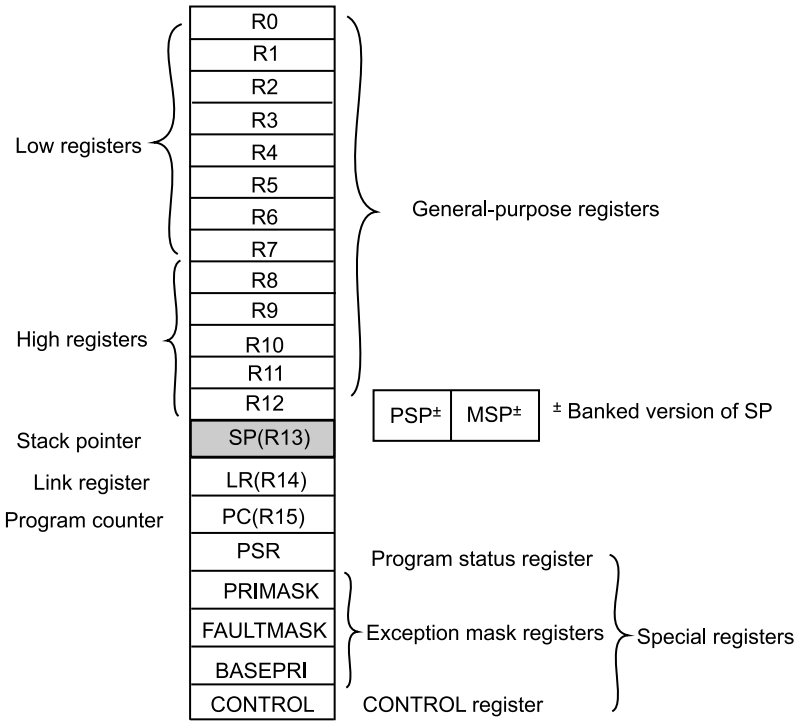

Figure 2.2 *Register map (ARM® Cortex™-M3 Technical Reference Manual, Revision r2p1)*

- **General-purpose registers:**

  1. R0 to R12 are general-purpose registers for data operations.
  2. Low registers: Registers R0 to R7 are accessible by all instructions that specify a general-purpose register.
  3. High registers: Registers R8 to R12 are accessible by all 32-bit instructions that specify a general-purpose register.

  Registers R13, R14 and R15 have the following special functions.

- **Stack pointer:** Register R13 is the stack pointer. It contains two stack pointers that are banked so that only one is visible at a time.

1. Main Stack Pointer (MSP) used by the stack pointer and exception handlers,
2. Program Stack Pointer (PSP) used by the application code.

- **Link register:** Register R14 is the link register. It stores the return information for subroutines, function calls and exceptions.

- **Program counter:** The program counter (PC) is Register R15. It holds the current program address. On reset, the PC is loaded with the value of the reset vector.

  Cortex™-M3 also has a number of special registers. They are:

- **Program status register (xPSR):** It provides arithmetic and logic processing flags, execution status and the number of the currently executing interrupt and includes the application PSR, the interrupt PSR and the execution PSR.

- **Interrupt mask registers:**

  1. PRIMASK, which disables all interrupts except the nonmaskable interrupt (NMI) and hard fault
  2. FAULTMASK, which disables all interrupts except the NMI
  3. BASEPRI, which disables all interrupts of a specified or lower priority levels

- **Control register (CONTROL):** The control register defines the privilege status and the pointer selection.

2.3.2 Operating Modes

The processor supports two modes of operation, namely, thread mode and handler mode.

- The processor enters the thread mode upon reset or when it returns from an exception. Both privileged code and user code can run in thread mode.
- The handler mode is entered into as a result of an exception. All code is privileged in the handler mode.

2.3.3 Operating States

The processor can operate in one of the two operating states:

- Thumb state: This is the normal execution mode, running 16-bit and 32-bit halfword-aligned Thumb and Thumb-2 instructions.
- Debug state: This is the state when halting in debug mode.

2.3.4 Privilege Levels

The code can execute as privileged code or unprivileged (user) code. Unprivileged execution limits or excludes access to some of the resources, while the privileged mode has access to all the resources.

- **Unprivileged** - In this mode, software has the following restrictions.
  - No access to system timer, NVIC or system control block
  - Restricted access to memory and peripherals
- **Privileged** - In this mode, software can use all instructions and has access to all resources. In thread mode, the CONTROL register controls whether the processor uses the main stack or the program stack. In handler mode, the processor always uses the main stack. The options for processor operations are summarised in Table **??**.

Table 2.1 *The various privilege levels and operating modes of the processor*

Processor mode	Use	Privilege level	Stack used
Thread	Applications	Privilege or user	Main or program stack
Handler	Exception handler	Privilege	Main stack

2.4 Memory Model

This section describes the processor's memory map, the behaviour of memory accesses and its bit-banding features.

2.4.1 Memory Map

The Cortex™-M3 processor has a memory-mapped system with up to 4 gigabytes of addressable memory space with predefined, dedicated addresses for code (code space), SRAM (memory space), external memories/devices and internal/external peripherals. There is also a special region to provide vendor-specific addressability. Figure 2.3 shows the memory map of the ARM® Cortex™-M3.

2.4.2 Bit-Banding

A bit-band region maps each word in a bit-band alias region to a single bit in the bit-band region. The bit-band region occupies the lowest 1MB of the SRAM and the peripheral memory regions.

The memory map has two 32MB alias regions, one in the SRAM and the other in the peripherals memory region that map to two 1MB bit-band regions in the respective spaces, whereby

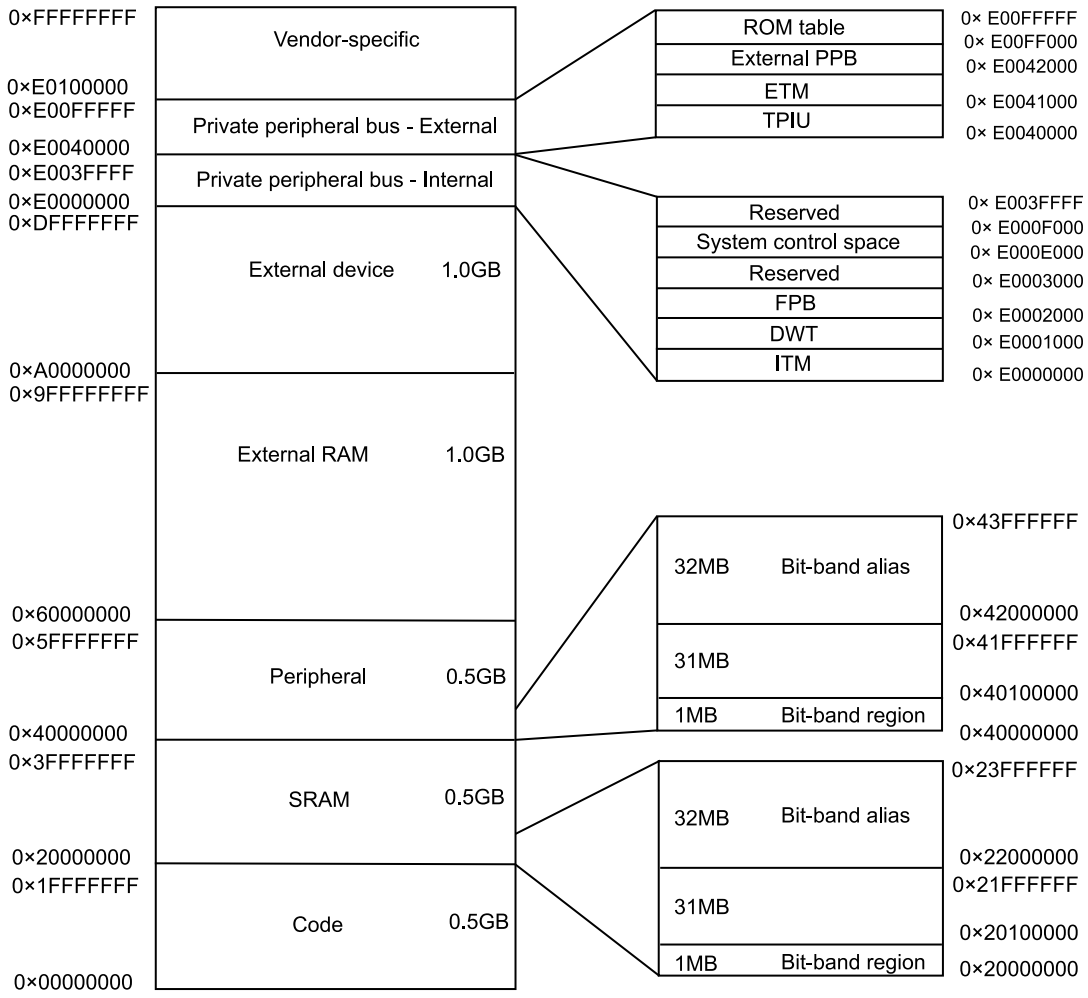

Figure 2.3 *Memory map*

- accesses to the 32MB SRAM alias region map to the 1MB SRAM bit-band region.
- accesses to the 32MB peripheral alias region map to the 1MB peripheral bit-band region.

A word access to the SRAM or peripheral bit-band alias region maps to a single bit in the SRAM or peripheral bit-band region. Bit band accesses can use byte, halfword, or word transfers. The bit band transfer size matches the transfer size of the instruction making the bit band access. Bit banding is further explained using Figure 2.4.

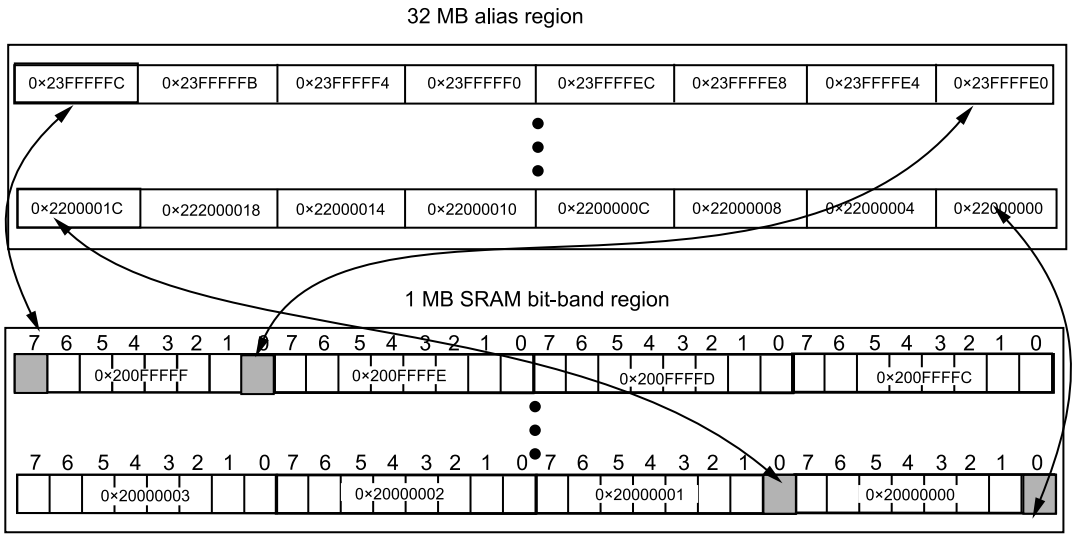

Figure 2.4 *Bit banding (Stellaris® LM3S608 microcontroller datasheet)*

2.5 Texas Instruments Stellaris® ARM® Cortex™-M3 Family

In the Stellaris® Guru development board, we use an ARM® Cortex™-M3 based microcontroller from the Stellaris® family of Texas Instruments. Before getting into the details of the controller used in the Stellaris® Guru kit, we would like to highlight some of the prime features available on the Stellaris® family of microcontrollers from Texas Instruments.

- **ARM® Cortex™-M3 v7-M Processor Core**
  - Up to 80 MHz
  - Up to 100 MIPS (at 80 MHz)
- **On-Chip Memory**
  - 256 kB Flash, 96 kB SRAM
  - ROM loaded with Stellaris® Driver-Lib, boot-loader, AES tables and CRC
- **External Peripheral Interface (EPI)**
  - 32-Bit dedicated parallel bus for external peripherals
  - Supports SDRAM, SRAM/Flash, M2M
- **Advanced Serial Communication**
  - 10/100 Ethernet MAC and PHY

  - Three CAN 2.0 A/B controllers
  - USB full speed, OTG/Host/Device
  - Three UARTs with IrDA and ISO 7816 support
  - Two I2Cs
  - Two synchronous serial interfaces (SSI)
  - Integrated interchip sound (I2S)
- **System Integration**
  - 32 Channel direct memory access (DMA) controller
  - Internal precision 16MHz oscillator
  - Two watchdog timers with separate clock domains
  - ARM® Cortex™ SysTick timer
  - Four 32-bit timers with real-time clock (RTC) capability
  - Low power battery backed hibernation module
  - Flexible pin-muxing capability
- **Advanced Motion Control**
  - Eight advanced PWM outputs for motion and energy applications
  - Two quadrature encoder inputs (QEI)
- **Analog**
  - 2×8 Channel 10-bit ADC (for a total of 16 channels)
  - Three analog comparators
  - On-chip voltage regulator (1.2V internal operation)

The controller used on the Stellaris® Guru board might not have all the features outlined above. We discuss the features and capabilities of the Stellaris® LM3S608 ARM® Cortex™-M3 based microcontroller, which is the heart of the Guru development kit. But before we do that, the entire TI Stellaris® family is briefly showcased in Table 2.2 for your reference.

The LM3S608 belongs to the 600 Series family characterised by 32 kB flash, 8 kB SRAM, a max speed of 50 MHz, 10 bit ADC, dual USART, I2C, SSI and motion control, and is available in a 48LQFP package.

2.6 Texas Instruments Stellaris® LM3S608 Microcontroller

2.6.1 Features and Peripherals

A Texas Instruments Stellaris® LM3S608 microcontroller forms the heart of the Stellaris® Guru board. The following sections describe the various features available on the controller and its pin out. Figure 2.5 shows the various features of the LM3S608.

Table 2.2 *Stellaris® Family of Cortex™-M3 microcontrollers*

Family	**Flash** (kB)	**SRAM** (kB)	**Max Speed**(MHz)	**I/O Pins**	**Package**	**Features**
x00 Series (Real Time MCUs)	8 kB to 64 kB	2 kB to 8 kB	20MHz to 50MHz	Upto 36 Pins	48LQFP, 48QFP	10bit ADC, USART I2C, SSI, Motion control
1000 Series (Real Time MCUs)	64 kB to 512 kB	19 kB to 96 kB	25MHz to 80MHz	Upto 67	64LQFP, 100LQFP, 108BGA	10/12 bit ADC, USART, I2C, SSI, Motion control, Hibernation module
2000 Series (CAN Connected MCUs)	64 kB to 512 kB	19 kB to 96 kB	25MHz to 80MHz	Upto 67	64LQFP, 100LQFP, 108BGA	10/12 bit ADC, USART, I2C, SSI, Motion control, CAN, Hibernation module
3000 Series (USB Connected MCUs)	16 kB to 256 kB	6 kB to 64 kB	50MHz	Upto 61	64LQFP, 100LQFP,	10/12 bit ADC, USART, I2C, SSI, Motion control, USB O/H/D, Hibernation module
5000 Series (USB + CAN MCUs)	16 kB to 512 kB	8 kB to 96 kB	50MHz to 80MHz	Upto 72	64LQFP, 100LQFP, 108BGA	10/12 bit ADC, USART, I2C, SSI, Motion control, USB O/H/D, CAN Hibernation module
6000 Series (Ethernet Connected MCUs)	64 kB to 512 kB	16 kB to 64 kB	25MHz to 80MHz	Upto 46	100LQFP, 108BGA	10/12 bit ADC, USART, I2C, SSI, Ethernet, Motion control, Hibernation module
8000 Series (Ethernet + CAN MCUs)	96 kB to 512 kB	64 kB	50MHz to 80MHz	Upto 46	100LQFP, 108BGA	10/12 bit ADC, USART, I2C, SSI, Ethernet, CAN Motion control, Hibernation module
9000 Series (Ethernet + CAN + USB MCUs)	128 kB to 512 kB	64 kB to	80MHz	Upto 72	100LQFP, 108BGA	10/12 bit ADC, USART, I2C, SSI, Ethernet, CAN USB O/H/D, Motion control, Hibernation module

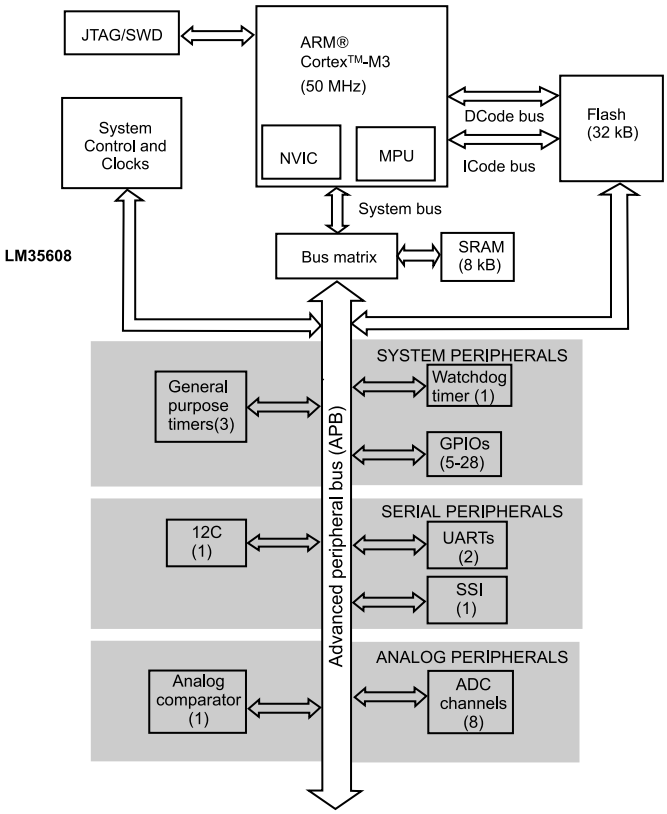

Figure 2.5 *Stellaris® diagram* (Stellaris® LM 3S608 microcontroller datasheet)

- **32-Bit RISC Performance**
  - 32-Bit ARM® Cortex™-M3 v7-M architecture
  - SysTick timer, providing a simple 24-bit clear-on-write, decrementing, wrap-on-zero counter
  - Thumb compatible Thumb-2 only instruction set
  - 50 MHz operation
  - Integrated nested vector interrupt controller
  - 23 Interrupts with eight priority levels
- **Internal Memory**
  - 32 kB single cycle flash
  - 8 kB single cycle SRAM

- **GPIOs**
  - 5-28 GPIOs, depending on configuration
  - 5 V-tolerant input configuration
  - Fast toggle capable of a change every two clock cycles
  - Programmable control for GPIO interrupts
  - Programmable control for GPIO pad configuration
- **General-Purpose Timers**
  - Three general-purpose timer modules, each of which provides two 16-bit timers/counters. Each can be configured independently:
    - as a single 32-bit timer
    - as one 32-bit real-time clock
    - for pulse width modulation.
- **ARM® FiRM-compliant Watchdog Timer**
- **Analog to Digital Converter**
  - Eight analog input channels
  - Single-ended and differential input configurations
  - On-chip internal temperature sensor
  - Sample rate of 500,000 samples/second
- **UART**
  - Two fully programmable 16C550-type UARTs
  - Separate 16×8 transmit (TX) and receive (RX) FIFOs to reduce CPU interrupt service loading
  - Fully programmable serial interface characteristics
    - 5, 6, 7 or 8 data bits.
    - Even/odd/stick or no-parity generation
    - 1 or 2 stop-bit generation
- **Synchronous Serial Interface**
  - Master or slave operation.
  - Programmable clock bit rate and operation
  - Programmable data frame size from 4 to 16 bits
- **Inter-Integrated Circuit**
  - 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
  - Two transmission speeds, standard (100 kbps) and fast (400 kbps)
  - Master and slave interrupt generation

- **Analog Comparators**

  - One integrated analog comparator
  - Configurable for output to drive an output pin, generate an interrupt or initiate an ADC sample sequence

- **Power Management**

  - On-chip low drop (LDO) voltage regulator with programmable output user adjustable from 2.25 V to 2.75 V.
  - Low power options on the controller: Sleep and Deep Sleep modes
  - Low power options for peripherals: software controls shutdown of individual peripherals
  - 3.3 V 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 bits instruction register (IR) chain for storing JTAG instructions
  - IEEE standard instructions: BYPASS, IDCODE, SAMPLE/PRELOAD, EXTEST and INTEST
  - Integrated ARM® serial wire debug (SWD)

2.6.2 Pin Out

The LM3S608 used on the kit is in the 48-pin thin quad flat package (TQFP). The pin-out for the LM3S608 is shown in Figure 2.6.

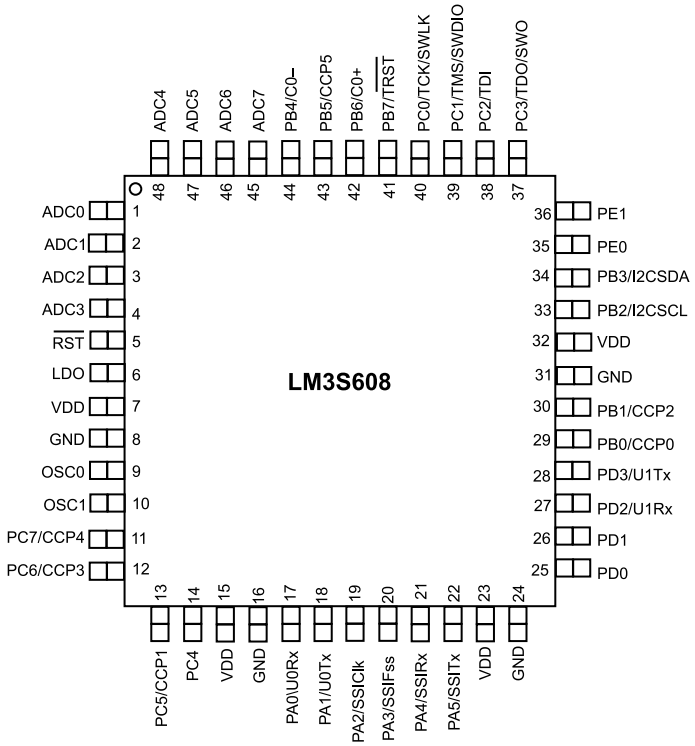

Figure 2.6 *LM3S608 pin-out (Stellaris® LM3S608 microcontroller datasheet)*

3 Stellaris® Guru Evaluation Kit

In the previous chapter we looked at the architecture of the Stellaris® microcontroller, especially the peripherals. In this chapter we look at the Guru kit in detail.

3.1 Guru Ahoy!

Now to the Guru evaluation kit! Figure **??** shows an annotated photograph of the Guru PCB. There are quite a few features on this circuit board that measures 2.7 inches by 2.5 inches.

As shown in Figure 3.1, the Guru kit has the following features

1. Uses an LM3S608 microcontroller from the TI Stellaris® 600 series
2. Offers user switches, uni-colour LEDs and an RGB LED
3. Reset push button and a power indicator LED
4. An LM35 analog temperature sensor connected to one of the ADC channels of the microcontroller
5. A thumbwheel potentiometer connected to an ADC channel of the microcontroller
6. A microphone with a high gain amplifier connected to an independent ADC channel of the microcontroller
7. An ambient light sensor using a LED operated in reverse bias
8. A standard ARM® 20-pin JTAG debug connector
9. Arduino-compatible interface connector
10. UART0 accessible through a USB virtual COM port (VCP)
11. A preinstalled boot loader. The user can download application code into the microcontroller through the USB virtual COM port.
12. The USB virtual COM port for communication and as well as for supplying power to the Guru

Some of these features are highlighted in Figure 3.1.

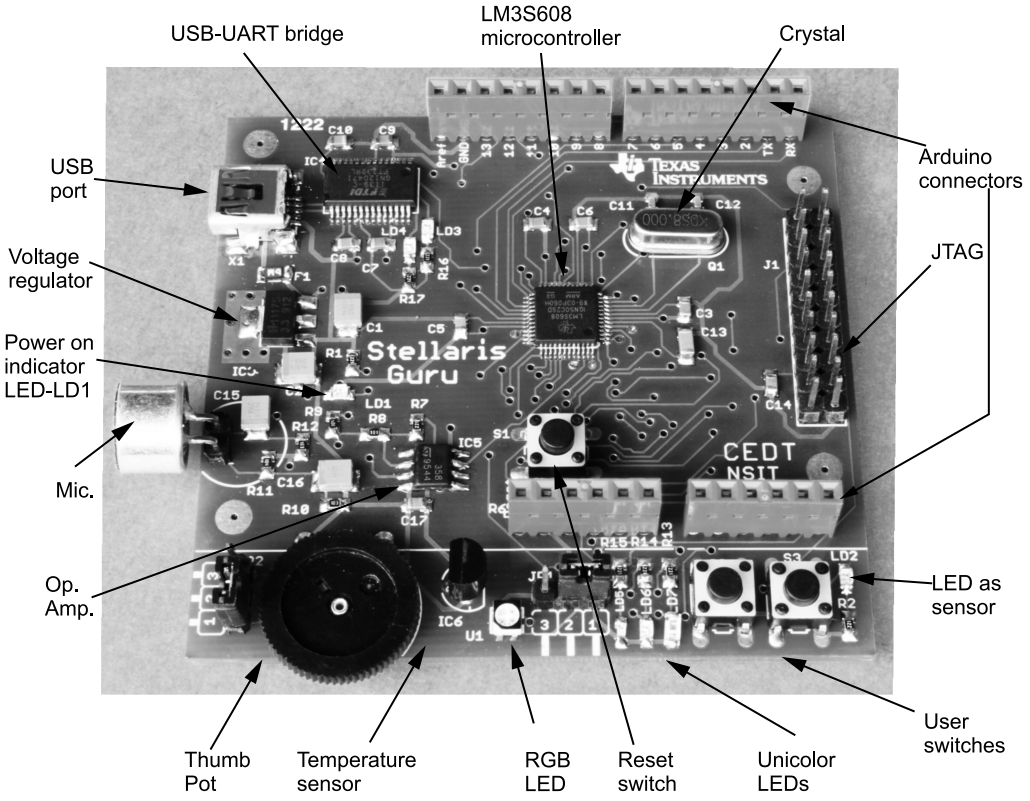

Figure 3.1 *Annotated photograph of the Guru printed circuit board*

3.2 Guru Schematic and Board Layout

Figure 3.2 shows the component placement of the Guru PCB. This figure is useful if you want to verify the placement of a particular component from the schematic diagram.

Figure 3.3 shows the complete schematic diagram of the Guru. Broadly, the sub-sections of the Guru kit include:

1. Power supply
2. Connections of the LM3S608 microcontroller to various other components
3. The JTAG interface
4. Arduino interface
5. USB interface to the personal computer
6. Microphone and audio amplifier
7. Light sensor
8. User programmable LEDs and switches
9. Temperature sensor and the thumb wheel potentiometer input

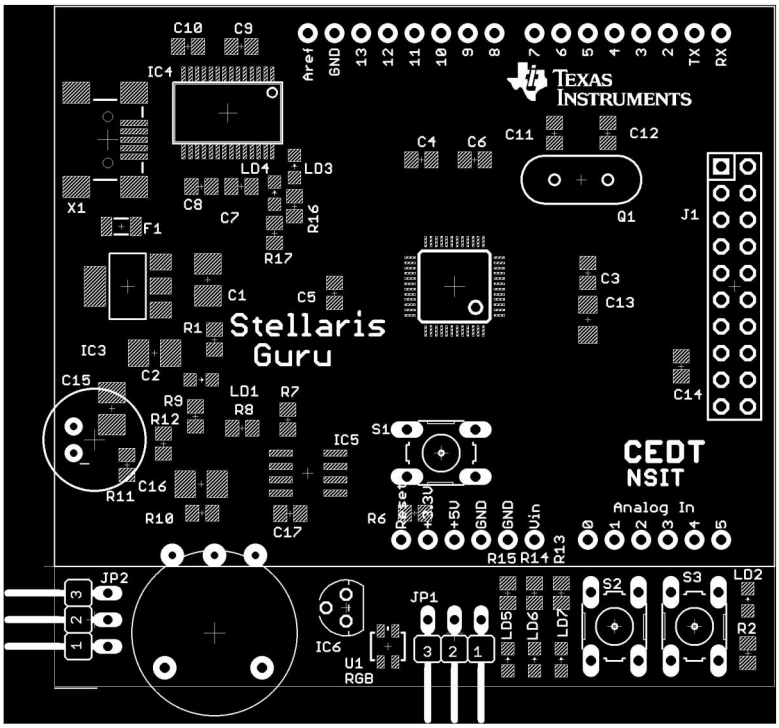

Figure 3.2 *Guru PCB component placement*

Let us now explore each and every component shown in the diagram in the next section.

3.3 Design of the Guru Evaluation Kit

The main objective of any evaluation kit is to allow a user to download test programs in the target microcontroller from the host computer with minimum difficulty. Since the evaluation kit is a standalone circuit, it needs a power supply for its operation.

3.3.1 Power Supply

Figure 3.4 shows the power supply section of the circuit. Power is derived from the USB port of the kit when it is connected to the host PC. A USB interface provides av+5 V voltage on its pins for use by the slave device to which it connects; this is used by the Guru.

The LM3S608 Cortex™-M3 microcontroller (IC1) on Guru requires +3.3 V for its operation. The power supply circuit uses a LM1117-3.3 low dropout (LDO) voltage regulator (IC3) from TI to provide the operating voltage to the microcontroller. Using an LDO in this circuit is critical since the maximum allowable drop

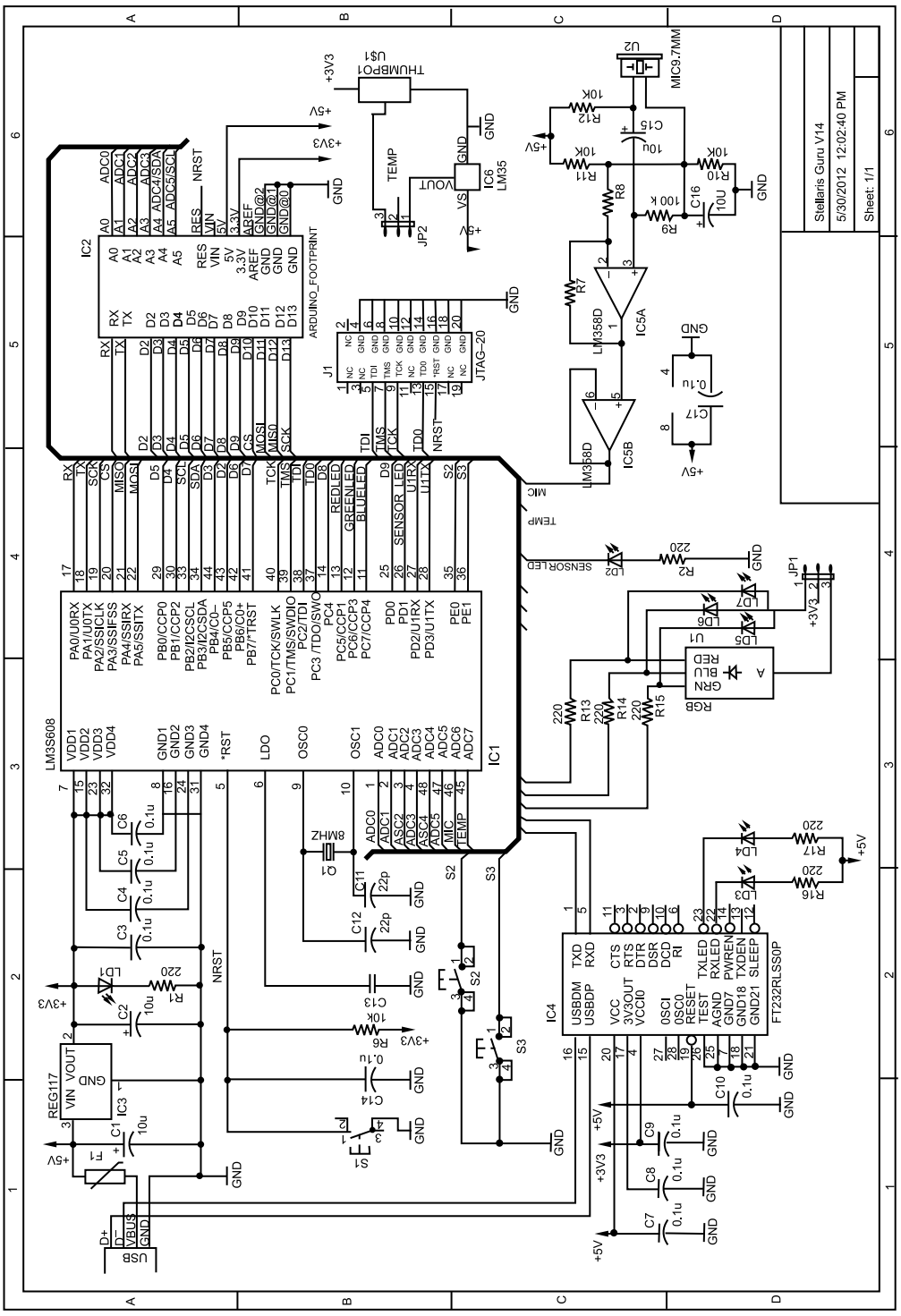

Figure 3.3 *Guru schematic diagram*

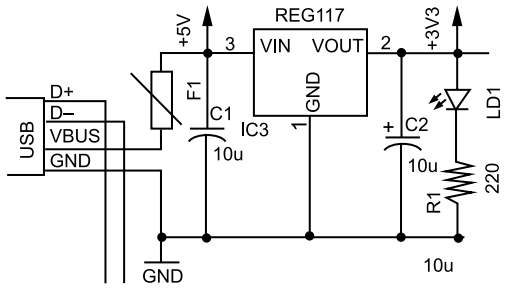

Figure 3.4 *Guru power supply section (REG117 refers to the LM117-3.3 V voltage regulator IC)*

voltage between the voltage input and output pins is 1.7 V. LM1117 series of LDO voltage regulators does a good job for the requirement here, since these regulators need only a maximum of 1.2 V dropout voltage.

According to the microcontroller datasheet, the maximum current required by the LM3S608 microcontroller is about 100 mA when it is operating at 50 MHz. The LM1117-3.3 LDO is capable of providing 800 mA current and thus meets the requirement of the microcontroller easily.

Some of the other components on the Guru such as the LM358 op-amp (IC5), the LM35 (IC6) temperature sensor and the Arduino interface connector require +5 V for their operation. The +5 V is sourced directly from the USB connector.

To protect the host PC's USB port from over-current fault condition on the Guru, a positive temperature coefficient (PTC) fuse (labelled F1 in the schematic diagram) is used. Capacitors C1 and C2 (both 10 uF/16 V Tantalum) are required by the LDO for filtering. LED (LD1) provides power-on indication.

3.3.2 Microcontroller Connections

Figure 3.5 shows the pin connections of the LM3S608 microcontroller (IC1). There are 4 power supply pins and 4 ground pins. A 0.1 uF capacitor is installed between each Vcc and ground pin for filtering the power supply. The power supply pins get the operating voltage from the LM1117-3.3 LDO.

Internally, the microcontroller has an additional LDO to power the controller's internal logic. This LDO derives power from the Vdd pins (+3.3 V) and has a nominal output of +2.5 V. This LDO requires an external capacitor of a value of 1 uF or more. A 1 uF ceramic capacitor (C13) is therefore connected to the LDO pin of the microcontroller (pin 6) as shown in the schematic diagram in Figure 3.5.

The microcontroller has several sources of reset. One of them is an external pin reset (called Reset pin assertion in the datasheet). This is Pin 5, labelled *RST. To allow a user to reset the microcontroller, a switch S1 is connected between the *RST pin and ground. A filter capacitor C14 is also connected in parallel to the switch. A pull-up resistor, R6, holds the *RST pin to logic '1' when the switch is not pressed.

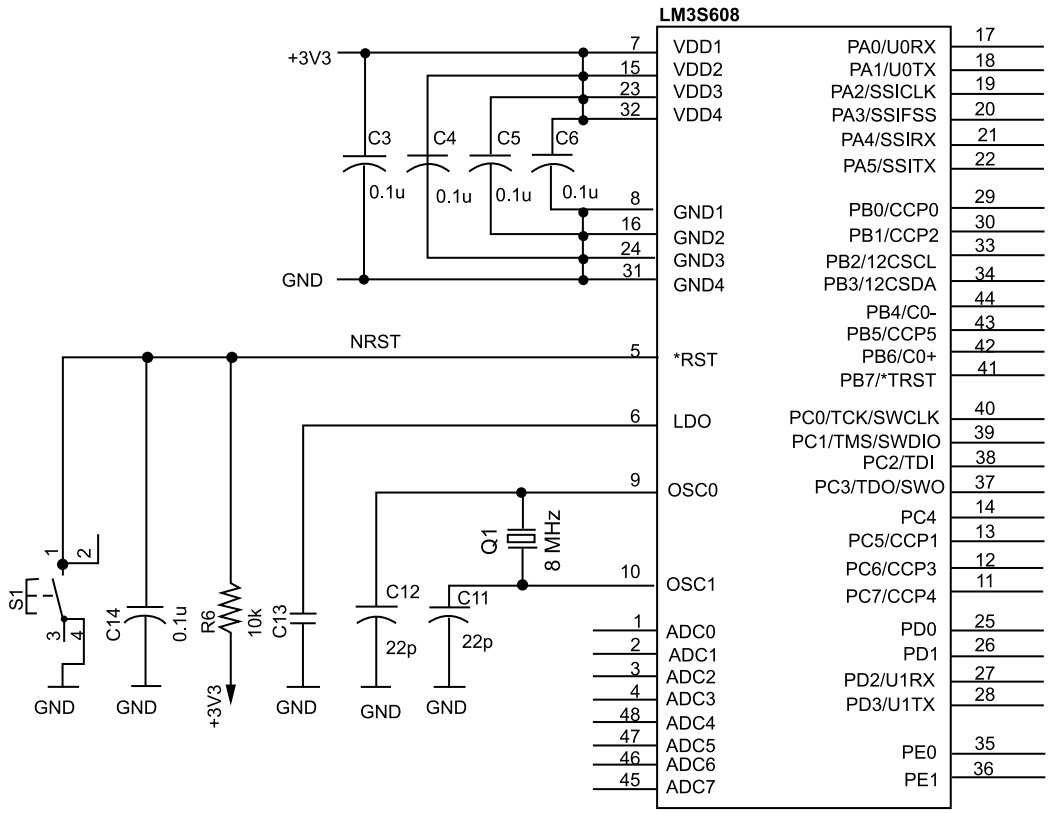

Figure 3.5 *LM3S608 Microcontroller pins on the Guru*

The OSC0 and OSC1 pins of the microcontroller, which are inputs of the main oscillator circuit of the microcontroller, are connected to a 8 MHz quartz crystal (Q1). The crystal requires two 22 pF capacitors (C11 and C12) connected to ground for reliable operation of the oscillator. The recommended value of the external quartz crystal can be any value between 1 MHz to 8.192 MHz.

The rest of the pins of the microcontroller are the port pins as shown in figure 3.5. These pins are appropriately connected to the onboard resources such as the LEDs, switches, etc., as discussed in the subsequent sections.

3.3.3 JTAG Interface

The microcontroller used in Guru offers a Joint Test Action Group[3](JTAG) port for debugging applications as well as for programming the internal flash memory. JTAG is a four-wire interface—the TDI, TDO, TMS and TCK pins are required by an external JTAG debugger/programmer to connect to the microcontroller pins.

The JTAG interface also requires access to the external reset input (*RST). Figure 3.6 shows the standard ARM® -20 JTAG connector. There are also other connector types in use, but ARM®-20 is one of the most popular ones.

Irrespective of the type of connector used, the JTAG interface signals (TDI, TDO, TCK and TMS) remain the same. These signals are shared with other peripheral functions as seen in Figure 3.6, e.g., TDI is also available as general purpose input/output (GPIO) pin mapped to PC2 port.

Serial wire debug (SWD) is, as the name suggests, an interface used for real-time debugging of the microcontroller circuits. SWD is a two-wire interface with a serial data (SWDIO) and serial clock (SWCLK). The signals used for JTAG interface are also shared with the SWD interface. Thus, the user is free to either use the JTAG or the SWD interface on the Guru board.

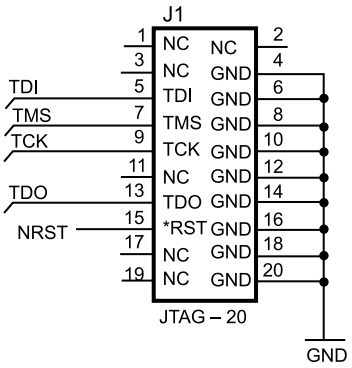

Figure 3.6 *JTAG interface to the LM3S608 microcontroller on the Guru*

3.3.4 Arduino Interface Connector

Arduino is an open source microcontroller learning and experimentation kit that allows non-specialists to use and access microcontrollers in cross-disciplinary applications. The hardware consists of an Atmel AVR microcontroller with four interface connectors for the power supply, input, output and analog pins. The board is programmed through a USB interface and the software consists of a wrapper software running on top of a GCC compiler for AVR, and an IDE (Integrated Development Environment) on the host PC to write, compile and download the programs into the target AVR chip which has a resident custom bootloader. The interface connectors consist of two 8-pin connectors and two 6-pin connectors and are arranged as shown in Figure 3.7.

The Stellaris®Guru kit uses a FT232RL (IC4) USB-to-serial bridge connected between UART0 of the microcontroller and the USB to field program the microcontroller over UART using the boot loader program (which is factory-programmed into the microcontroller).

The USB port can also be used to send data from the microcontroller to a connected host PC. Appropriate code on the microcontroller can simply transfer data through the

UART0 port of the microcontroller, which is converted to USB protocol by the FT232RL USB-to-serial bridge IC. A connected PC can receive the data using one of several terminal emulation application programs such as Hyper Terminal, Bray++, etc[4]. The connections between the microcontroller and the FT232RL are shown in Figure 3.9.

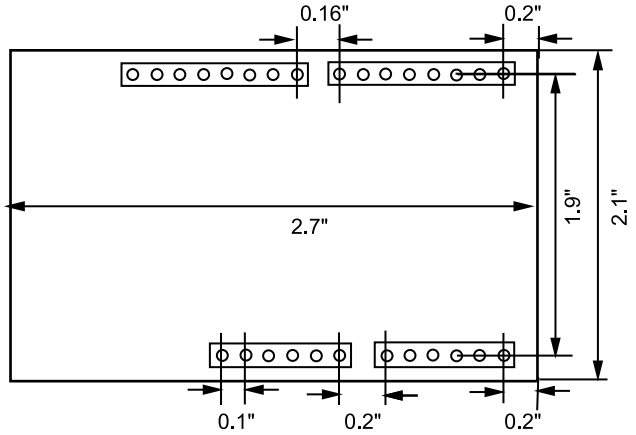

Figure 3.7 *Dimensions of the Arduino PCB and the placement and spacing of the interface connectors. The actual Arduino PCB outline is more curvy than shown here.*

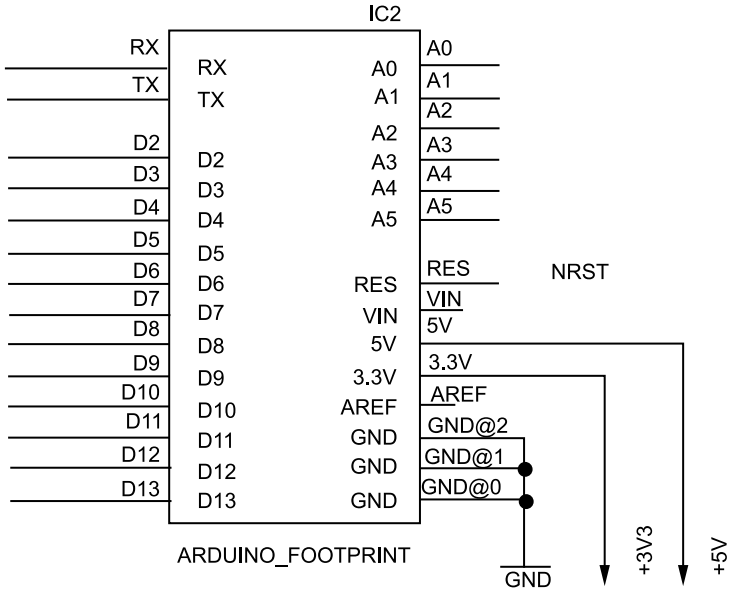

Figure 3.8 *Arduino interface on the Guru*

3.3.5 USB Virtual COM Port

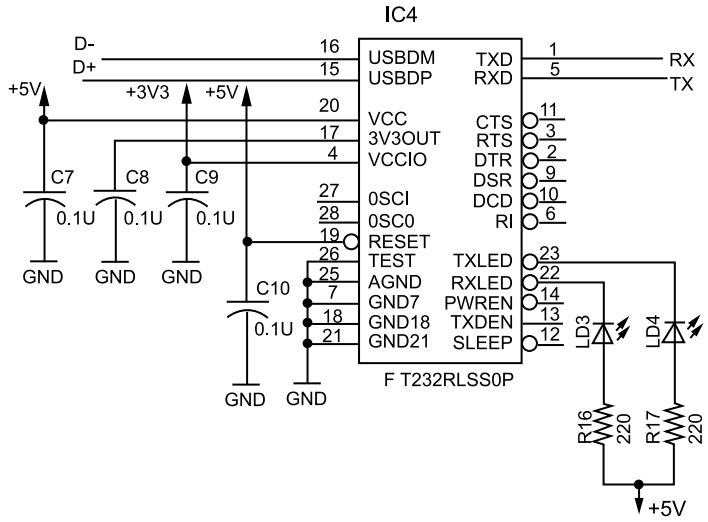

Figure 3.9 *Guru USB connection*

3.3.6 Audio Input

The audio amplifier is a two-stage, ac-coupled microphone amplifier. We have used a capacitive microphone that requires a biasing voltage of +5 V supplied through R12. The supply voltage to the LM358 is also +5V and is taken directly from the USB bus. LM358 is a two-stage operation amplifier. The first stage uses one op-amp in non-inverting configuration while the second stage is a voltage follower.

Resistors R10 and R11 provide a bias voltage equal to half the supply voltage to input of the first op-amp. The DC voltage to the op-amp is blocked by the coupling capacitor C15 while the AC voltage from the microphone is passed by it and the signal is amplified with a gain determined by resistors R7 and R8. The output of the LM358 is connected to ADC6 of the microcontroller.

The schematic diagram of the audio amplifier is illustrated in Figure 3.10.

3.3.7 Light Sensor

The Guru kit uses a LED as a sensor (LED2) to measure light intensity. The use of a normal LED as a light sensor may seem novel but there is a lot of literature available that explains the operation of a LED as a light sensor. To confirm the operation of a LED as a sensor, the reader is encouraged to take a normal LED and connect the terminals to a digital voltmeter and expose the LED to ambient light. The voltmeter should show a voltage reading which may be as much as 1.2V in bright daylight.

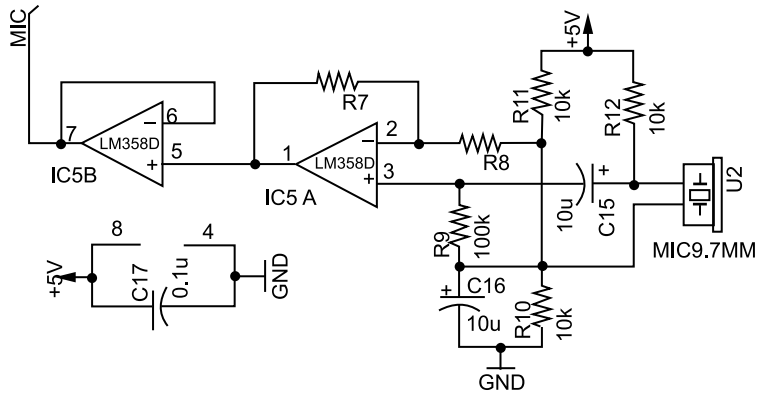

Figure 3.10 *Microphone amplifier on the Guru*

The model of an LED as a sensor[5] consists of an internal capacitance in parallel to an internal current source. The value of the current source is directly proportional to the ambient light falling on the LED. To operate the LED as a sensor, it is connected to a programmable pin of the microcontroller and is reverse biased (i.e. the anode is connected to ground and the cathode is connected to the pin) by setting the pin to logic '1'. This charges the internal capacitance to the voltage associated with logic '1' (in the present case, around 3.3V). After a short duration in this state, the pin to which the LED cathode is connected is converted to an input pin (through a program executing on the microcontroller).

The internal current source parallel to the capacitor now discharges the capacitor. The rate of discharge depends upon the strength of the current source, which in turn depends upon the ambient light to which the LED is exposed. An internal timer of the microcontroller (or a program loop) is used to measure the time it takes for the capacitor voltage to discharge (from the original logic '1' voltage) to logic '0'. This time is proportional to the ambient light. Thus a normal LED with the help of the simple resources on a microcontroller can be used to measure ambient light.

The circuit is shown in Figure 3.11. The resistor R2 in series with the LED is only for protection and does not affect or assist the measurement process.

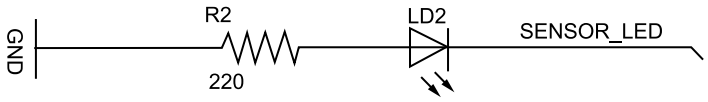

Figure 3.11 *Guru light sensor based on an LED*

3.3.8 LEDs and Switches

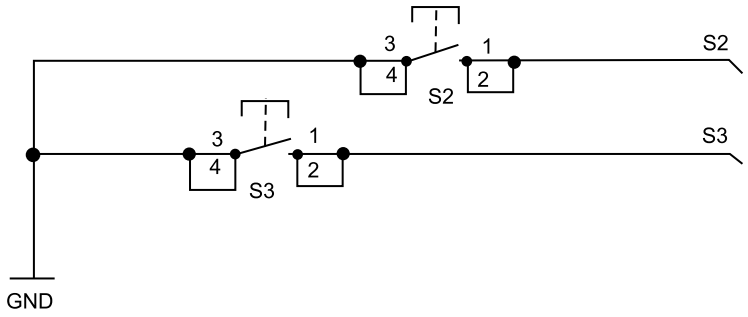

Figure 3.12 *User switches on the Guru*

The Guru kit consists of three unicolour-LEDs: LD5, LD6 and LD7, one RGB LED: RGB1 and two user pushbuttons: S2 and S3. S2 and S3 are connected to pins PE0 and PE1 respectively. The cathodes of LEDs are connected to pins PC5, PC6 and PC7 of the micro controller. Choice can be made between the three uni-colour LEDs and the RGB LED by placing an appropriate jumper on J1. The anodes of all LEDs are connected to +3.3 V. The LEDs are connected to the Capture/Compare/PWM (CCP) and hence can be used with pulse width modulation for intensity control. The connections are shown in Figures 3.12 and 3.13.

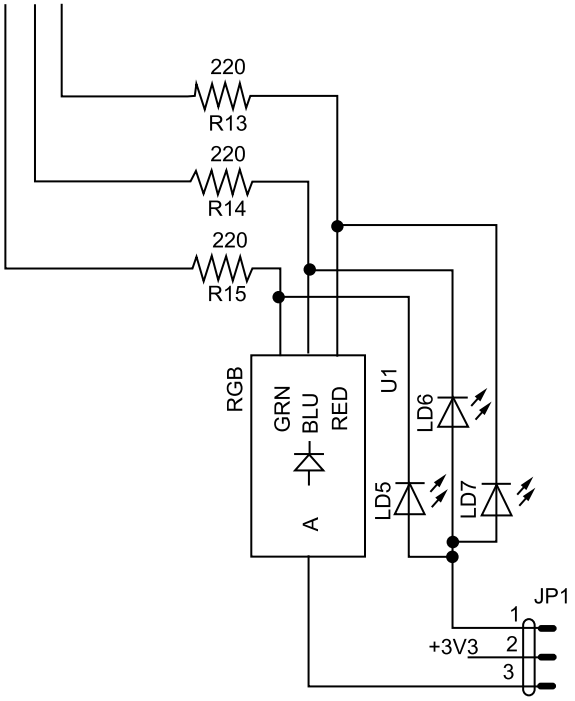

Figure 3.13 *Uni-colour and RGB LED on the Guru*

3.3.9 Temperature Sensor and Thumbwheel Potentiometer Input

To evaluate the analog to digital conversion capabilities of the microcontroller, a LM35 temperature sensor (IC6) and a thumbwheel potentiometer (POT1) are provided on the Guru board. The two are connected to the same analog channel, ADC7, of the microcontroller. Either of the two can be selected by using jumper J2. The LM35 is powered by +5V coming from the USB bus. Figure 3.14 shows the circuit connections.

3.4 Guru's Arduino Interface in Detail

An important feature of the Guru is the Arduino hardware interfiace connectors. These connectors allow a user to extend the capabilities of Guru by interfacing suitable 'shields'[6] with the required additional hardware features that are not available on the Guru board. A user can either design these required shields or use existing third party shields. A master list of such shields is available on the Internet (http://www.shieldlist.org).

However, in case users wish to design their own shields, then details of the Guru's resident microcontroller signals that are available on the Arduino style connectors would be required. Table 3.1 shows the details of these signals, their default functions as well as the additional functions these signals can offer. The first column of the table enumerates the Arduino designated signals.

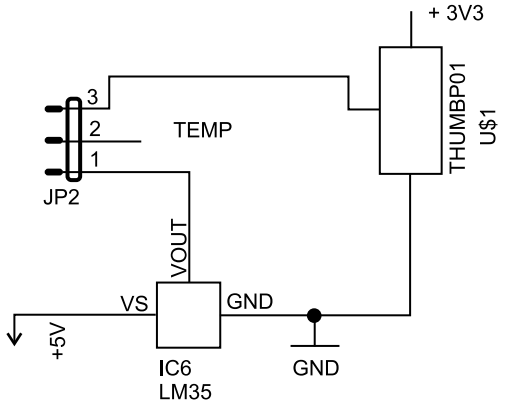

Figure 3.14 *Temperature sensor and thumb wheel potentiometer on the Guru*

The next chapter will deal with installing and setting up the required tool chain for the ARM® microcontrollers in general and the Guru in particular. These software tools include the Sourcery CodeBench Lite Edition, the Eclipse[7] Integrated Development Environment (IDE) and the LM Flash programmer. Once these tools are installed, the user is ready to write and test application programs on the Guru board.

Table 3.1 *Guru's Arduino interface pins and their functions*

Pin name	Type (Digital/ Analog/Other)	Default function	Alternate use
RX	Digital	UART Receive, PD2	Digital I/O
TX	Digital	UART Transmit, PD3	Digital I/O
D2	Digital	Digital I/O, PB5	CCP5 (Motion Control)
D3	Digital	Digital I/O, PB4	C0 (Analog comparator)
D4	Digital	Digital I/O, PB1	CCP2 (motion control)
D5	Digital	Digital I/O, PB0	CCP0 (motion control)
D6	Digital	Digital I/O, PB6	C0+ (Analog comparator)
D7	Digital	Digital I/O, PB7	
D8	Digital	Digital I/O, PC4	
D9	Digital	Digital I/O, PD0	
D10	Digital	Digital I/O, PA3	SSIFSS (SPI)
D11	Digital	Digital I/O, PA5	SSITX (SPI)
D12	Digital	Digital I/O, PA4	SSIRX (SPI)
D13	Digital	Digital I/O, PA2	SSICLK (SPI)
A0	Analog	ADC input0	
A1	Analog	ADC input1	
A2	Analog	ADC input2	
A3	Analog	ADC input3	
A4	Analog	ADC input4/SDA(I2C)	Digital I/O, PB3
A5	Analog	ADC input5/SCL(I2C)	Digital I/O, PB2
RES	Other	Reset input	
VIN	Other	DC input voltage	Not connected on Guru
5V	Other	5V supply	
3.3V	Other	3.3V supply	
AREF	Other	Analog reference voltage	Not connected on Guru
GND	Other	Ground	

4 GNU ARM® Toolchain

4.1 Introduction

The language of a barebones computer is either a '0' or '1'. Hence, a computer talks and interacts in binary numbers. You must be familiar with coding in C/C++ or Java. All these languages are known as *high-level languages* and resemble the dialect of normal written English. However, the computer cannot understand programs written in a high-level languages directly. It is the job of a compiler to translate this high-level code into a series of '0's and '1's which the computer understands.

Now, it is difficult to write code as a pattern of binary numbers. Therefore, we prefer to write a program in a high level language and then leave it to the compiler to do rest of the work for us. This chapter discusses compilers, linkers and assemblers which enable us to translate high level code into machine language. It details all the necessary steps to be followed for setting up one's computer to compile programs for the ARM® target.

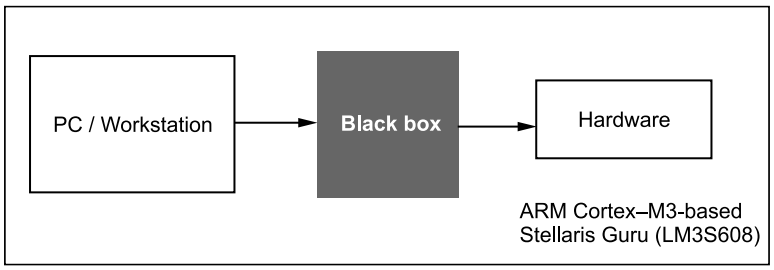

Figure 4.1 *The black box model for the Stellaris® Guru programming environment*

The black box shown in Figure 4.1 is responsible for making specific target executable files for the hardware. The primary aim of this chapter is to expose the user to tools inside this box and show how different tools come together to finally get the required binary executable format before burning it onto the Stellaris® Guru.

The contents of the black box include:

1. **Integrated Development Environment (IDE)** - The IDE is the front-end of the black box and is responsible for directly interacting with the user. It represents the editor and file/project explorer to the user.

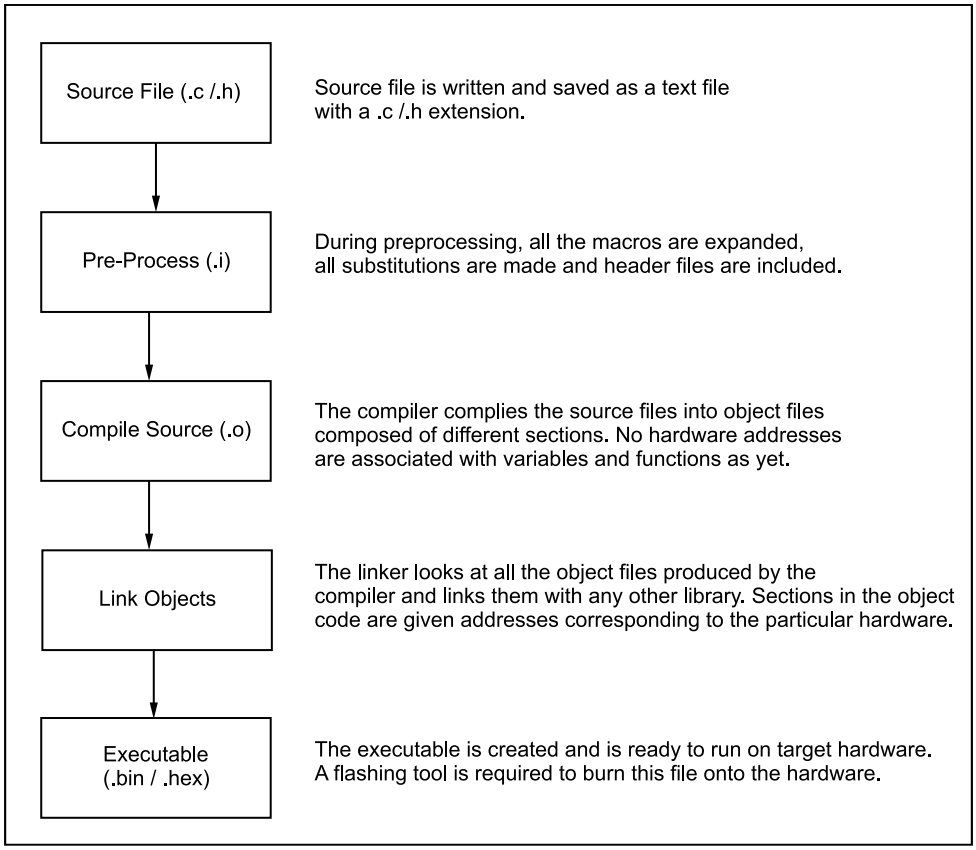

Figure 4.2 *The build process for Stellaris® Guru*

2. **Toolchain** - The toolchain is the most critical part of the black box responsible for carrying out the build process, i.e., everything from pre-processing, compiling, to assembling and finally, to linking the object modules. There are a variety of tools in a toolchain that convert the code in high level languages to the specified format of 1s and 0s depending on the arguments supplied. They are Compilers, Assembler and Linkers, besides which the Toolchain also provides a set of tools for debugging and a set of tools for converting between executable file formats. (In this manual, we willl use the C language to program the Stellaris® Guru kit.) Figure 4.2 shows the build process and Figure 4.3 the various software modules of the toolchain.

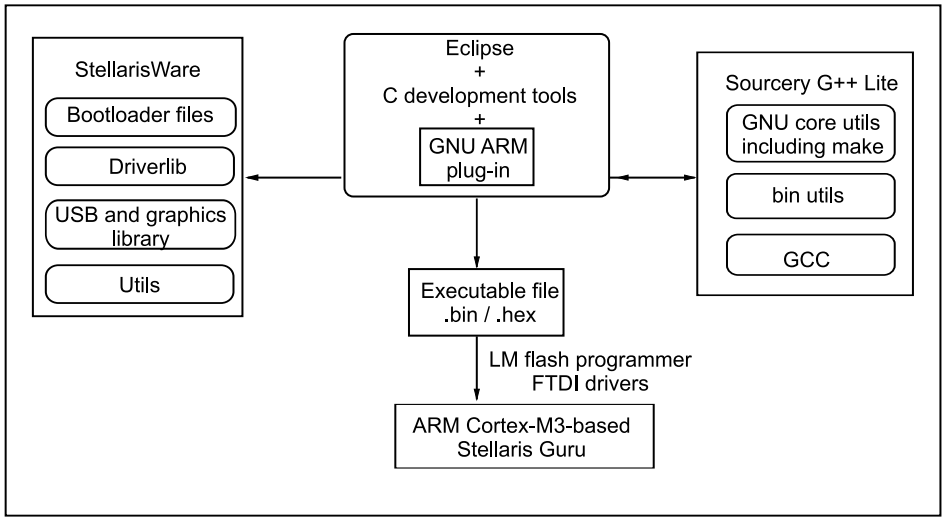

Figure 4.3 *Various software modules of the toolchain*

Outline of Steps

Pre-Processing

Pre-processing is the first step in the build process. It mainly consists of three sub-steps,

(a) Macro substitution
(b) Inclusion of header files
(c) Removal of comments

The C pre-processor, used in building C projects, is the most commonly used pre-processor. This ensures that all the pre-processor directives (`#define`, `#include`, `#ifdef`, etc.) are now expanded for the compilation process. Primarily, a .i extension is used to indicate a fully expanded source code; this may however differ in different toolchains.

```
Command: cpp [options] file ...
```

Compiling

The C compiler's job is to convert a C file into the language that the computer can understand. During compilation the compiler produces assembly code. The compiler also optimises the code, either for size or for execution speed. When an executable file is compiled for a different platform other than the one on which the compiler is running, the compilation process is called cross-compilation. Almost all development

environment for microcontrollers consists of cross-compilers since the host and target machines are actually different. All compilers have command-line arguments for specifying the target machine.

```
Command: gcc [options] file...
```

Assembling

The assembler translates the assembly program into relocatable object code (object file). With this the object program is complete except for the exact memory addresses to be assigned to the code and data sections. Object code is the sequence of (suitably encoded) machine instructions that correspond to the C instructions that the programmer has written.

```
Command: as [options] file...
```

Linking

The linker is responsible for creating the executable program by combining different object files. A user program may refer to variables and functions that are declared in some other file, say a library. When the compiler generates the object code for this program, it will have generated a reference to the external variable or function. The linker actually includes the external variables and functions as part of the executable program, so that these external references are properly resolved. The output of the linker generally has a .ld extension.

```
Command: ld [options] file...
```

3. **IDE Plug-ins** - These link the toolchain and the IDE. The toolchain can be viewed as the back-end run by a visually appealing front-end, the IDE. The IDE and the toolchain are connected by a plug-in which is responsible for connecting tools in the toolchain to the front-end.
4. **Flash Tool** - Post the build process, the binary file generated by the toolchain needs to be burned into the target board. Hardware manufacturers provide such software free of cost with their kits. Many of these software are proprietary and hide the underlying layer of communication between the hardware and the tool.

4.2 Programming Environment: Stellaris® Guru

The programming environment of the Stellaris® Guru is based on a blackbox model similar to the one discussed above. Different software are integrated to get the whole environment up and running,

- **Eclipse as IDE for C/C++ Developers**
  Eclipse is an open source project comprising an integrated development environment and a rich framework of plug-ins, all integrated to provide a complete language-independent software development environment. Eclipse 4.2 (Juno) is the latest release from the Eclipse Foundation. It provides a clean GUI-based interface for writing and editing code, and can be linked to the Sourcery GNU ARM® Toolchain using the GNUARM plug-in.

- **Sourcery G++ GNU toolchain[8] for ARM® microprocessors**
  Hosted by Codesourcery, this GNU toolchain, also known as Sourcery CodeBench Lite Edition, is available for download and use at no cost. All the tools except the newlib C Library are licensed under the GNU Public License version 3 (GPL3). The GNU C Library is licensed under the BSD License. Included in the GNU toolchain are the binary utilities (binutils), the GNU Compiler Collection (GCC), the GNU Remote Debugger (GDB), the GNU make and the GNU core utilities. The Sourcery G++ Debug Sprite for ARM® and the Codesourcery common Startup Code Sequence are licensed under the Codesourcery License. Also included in the Sourcery G++ Lite package is extensive documentation of the GNU toolchain tools, including the GNU Coding Standard document. The different commands invoked during the build process with this toolchain are:

```
arm-none-eabi-cpp [options] file... Invokes Pre-Processor
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
arm-none-eabi-gdb [options] file... Invokes Debugger
```

  Due to ARM's popularity, different vendors have different toolchains with different options to invoke the build steps. A detailed documentation accompanies each toolchain and should be read as a prerequisite to build process.

- **GNUARM Plug-in[9]**
  Developed as a part of an open source project, this plug-in helps in managing ARM® projects within Eclipse. It also integrates the entire build process of a command-line toolchain (Sourcery G++ Lite) into a visually appealing IDE. It runs on Microsoft Windows, Linux and Apple Mac OS X.

- **Texas Instruments Development Package: (StellarisWare + USB to Serial Drivers + LM Flash Programmer)**
  The Texas Instruments development package includes StellarisWare, Future Technology Devices International (FTDI) USB-to-serial drivers[10] and the LM Flash programmer. The StellarisWare library contains the Stellaris® Peripheral Driver Library, USB and Graphics Library. The FTDI drivers are used for USB-serial interaction with the target board. The LM Flash programmer is responsible for finally burning the executable onto the Guru.

4.3 Setting up the Development Environment

Make sure you have installed all the software described in the previous section before proceeding.

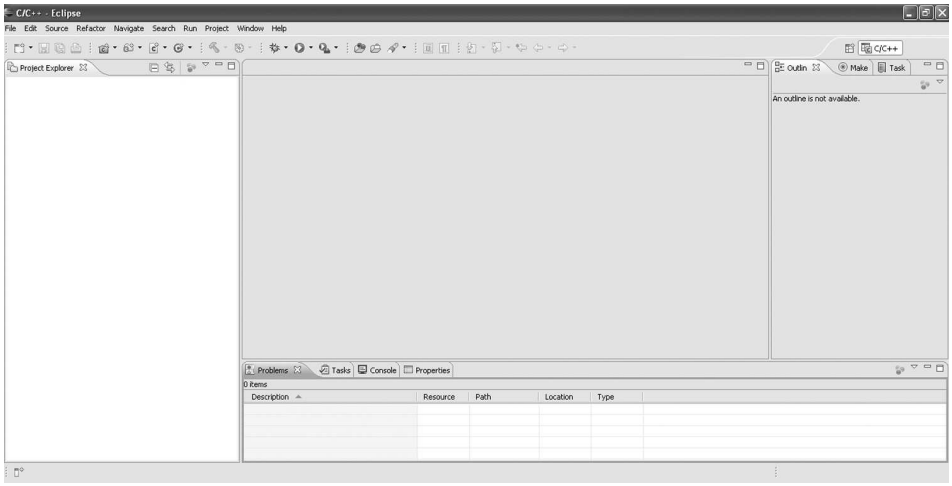

Figure 4.4 *The Eclipse IDE*

Fire up the Eclipse IDE.

A window similar to that shown in Figure 4.4 comes up.

1. Installing the GNU ARM® plug-in
   (a) Go to Help - Install New Software - Add - Archive
   (b) A window similar to that shown in Figure 4.5 comes up.
   (c) Browse to the GNUARM plug-in zip file downloaded earlier.
   (d) Click OK and Next to install the plug-in.
   (e) A window similar to that shown in Figure 4.6 will appear.

2. Creating your first project:

   (a) Go to File - New - C Project.

   (b) Enter the project name. Make sure the selections are as shown in Figure 4.7.

   (c) Click Next.

   (d) Make sure the selection is same as shown in Figure 4.8.

   (e) Then, press Finish.

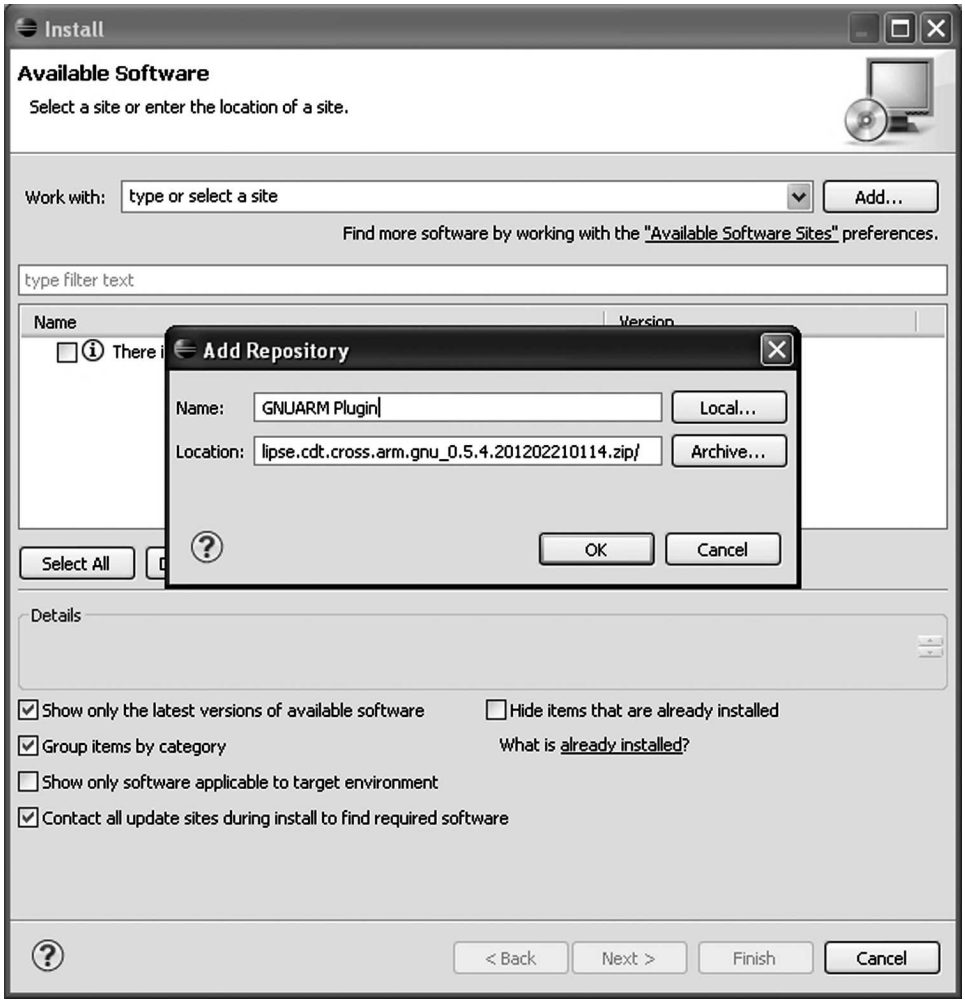

Figure 4.5 *Adding a plug-in repository to Eclipse*

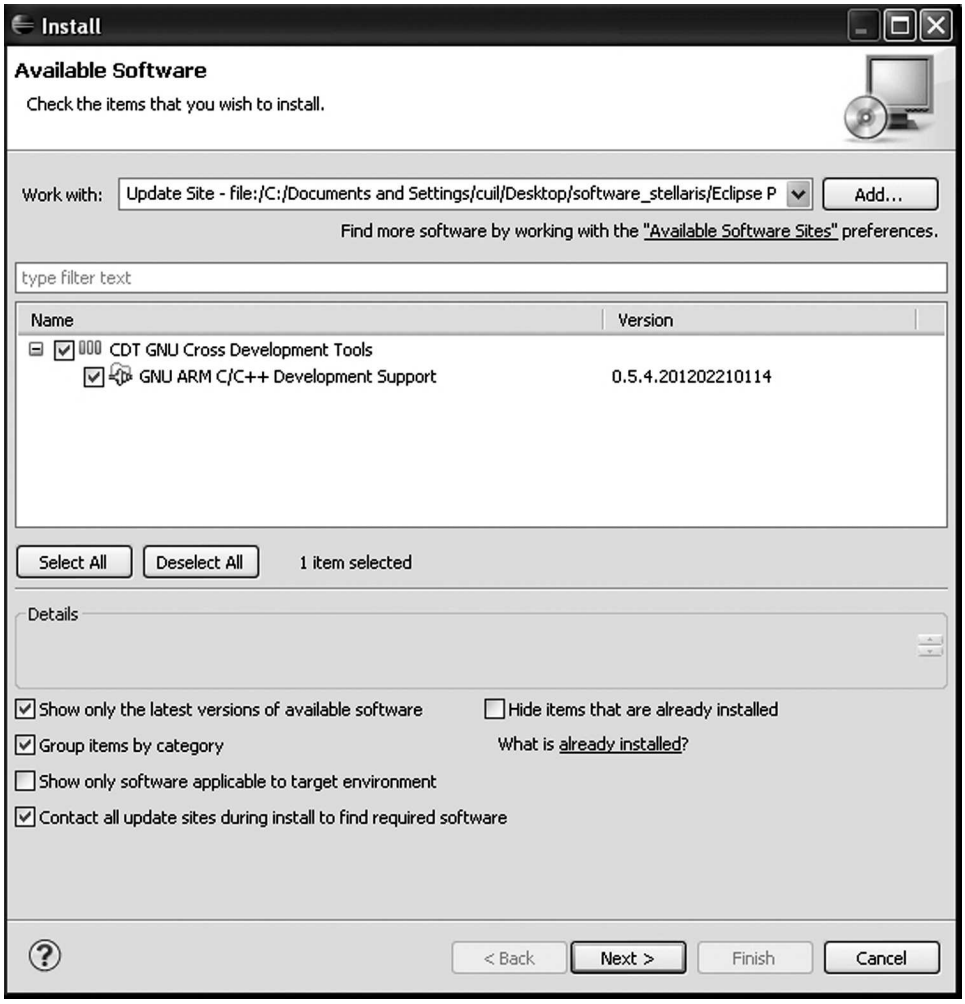

Figure 4.6 *Eclipse plug-in install options*

3. Importing the existing file system
   (a) Go to File - Import - General - File System.
   (b) A window similar to that shown in Figure 4.10 will crop up.
   (c) Press Next.
   (d) Browse and select any project from the LM3S608 folder. Make sure that the same options as shown in Figure 4.10 are selected.
   (e) Press Finish.

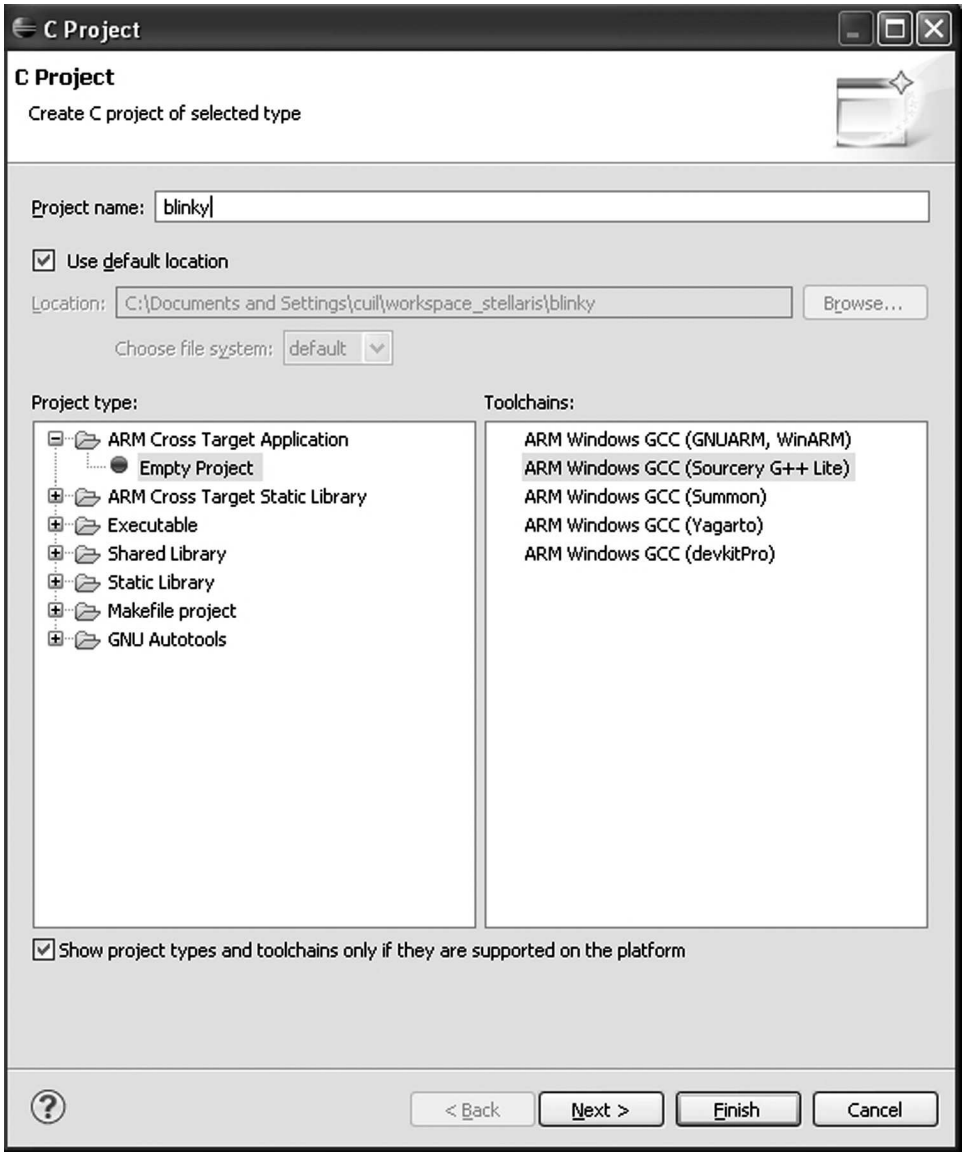

Figure 4.7 *New project window*

So, we have now set up a new project and instructed the Eclipse IDE as to which cross compiler to use for development.

Your screen should look similar to that shown in Figure 4.11.

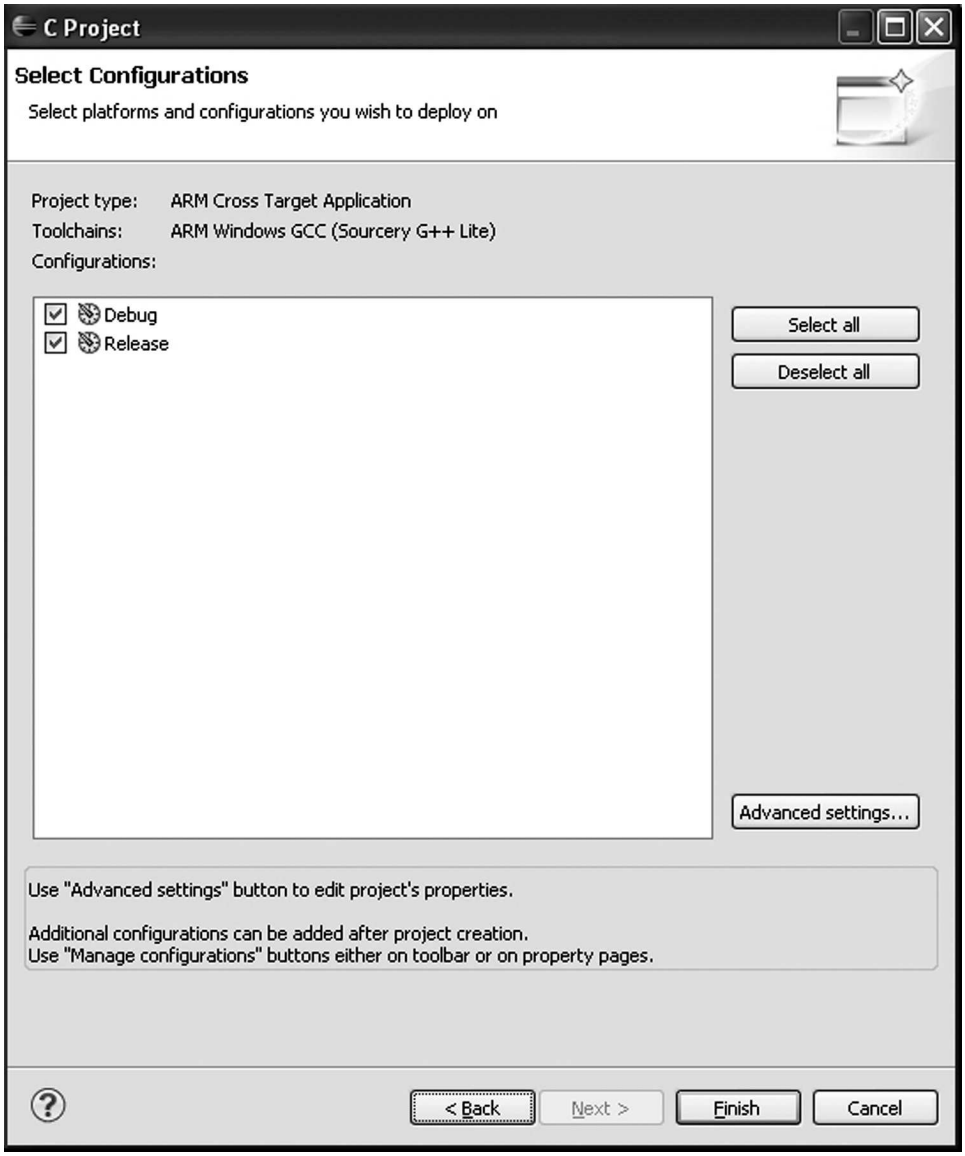

Figure 4.8 *Project configuration window*

4. Setting up project properties
   (a) Go to Project Properties.
   (b) Go to Settings - Tool Settings - ARM® Sourcery Windows GCC C Compiler - Directories.

(c) Include the directory paths as shown in Figure 4.12 and Figure 4.13.

(d) In the Project Properties window, go to Tool Settings - ARM® Sourcery Windows GCC C Compiler - Preprocessor, and define the symbol "gcc" as shown in Figure 4.14.

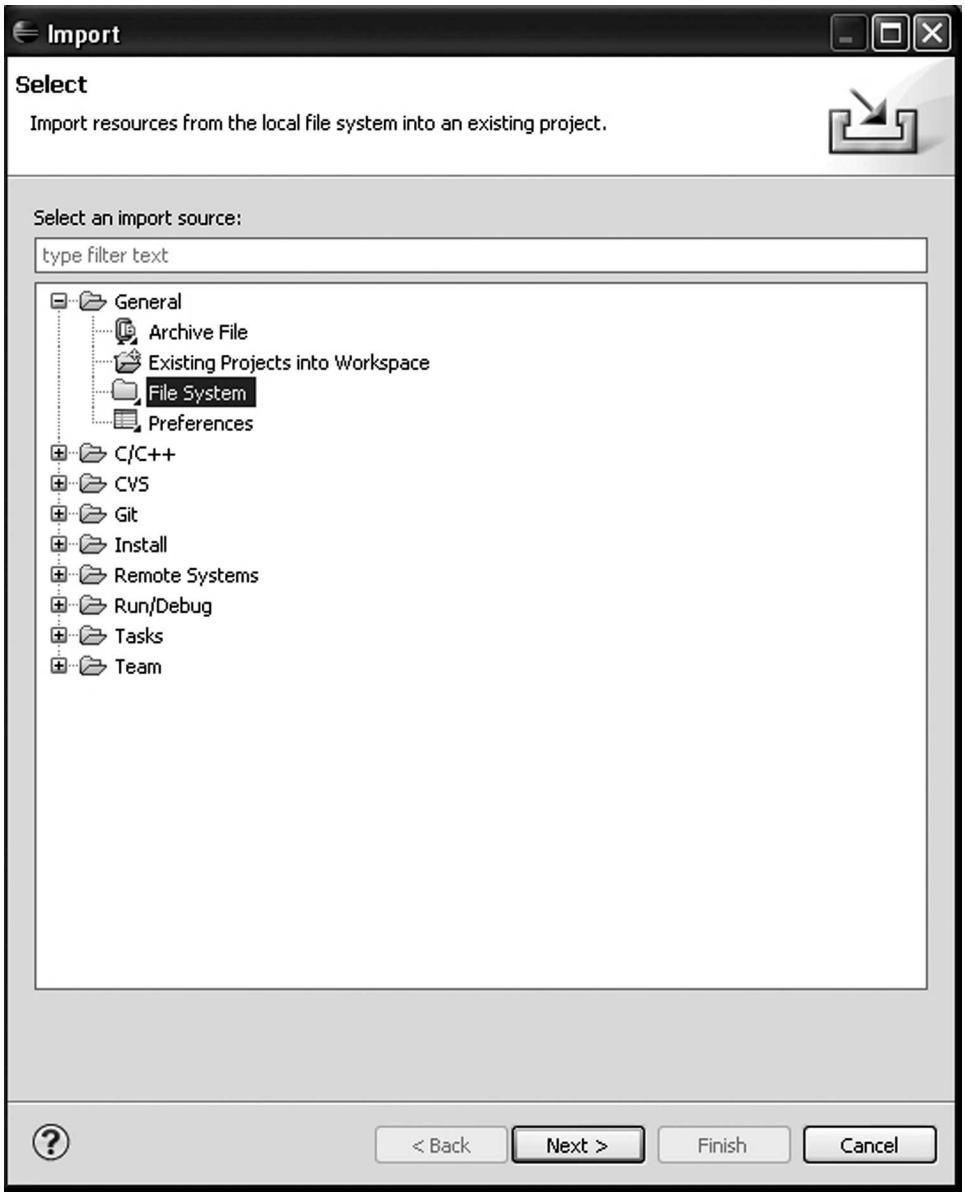

Figure 4.9 *Import window*

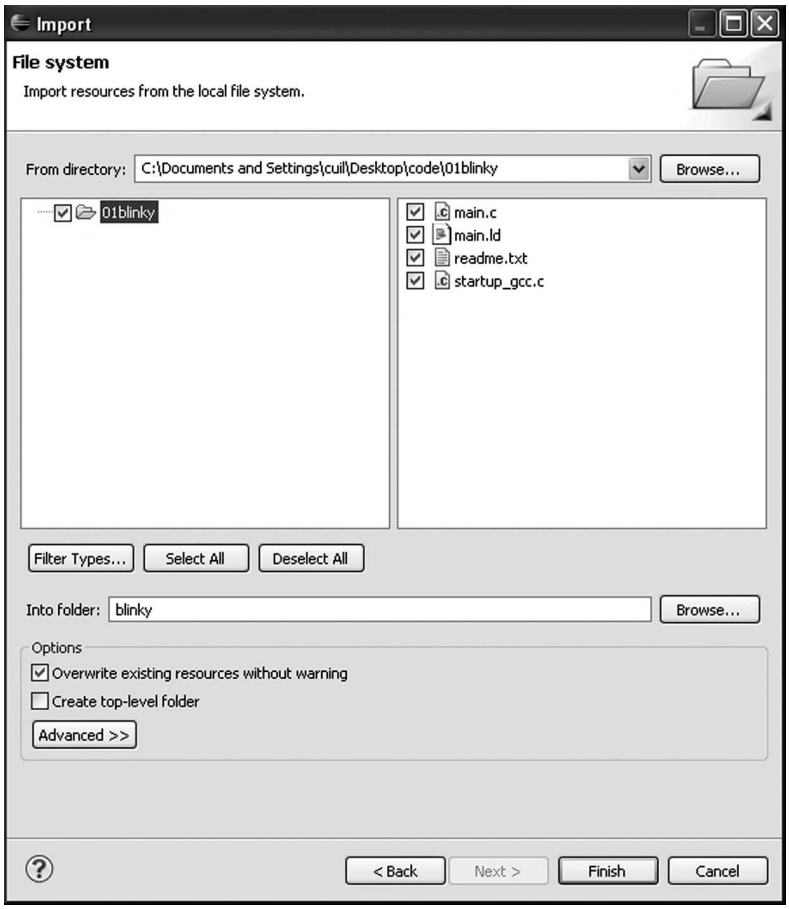

Figure 4.10 *File system import window*

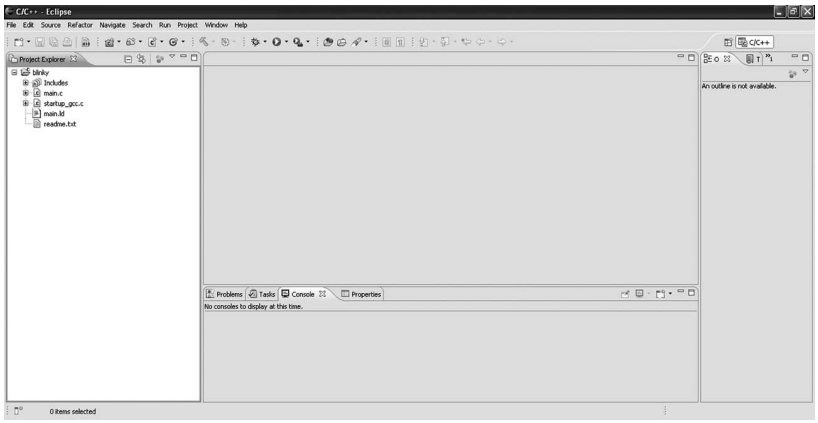

Figure 4.11 *Eclipse IDE window with imported project*

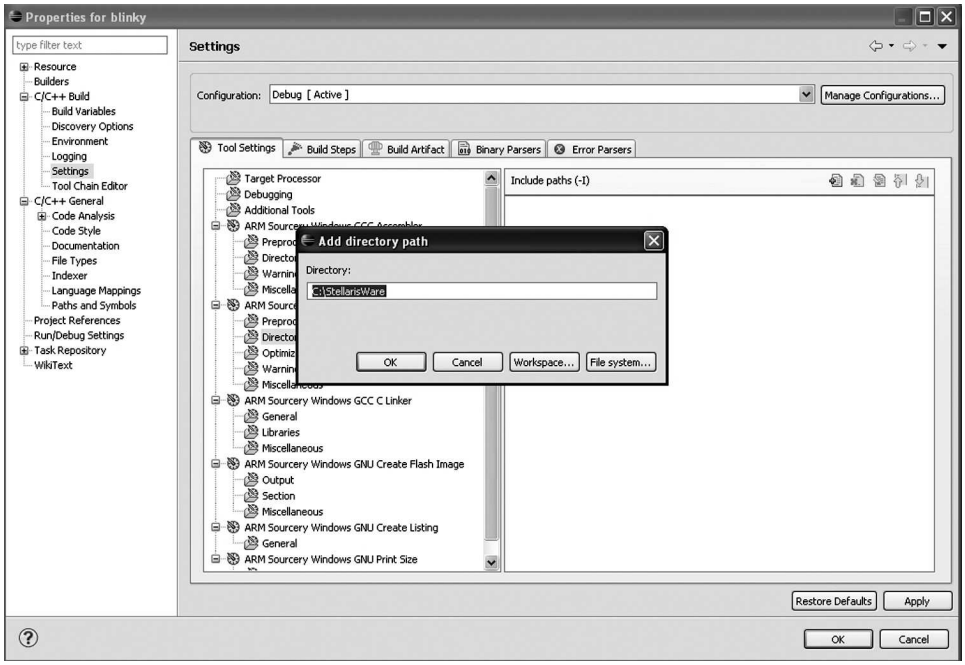

Figure 4.12 *Project properties window*

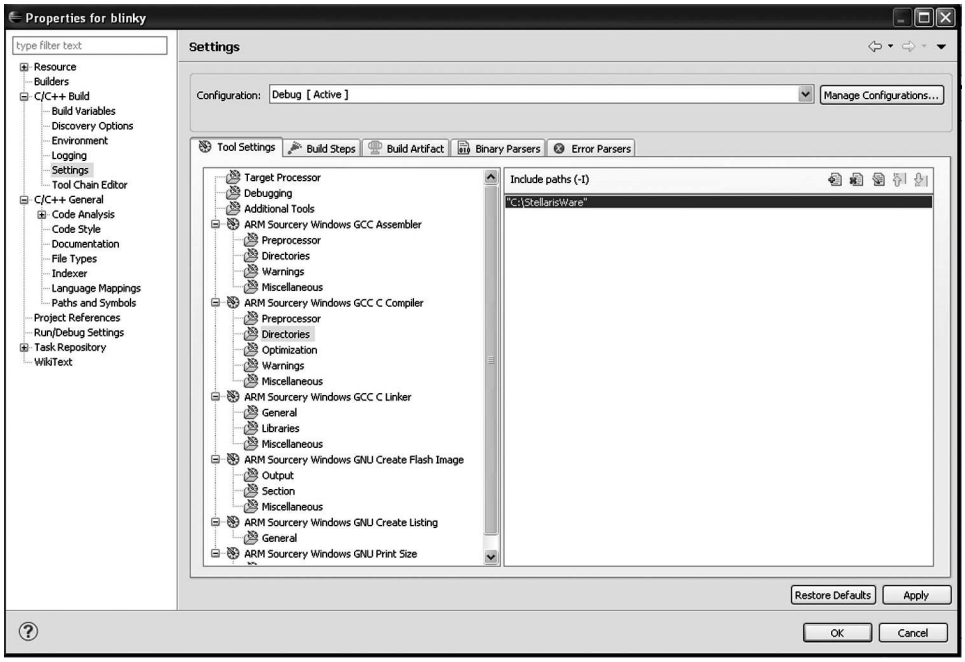

Figure 4.13 *Including StellarisWare directories*

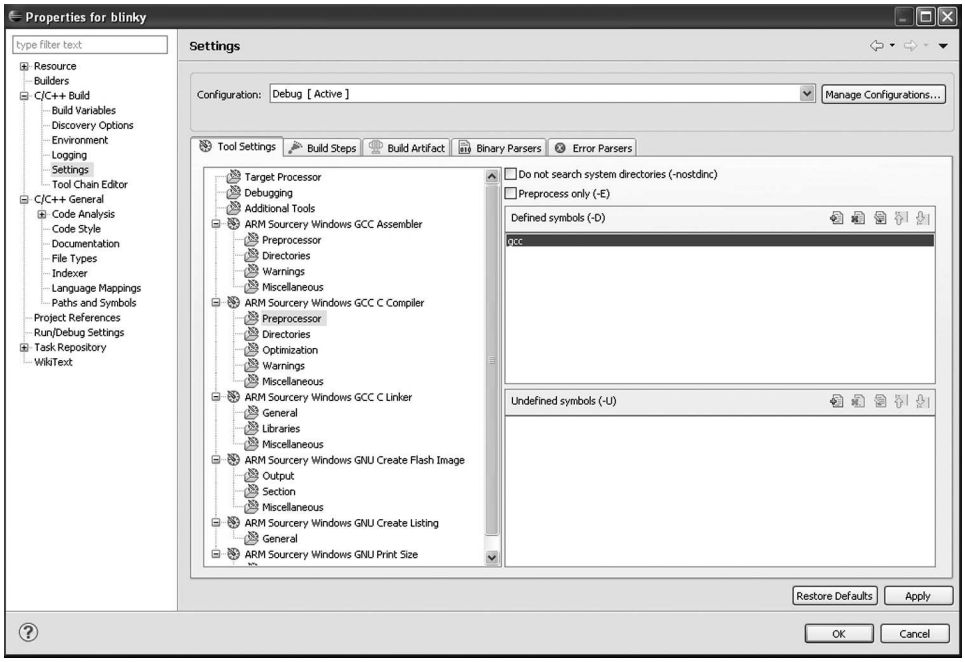

Figure 4.14 *Setting up the preprocessors*

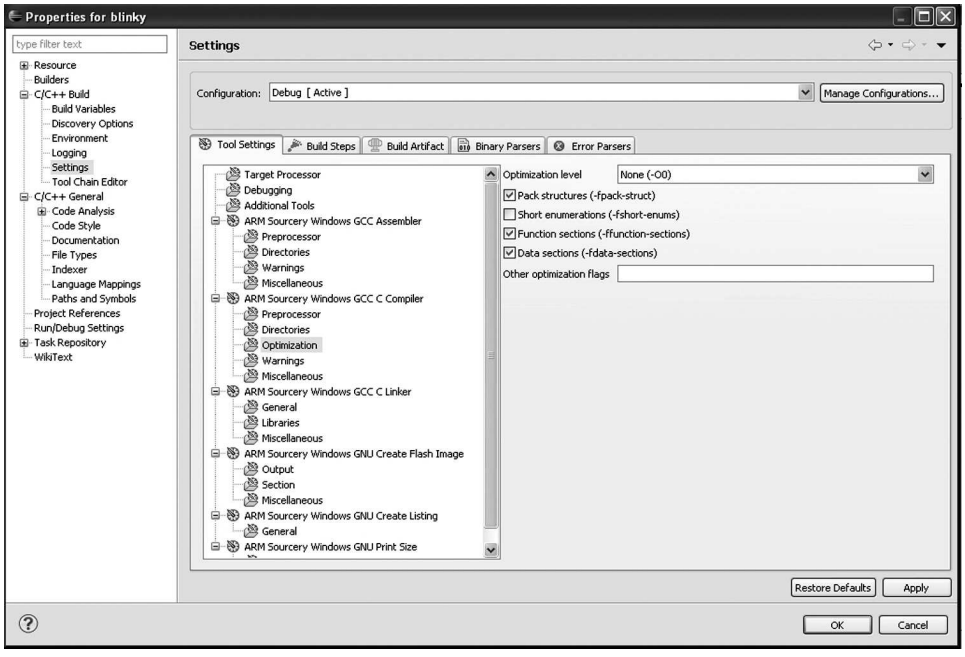

Figure 4.15 *Setting up code optimisation*

5. Setting up code optimisation
   (a) In the Project Properties window, go to Tool Settings - ARM® Sourcery Windows GCC C Compiler - Optimization.
   (b) From the drop down menu next to Optimization Levels, select "None (-O0)" and make selections as shown in Figure 4.15.

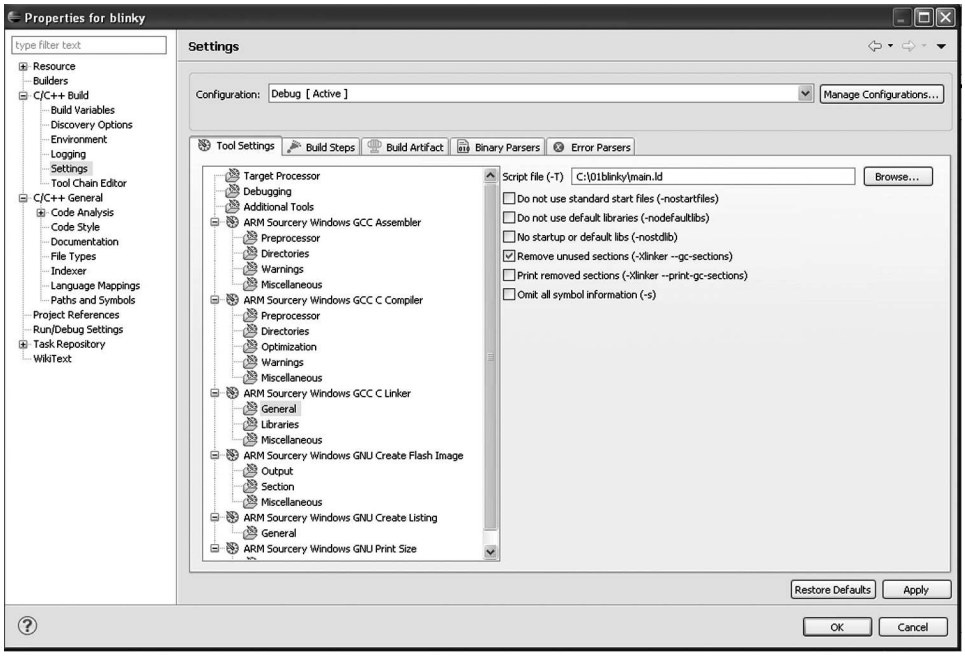

Figure 4.16 *Configuring the location of the linker file*

6. Including the linker
   (a) In the Project Properties window, go to Tool Setting - ARM® Sourcery Windows GCC C Linker - General.
   (b) Browse and select the linker file copied to the workspace. In our case, it is "main.ld".
   (c) Make sure the same options are selected as shown in Figure 4.16.
7. Adding the libraries
   (a) In the Project Properties window go to Tool Setting - ARM® Sourcery Windows GCC C Linker - Libraries.
   (b) Select Add Directory Path. A window similar to that shown in Figure 4.17 will be displayed.
   (c) Add the following directory path of your Driver Library and the library 'driver-cm3' as shown in Figure 4.18.

```
C:/StellarisWare/driverlib/gcc-cm3
```

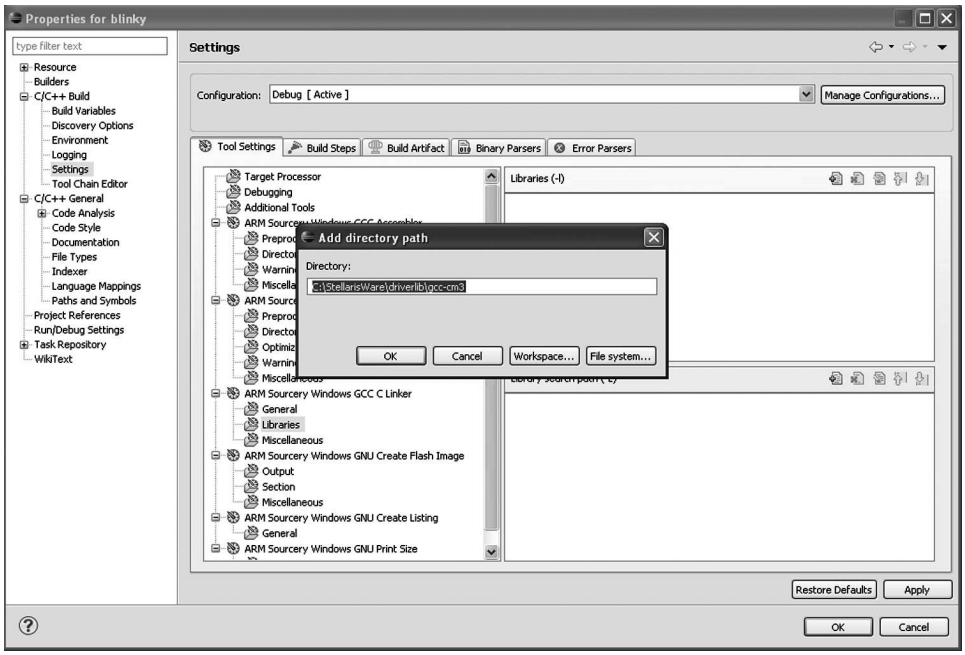

Fig. 4.17 *Adding a directory path*

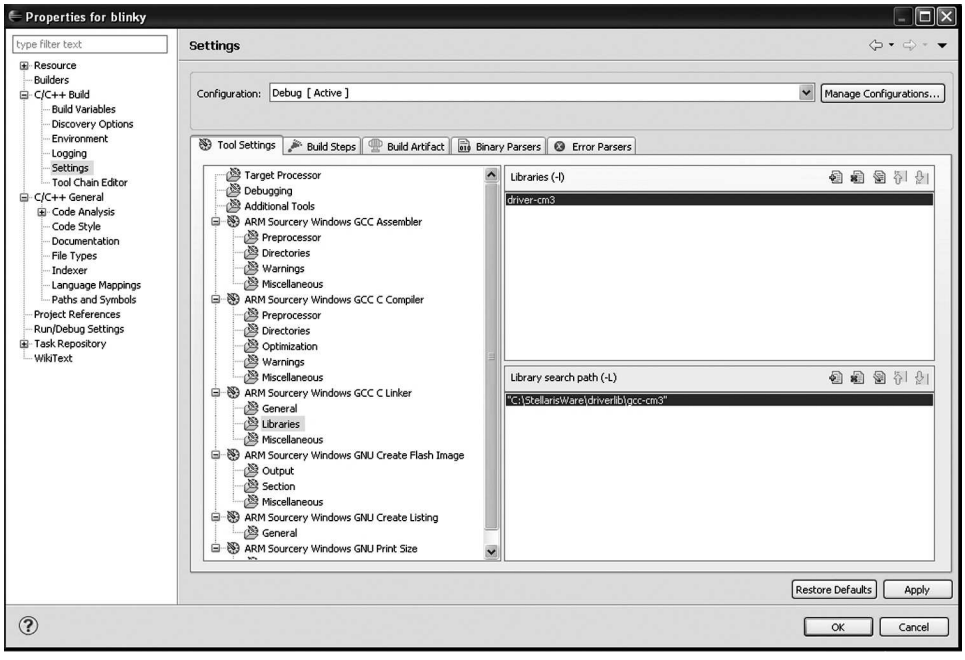

Figure 4.18 *Properties windows with directory path included*

8. Include source location
   (a) In the Project Properties window, go to C/C++ General - Paths and Symbols - Source Location - Link Folder.
   (b) Include the driverlib and utilsfolder, which contain the C source files. These are placed inside the StellarisWare folder in the C: drive. This is shown in Figure 4.19 and Figure 4.20.
   (c) Include the files as shown in Figure 4.21 and select Apply.

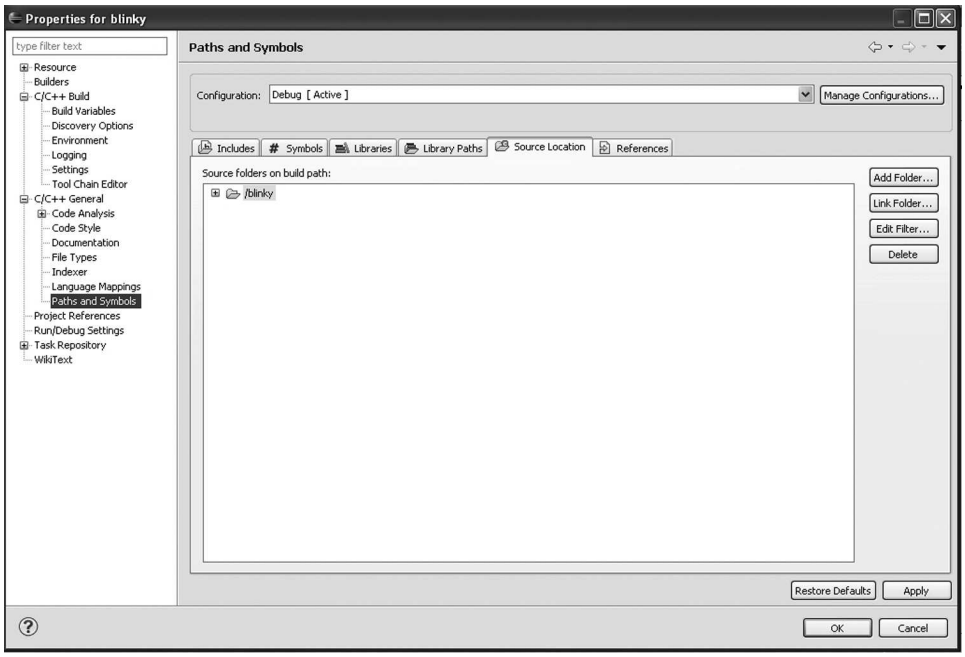

Figure 4.19 *Paths and Symbols configuration*

9. Post-build steps
   (a) In the Project Properties window, 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"
```

   (b) Your window should look similar to that shown in Figure 4.22.
   (c) Press OK.

After returning to the main Eclipse IDE window, in your workspace, select main.c. Your screen should look similar to that shown in Figure 4.23.

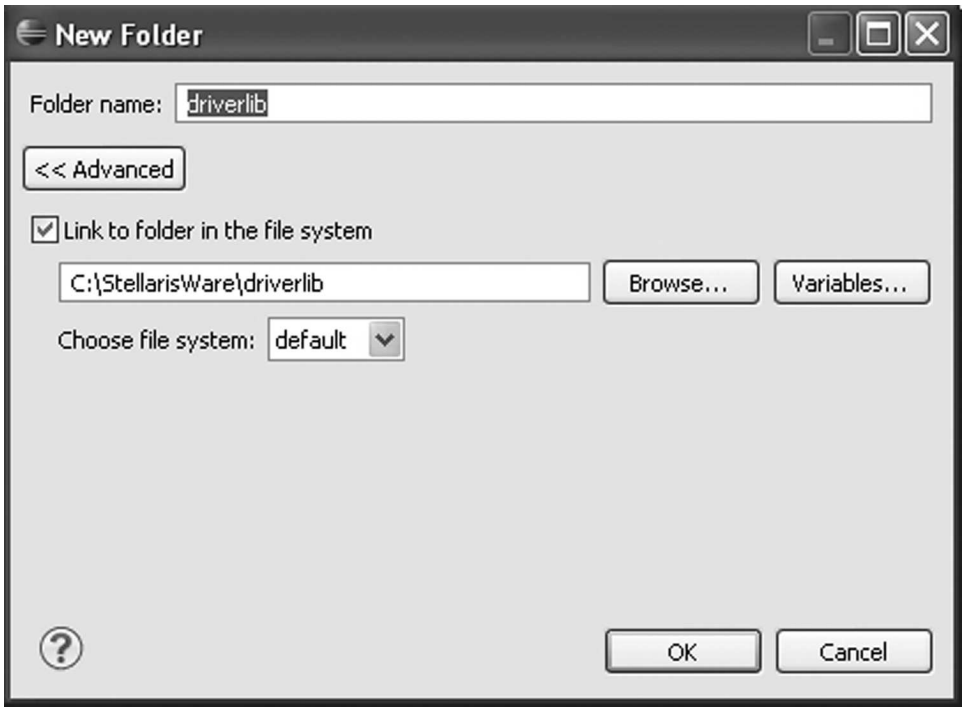

Figure 4.20 *Defining the link folder*

Figure 4.21 *Paths and Symbols window with linked folders*

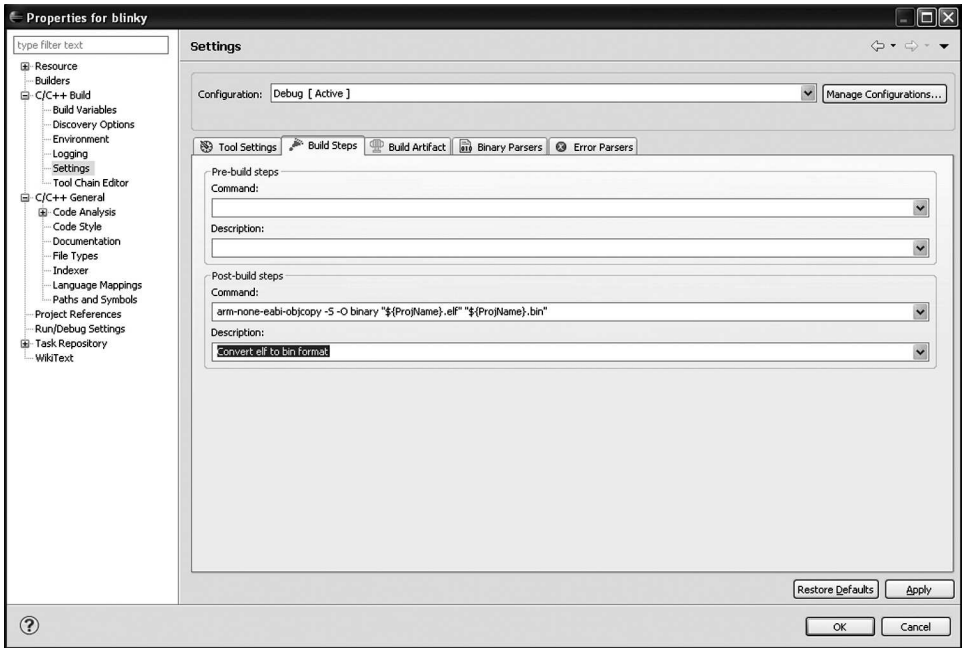

Fig. 4.22 *Post-build steps configuration*

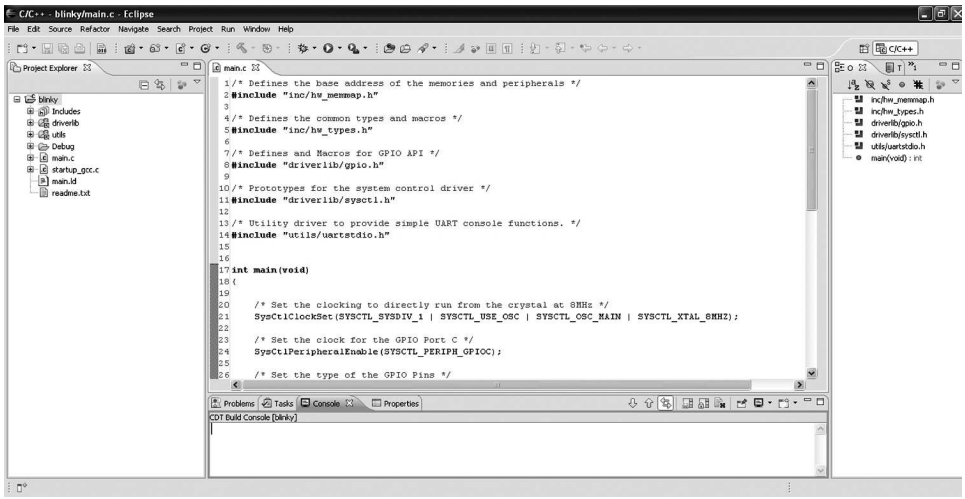

Figure 4.23 *Main Eclipse IDE window with configured project*

Right click on the project opened in the workspace, go to Build Configurations and first build the project and then clean the project shown in Figure 4.24. If everything goes right, it will build and clean with ease.

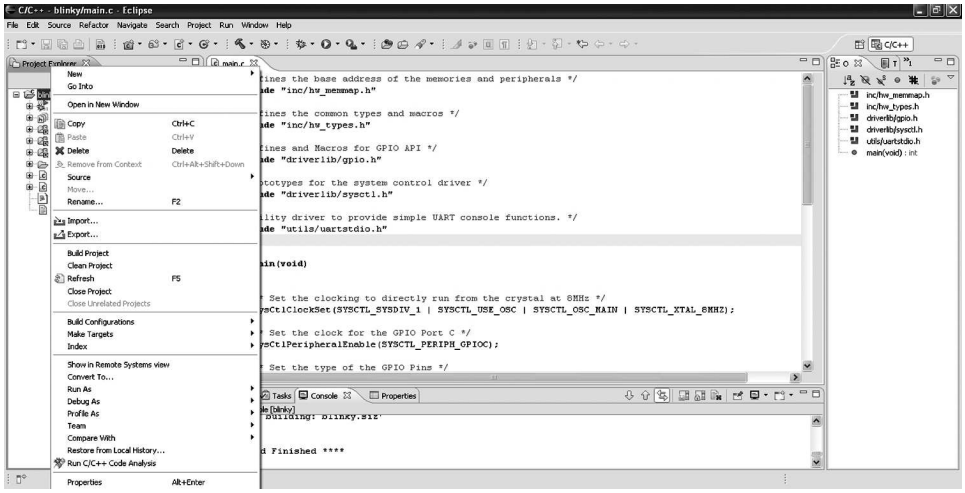

Figure 4.24 *Build and clean project*

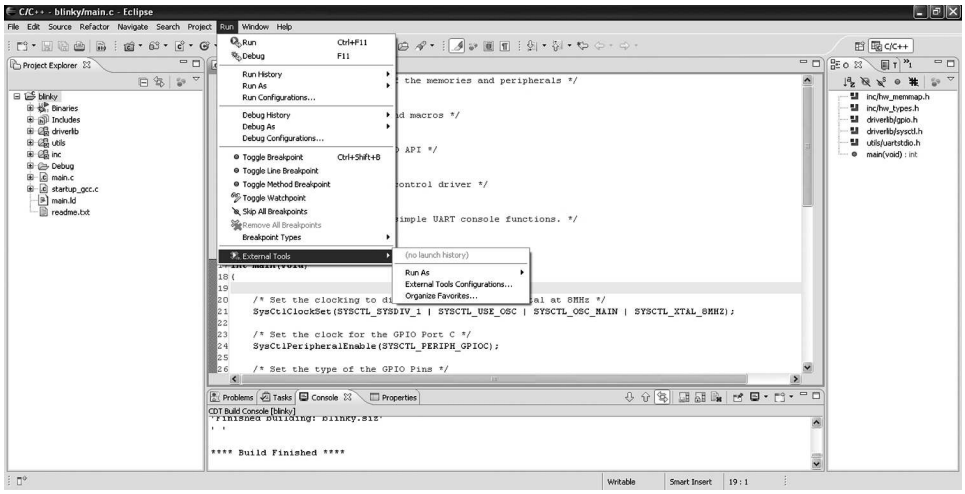

Figure 4.25 *External tool configuration*

10. Integrating LM Flash programmer with Eclipse IDE

    (a) Go to Run - External Tools - External Tools Configuration as shown in Figure 4.25.

    (b) A window similar to that shown in Figure 4.26 should be displayed.

(c) In the window, set the location to LMFlash.exe located under "C:/Program Files/Texas Instruments/"

(d) Clicking Run launches the LM Flash programmer.

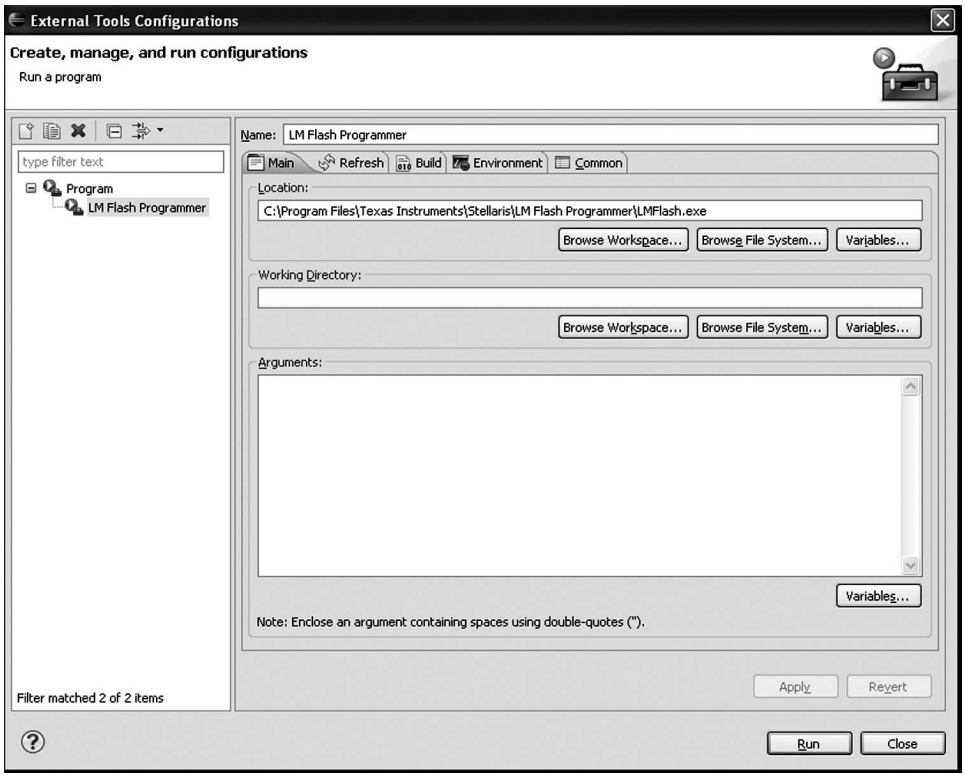

Figure 4.26 *External tool configuration*

(e) After clicking on Run, a window similar to that shown in Figure 4.25 should display on the screen. This is the LM Flash programmer.

(f) In the configuration tab, choose the COM port depending on the port the device is attached to.

(g) Go to the Program tab, and select the .bin file by browsing your project in the Eclipse workspace folder.

(h) Choose the same options as shown in Figure 4.28 with the same parameters. Make sure the address offset is 0x800.

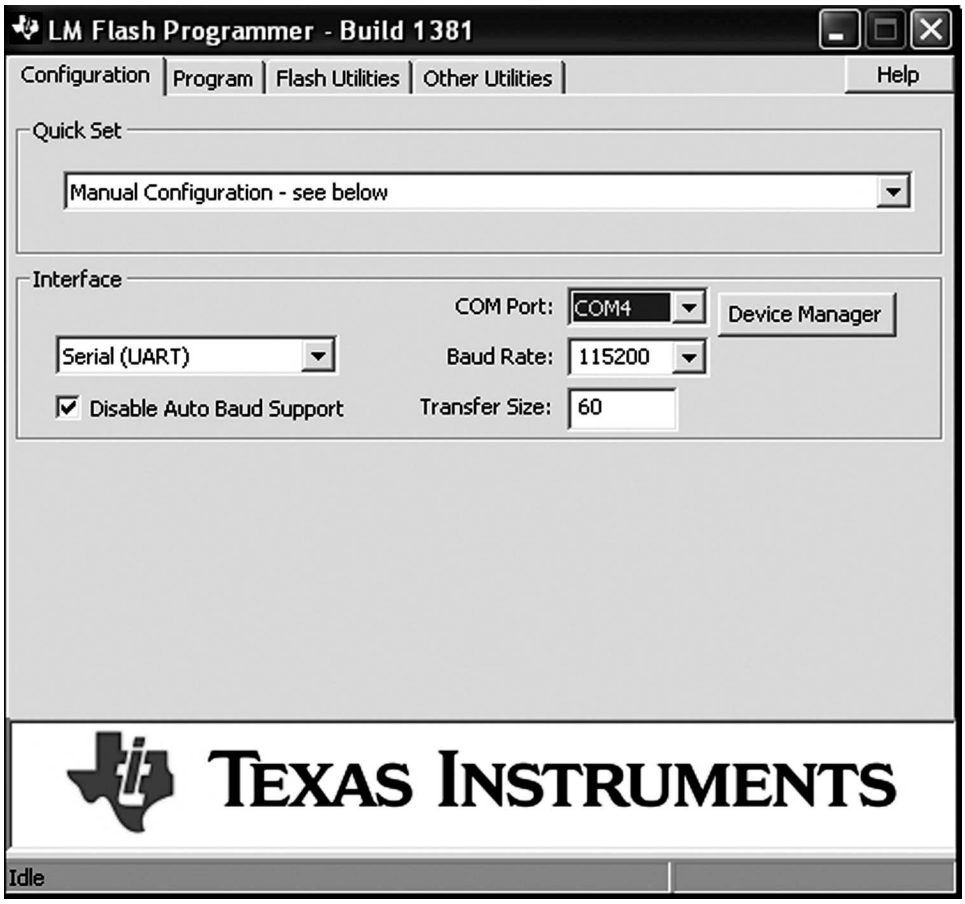

Figure 4.27 *LM Flash Programmer (Configuration tab)*

11. Uploading code to board
    (a) To upload your application program to the Stellaris® Guru kit, you need to get the kit in the "Program Update" mode. Pressing the Reset and S2 switches simultaneously and then releasing the Reset switch followed by S2 brings the microcontroller on the Stellaris® Guru kit in the program update mode.
    (b) Clicking on Program starts transferring the application binary file to the Stellaris® Guru board as shown in Figure 4.29.

If you followed the earlier steps correctly, you have just programmed your very first ARM®-powered microcontroller. Congratulations!

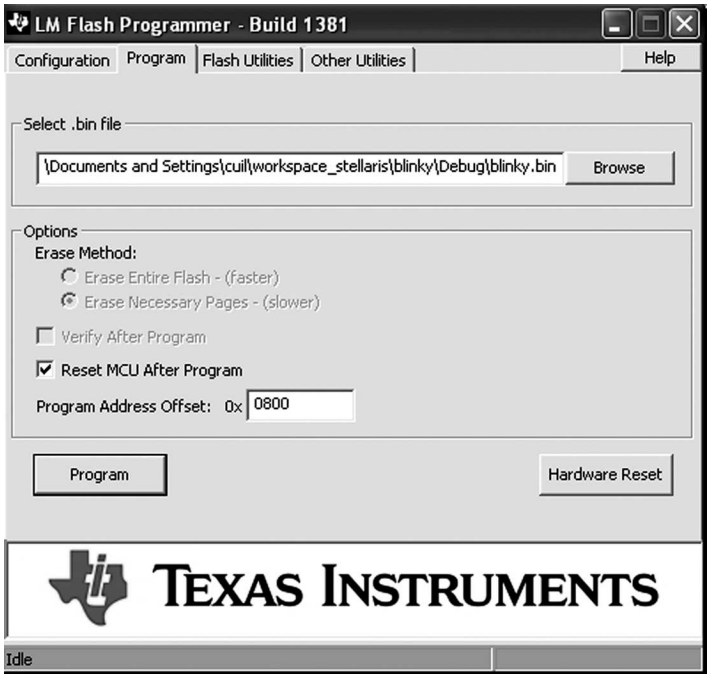

Figure 4.28 *LM Flash Programmer (Program tab)*

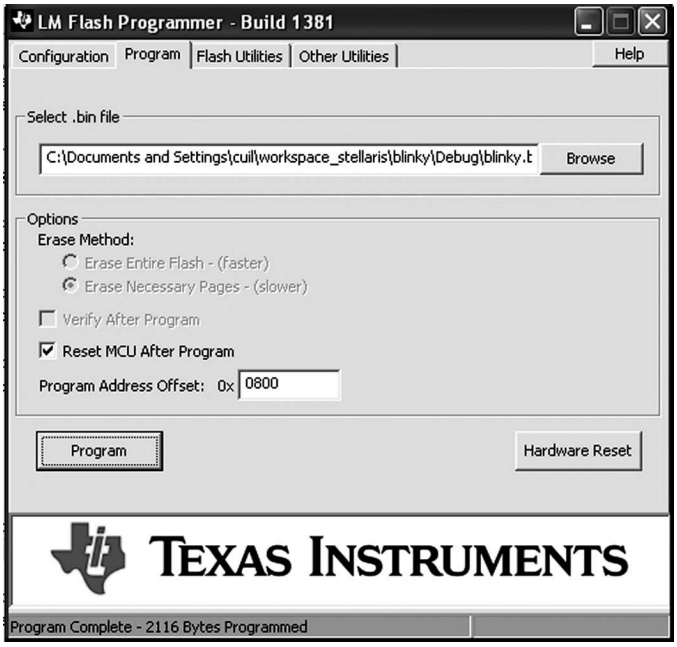

Figure 4.29 *Programming the target*

5 Anatomy of a C Program

This chapter describes the basic structure of a C program for Stellaris MCU. Although the chapter presents a general structure of a program, it can be adopted for all experiments in this book. A typical C program for microcontrollers consists of the following parts.

1. **Header Files** - This part of the program lists out all the files that are to be included on compilation of the program. For example, `#include inc/lm3s608.h`
2. **System Initialisation** - All the clock selections, phase locked loop (PLL) synthesis and frequency selection is done in this segment of your C code. It is the most vital part of your program without which the other segments will fail to function. For example, the function `SysCtlClockSet( )` (discussed in the next chapter) comes here.
3. **Peripheral Initialisation** - Suppose the experiment requires us to toggle a pin on a General-Purpose Input/Output(GPIO) port. Before toggling the pin, it is required to enable the GPIO peripheral and set its data direction, i.e., as an input or an output. The enabling of peripherals and its functions is done in this part of the C program. For example, `SYSCTL_RCGC2_R = SYSCTL_RCGC2_GPIOC` enables GPIO port C.
4. **Peripheral Configuration** - This part of the program is used to set individual configuration bits for a selected register or GPIO port. For example, `GPIO_PORTC_DIR_R | = 0x80`; configures the data direction of Bit 7 of port C as output and all the other bits of port C as inputs. Such a type of peripheral configuration is often referred to as bit-packed representation.[11]
5. **Application Loop** - Once the clock is enabled and peripherals are configured, it now time to jump to the main application of the code. This code is usually repeated in a loop over the entire execution cycle.

For convenience, the program flow diagram for the experiments included in this manual have been drawn in such a way as to segregate each of these subsections of the general C program. To further illustrate the functional parts of a C program, let us write some

applications concerned with GPIO ports using the the Register Access Model, where we access individual peripheral registers to toggle individual bits.

5.1 Example

The GPIO module on the LM3S608 is composed of five physical GPIO blocks, each corresponding to an individual GPIO port (Port A, Port B, Port C, Port D and Port E). The GPIO module supports 5-28 GPIO pins depending on the peripherals being used. Each GPIO pin has the following capabilities:

- Can be configured as an input or output port. On reset, they default to being an input, with the exception of the five JTAG pins.
- 5 V-tolerant in input configuration
- In input mode, can generate interrupts on low level, high level, rising edge, falling edge or both edges.
- Pins configured as inputs are Schmitt-triggered.
- Fast toggle capable of a change every two-clock cycle.
- Optional weak pull-up or pull-down resistors.
- Optional open-drain configuration. On reset, they default to standard push/pull configuration.
- Can be configured to be a GPIO or a peripheral pin. On reset, they default to being GPIOs.

The block diagram of the GPIO module on the LM3S608 is showm in Figure 5.1.

5.2 Experiment 1

5.2.1 Objective

To blink the LED, LD5, connected to PC7 on the Stellaris® Guru kit.

5.2.2 Program Flow

This program requires us to blink the LED connected to a pin on a GPIO port. The block diagram for the experiment is shown in Figure 5.2. The standard algorithm to be followed is:

1. Enable the system clock and the GPIO port.
2. Set the direction of the data register to *input* or *output*. In our case, it will be *output*.
3. Alternately set and reset the data bit driving the LED in order to generate a blinking effect.

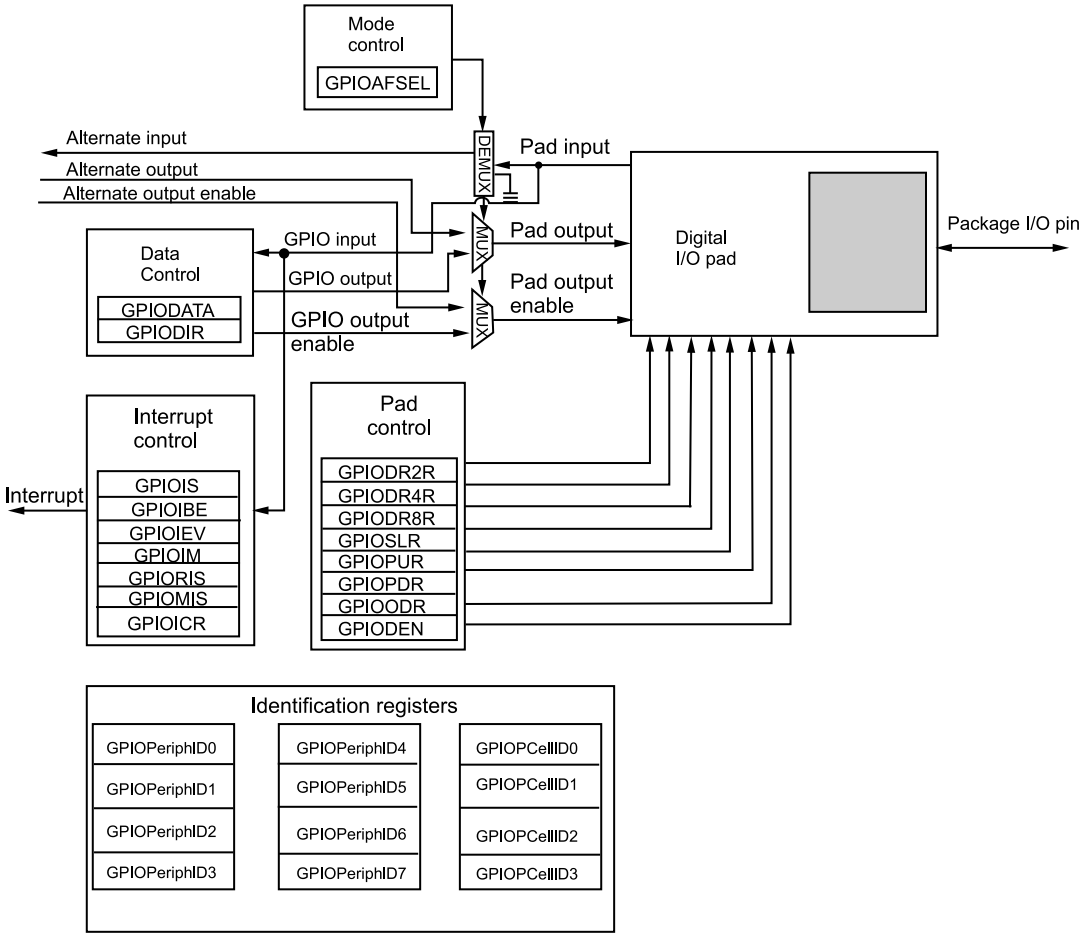

Figure 5.1 *GPIO module*

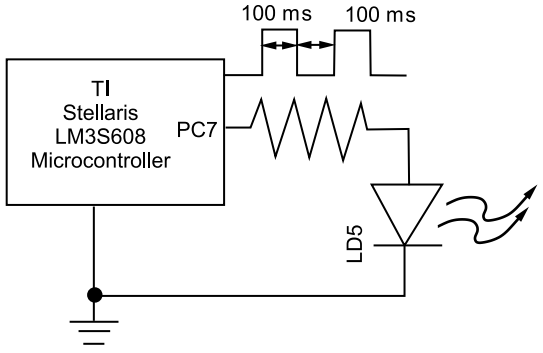

Figure 5.2 *Block Diagram for Experiment 1. The delay is generated using a* `for` *loop. PC7 is Pin 7 of port C.*

Figure 5.3 shows a suggested program flow for the experiment.

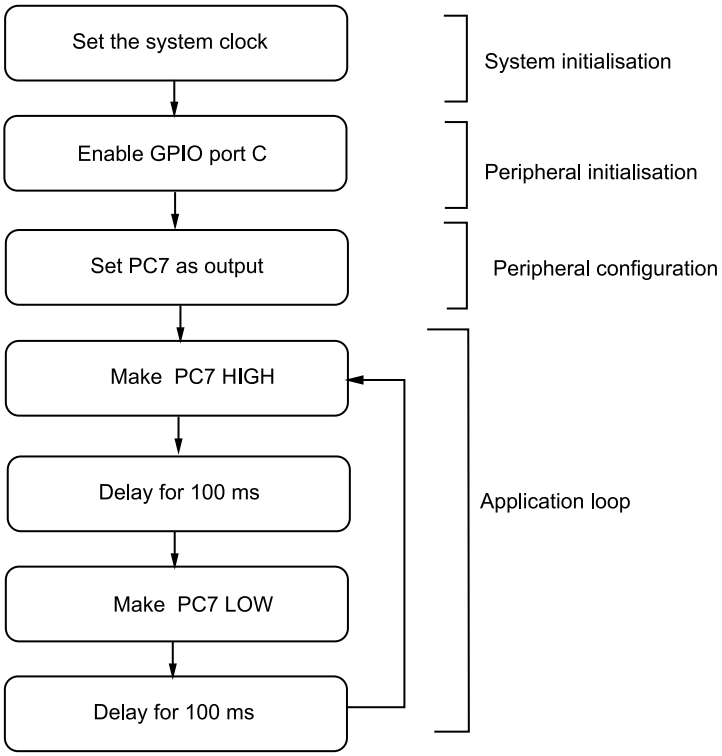

Figure 5.3 *Program flow for Experiment 1*

5.2.3 Register Accesses

To use the register access model, include the header file `inc/lm3s608.h`. This header file contains all the definitions and macros for using all the peripheral registers in LM3S608.

The registers that need to be accessed for system configuration and digital output are:

```
GPIO Data Register on Port C - GPIO_PORTC_DATA_R
GPIO Alternate Access Register on Port C - GPIO_PORTC_AFSEL_R
GPIO Direction Register on Port C - GPIO_PORTC_DIR_R
GPIO Digital Enable Register on Port C - GPIO_PORTC_DEN_R
```

5.2.4 Program Code

The complete C program code for the experiment is given below. The program has been broken up into segments with well-commented statements.

```
#include "inc/lm3s608.h"

int main(void)
{
//
// Delay Loop Variable
# define DELAYVALUE 200000
unsigned long ulLoop;

//
// Enable Clocking to the GPIO port that is used for the on-board
// LED.
//
SYSCTL_RCGC2_R = SYSCTL_RCGC2_GPIOC;

//
// Do a dummy read to insert a few cycles after enabling the
// peripheral.
//
ulLoop = SYSCTL_RCGC2_R;

//
// Enable the GPIO pin for the LED (PC7).  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_PORTC_DIR_R |= 0x80;
GPIO_PORTC_DEN_R |= 0x80;
GPIO_PORTC_AFSEL_R = 0x00;

//
// Application Loop
//
while(1)
{

    //
    // Turn off the LED.
    //
    GPIO_PORTC_DATA_R |= 0x80;
```

```
    //
    // Delay for 100 ms.
    //
    for(ulLoop = 0; ulLoop < DELAYVALUE; ulLoop++)
    {
    }

    //
    // Turn on the LED.
    //
    GPIO_PORTC_DATA_R &= ~(0x80);

    //
    // Delay for 100 ms.
    //
    for(ulLoop = 0; ulLoop < DELAYVALUE; ulLoop++)
    {
    }
       }
}
```

5.3 Experiment 2

5.3.1 Objective

To use switch S2 connected to PE0 as an input and turn the LD5 connected to PC7 on/off whenever the switch is asserted.

5.3.2 Program Flow

Lighting up an LED whenever a switch is asserted is the aim of this experiment. Whenever we press switch S2 on PE0, the LED connected to PC7 should light up, and it should turn off when switch is de-asserted. The block diagram for the experiment is shown in Figure 5.4. Building on the algorithm from the previous experiment, we can write the new algorithm for the experiment as:

1. Enable the system clock and the GPIO ports C and E.
2. Set the direction of the data register to *input* or *output*. In our case, port C will be set to *output* and port E will be set to *input*.
3. Set the individual bits which are to be on both the ports.
4. Wait for PE0 to be asserted.
5. When pin PE0 is asserted, turn on the LED LD5 connected to PC7. When PE0 is de-asserted, turn off LD5. The turning on/off can be controlled by writing suitable values to the data register of the GPIO port C.

The program flow for this experiment is shown in Figure 5.5.

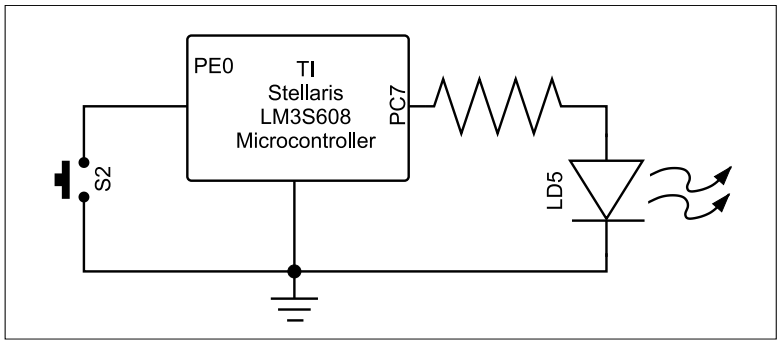

Figure 5.4 *Block diagram for Experiment 1. This method is also known as the polling method.*

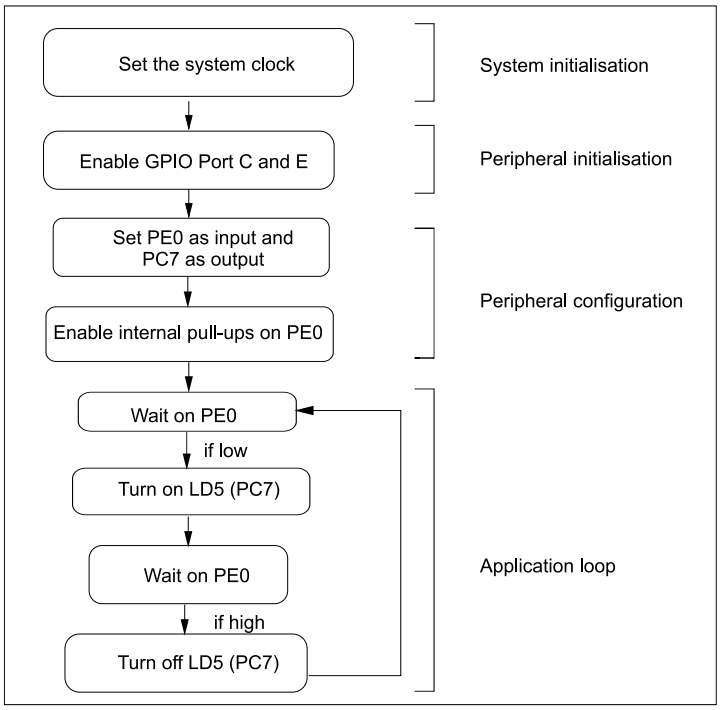

Figure 5.5 *Program flow for Experiment 2*

5.3.3 Register Accesses

To use the register access model, include the header file `inc/lm3s608.h`. This header file contains all the definitions and macros for using all the peripheral registers in LM3S608.

For the LED, the registers accessed are:

```
GPIO Data Register on Port C - GPIO_PORTC_DATA_R
GPIO Alternate Access Register on Port C - GPIO_PORTC_AFSEL_R
GPIO Direction Register on Port C - GPIO_PORTC_DIR_R
GPIO Digital Enable Register on Port C - GPIO_PORTC_DEN_R
```

For the switch, the following registers are accessed.

```
GPIO Data Register on Port E - GPIO_PORTE_DATA_R
GPIO Alternate Access Register on Port E - GPIO_PORTE_AFSEL_R
GPIO Direction Register on Port E - GPIO_PORTE_DIR_R
GPIO Digital Enable Register on Port E - GPIO_PORTE_DEN_R
GPIO Pull-up Enable Register on Port E - GPIO_PORTE_PUR_R
```

5.3.4 Program Code

```
#include "inc/lm3s608.h"

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

//
// Enable clocking to the GPIO port that is used for the on-board
// LED.
//
SYSCTL_RCGC2_R = SYSCTL_RCGC2_GPIOC | SYSCTL_RCGC2_GPIOE;

//
// Do a dummy read to insert a few cycles after enabling the
// peripheral.
//
ulLoop = SYSCTL_RCGC2_R;

//
// Enable the GPIO pin for the LED (PC7).  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_PORTC_DIR_R |= 0x80;
GPIO_PORTC_DEN_R |= 0x80;
GPIO_PORTC_AFSEL_R = 0x00;

//
// Set configuration for the GPIO E Peripheral.
//
GPIO_PORTE_DIR_R &= ~(0x01);
GPIO_PORTE_DEN_R |= 0x01;
GPIO_PORTE_AFSEL_R = 0x00;

//
// Pulling up PE0 Pin
//
GPIO_PORTE_PUR_R |= 0x01;

//
// Application
//
while(1)
{
    //
    // If the Button is Pressed
    //
    if(((GPIO_PORTE_DATA_R) & (0x01)) != (0x01))
    {

    //
    // LED is on
    //
    GPIO_PORTC_DATA_R &= ~(0x80);

    }

    else
    {

    //
    // LED is off
    //
    GPIO_PORTC_DATA_R |= 0x80;
    }
    }
}
```

6 StellarisWare API

This chapter introduces the reader to the StellarisWare application programming interface (API). Also discussed are two of the most common programming models used with this layer of abstraction.

6.1 StellarisWare Peripheral Driver Library: A Layer of Abstraction

The StellarisWare Peripheral driver library provided by Texas Instruments is a set of drivers for accessing the peripherals found on the Stellaris family of ARM® Cortex™-M based microcontrollers. The driver library, as shown in Figure 6.1, acts as a layer of abstraction for programmers and exposes only the functions of the library without going deeper into the lower-level working of the controller. This speeds up the development of end user applications.

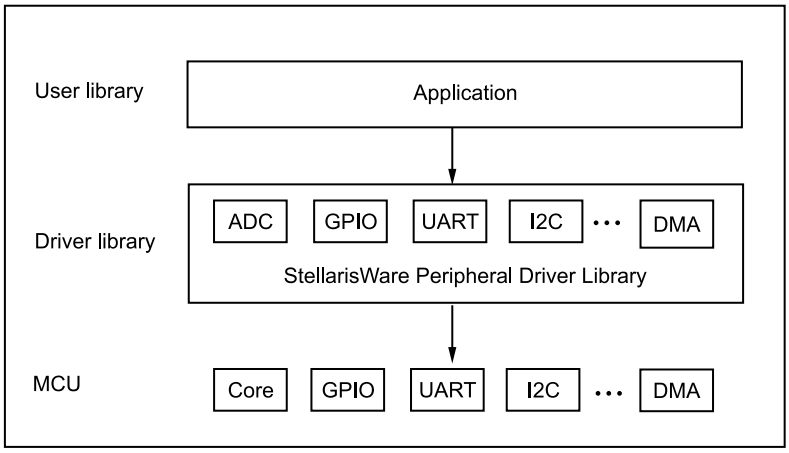

Figure 6.1 *Layers of abstraction*

6.2 Features of the StellarisWare Peripheral Driver Library

- Simplifies and speeds up the development of applications.
- Free license and royalty-free use.
- The high-level API interface includes all peripherals.
- Available as object library and as source code.
- Compiles across toolchains, ARM®/Keil[12], IAR[13], Code Red[14], CCS[15] and GNU tools.
- Peripheral driver library functions are pre-programmed in ROM for select Stellaris® MCU.
- Includes drivers for all classes of Stellaris® MCUs.

6.3 Programming Models

The peripheral driver library provides support for two programming models: the *direct register access model* and the *software driver model*. Each model can be used independently or can be combined, based on the need of the application or the programming environment. Each programming model has its advantages and disadvantages. The use of direct register access model generally results in smaller and more efficient code than with software driver model. However, the direct register access model requires a detailed knowledge of the operation of each register, each bit field and interactions, and any *sequencing* required for the proper operation of the peripheral. In the software driver model, the developer is insulated from these details, and it generally takes lesser time to develop applications using this model.

6.3.1 Direct Register Access Model

We have already used the direct register access model programming model in developing the programs of Chapter 5. Here the values are directly written into the peripheral registers. The `inc` directory includes part-specific header files for different Stellaris® MCUs containing macros mapping peripherals to their corresponding addresses. For example, the header file for the LM3S608 microcontroller is `inc/lm3s608.h`. The following naming conventions used in the model make it convenient for for use.

- Macro definitions ending in `_R` are used to access the value of the register. For example,

  `GPIO_PORTA_DATA_R` is used to access the data register for the GPIO port A.

- Macro definitions ending in `_M` are used for masking multiple-bit field values in a register.

- Macro definitions ending in `_S` represent the number of bits to shift in order to align it with a multi-bit field. These values match the macro with the same base name but ending with `_M`.
- Other macro definitions represent the value of the bit-field. All register bit fields start with the module name.
- Macro definitions mapping onto peripheral devices containing the module name and instance number. For example, use SSI0 for the first SSI (serial synchronous interface) module, followed by the name of registers as it appears in the data sheet.

For example,

1. The following statement will set the pin PD5 to HIGH.

```
GPIO_PORTD_DATA_R |= 0x20;
```

2. The following code will read and return the values from PD4-PD7.

```
return (GPIO_PORTD_DATA_R >> 4);
```

6.3.2 Software Driver Model

The software driver model uses the API provided by the peripheral driver library to control all the peripherals present in the Stellaris® MCUs. The API provides the capability to write complete applications using only pre-built functions, which helps in reducing the application development time. However, when compared to the direct register access model, the software driver model is not as efficient but enables rapid development of applications without requiring a knowledge of the low-level details of the operation of the peripherals. All example codes in this manual follow the software driver model. The functions of the API will be discussed as and when the need arises.

6.3.3 Using Both Models

Both the programming models mentioned above can be mixed in the same program, if necessary, to benefit from both of them. As mentioned above, the software driver model reduces the development time by hiding the low-level details of the peripherals, while the direct register access model will lead to programs that are faster and more space-efficient. A recommended practice is to use the direct register access model for the performance-critical portions of the code.

6.4 Useful StellarisWare API Function Calls

6.4.1 SysCtlClockSet

This sets the clocking of the device.

Prototype: `void SysCtlClockSet`(unsigned long `ulConfig`)

Parameters: `ulConfig` is the required configuration of the device clocking.

Description: This function configures the clocking of the device and all of the following can be specified using this function—the crystal frequency, the oscillator to be used, the usage of the on-chip PLL and the system clock divider. The `ulConfig` parameter is the logical OR of several different values, many of which are grouped into sets where only one can be specified. The system clock divider is specified with one of the following values:
`SYSCTL_SYSDIV_1, SYSCTL_SYSDIV_2,`
`SYSCTL_SYSDIV_3, SYSCTL_SYSDIV_64.`

Table 6.1 summarises the system divider and the clock settings.

Tabble 6.1 *System divider and clock settings*

SYSDIV	Divisor	Frequency	StellarisWare parameter
0x0	/1	Reserved	`SYSCTL_SYSDIV_1`
0x1	/2	Reserved	`SYSCTL_SYSDIV_2`
0x2	/3	Reserved	`SYSCTL_SYSDIV_3`
0x3	/4	50 MHz	`SYSCTL_SYSDIV_4`
0x4	/5	40 MHz	`SYSCTL_SYSDIV_5`
0x5	/6	33.33 MHz	`SYSCTL_SYSDIV_6`
0x6	/7	28.57 MHz	`SYSCTL_SYSDIV_7`
0x7	/8	25 MHz	`SYSCTL_SYSDIV_8`
0x8	/9	22.22 MHz	`SYSCTL_SYSDIV_9`
0x9	/10	20 MHz	`SYSCTL_SYSDIV_10`
0xA	/11	18.18 MHz	`SYSCTL_SYSDIV_11`
0xB	/12	16.67 MHz	`SYSCTL_SYSDIV_12`
0xC	/13	15.38 MHz	`SYSCTL_SYSDIV_13`
0xD	/14	14.29 MHz	`SYSCTL_SYSDIV_14`
0xE	/15	13.33 MHz	`SYSCTL_SYSDIV_15`
0xF	/16	12.5 MHz (default)	`SYSCTL_SYSDIV_16`

The use of the PLL is opted for with either `SYSCTL_USE_PLL` or `SYSCTL_USE_OSC`. The external crystal frequency is chosen with one of the following values:

```
SYSCTL_XTAL_1 MHZ,      SYSCTL_XTAL_1_84 MHZ,
SYSCTL_XTAL_2 MHZ,      SYSCTL_XTAL_2_45 MHZ,
SYSCTL_XTAL_3_57 MHZ,   SYSCTL_XTAL_4 MHZ,
SYSCTL_XTAL_4_09 MHZ    among others.
```

The oscillator 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

6.4.2 SysCtlClockGet

This gets the processor clock rate.

Prototype: unsigned long `SysCtlClockGet(void)`

Description: This function determines the clock rate of the processor, which is also the clock rate of the peripheral modules (with the exception of PWM, which has its own clock divider; some other peripherals too may have different clocking. See the device data sheet for details).

Note: This cannot return accurate results if `SysCtlClockSet()` has not been called to configure the clocking 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.

Returns: The processor clock rate.

6.4.3 SysCtlPeripheralEnable

Enables a peripheral.
Prototype: `void SysCtlPeripheralEnable`(unsigned long `ulPeripheral`)

Parameters: `ulPeripheral` is the peripheral to enable.

Description: This function enables peripherals. At power-up, all peripherals are disabled; they must be enabled in order to perform or respond to register reads/writes. The `ulPeripheral` parameter must be one of the following values only:

```
SYSCTL_PERIPH_ADC0,    SYSCTL_PERIPH_ADC1,
SYSCTL_PERIPH_CAN0,    SYSCTL_PERIPH_CAN1,
SYSCTL_PERIPH_CAN2,    SYSCTL_PERIPH_COMP0,
```

`SYSCTL_PERIPH_COMP1, SYSCTL_PERIPH_COMP2,`
`SYSCTL_PERIPH_EEPROM0,`
`SYSCTL_PERIPH_EPI0,`
`SYSCTL_PERIPH_ETH` etc.,

The parameter depends on the microcontroller in use, and the device datasheet should be consulted before programming.

Returns: None

6.4.4 SysCtlDelay

This provides a small delay.

Prototype: `void SysCtlDelay`(unsigned long `ulCount`)

Parameters: `ulCount` is the number of delay loop iterations to perform.

Description: This function provides a means of generating a desired delay of up to $2^{32} \times 3$ clock cycles, where 2^{32} is the size of unsigned long. The routine is written in assembly language to keep the delay consistent across tool chains, avoiding the need to tune the delay based on the toolchain. It takes 3 cycles/loop.

Returns: None.

6.4.5 SysCtlReset

Resets the device.

Prototype: `void SysCtlReset(void)`

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 previous value but has the *software reset bit* set as well).

Returns: None

6.4.6 IntMasterEnable

This enables the processor interrupt.

Prototype: tBoolean IntMasterEnable(void)

Description: This function allows the processor to respond to interrupts. It does not affect the set of interrupts enabled in the interrupt controller but just gates the single interrupt from the controller to the processor.

Returns: True if interrupt-enabled.

6.4.7 GPIOPinRead

This reads the values present of the specified pin(s).

Prototype: long `GPIOPinRead`(unsigned long `ulPort`, unsigned `char ucPins`)

Parameters: `ulPort` is the base address of the GPIO port. `ucPins` is the bit-packed representation of the pin(s).

Description: The values at the specified pin(s) are read, as specified by `ucPins`. Values are returned for both input and output pin(s), and the value for pin(s) that are not specified by `ucPins` are set to 0. The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns: A bit-packed byte providing the state of the specified pin, where bit 0 of the byte represents pin 0 of the GPIO port, bit 1 represents pin 1 of the GPIO port, and so on. Any bit that is not specified by `ucPins` is returned as a 0. Bits 31:8 should be ignored.

7 Digital Input/Output

This chapter revisits the two experiments performed in Chapter 5 and implements them using calls to the StellarisWare API rather than the register access model used earlier. It is left to the reader to use any programming paradigm in future experiments listed in this manual. There are other experiments provided in this chapter that utilise digital I/O concepts to control intensity of LEDs and using LED as a light sensor.

7.1 Experiment 3

7.1.1 Objective

Make the LED (LD5) connected to PC7 on the Stellaris® Guru development kit blink with a delay of 100 milliseconds.

7.1.2 Program Flow

This program requires that the LED connected to a pin on a GPIO port blink. The block diagram for the experiment is shown in Figure 7.1. The standard algorithm to be followed is:

1. Enable the system clock and the GPIO port.
2. Set the direction of the data register as output. In our case, it will be output.
3. Set and reset the data bit for the pin alternately with a delay to generate a blinking pattern.

Figure 7.2 shows a suggested program flow for the experiment.

.

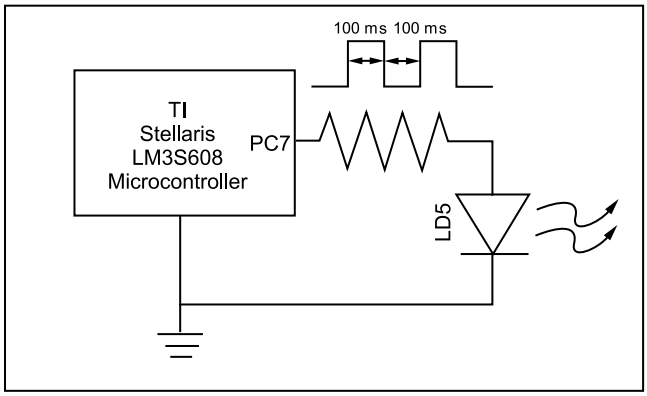

Figure 7.1 *Block diagram for Experiment 3. Delays are generated using* `SysCtlDelay`

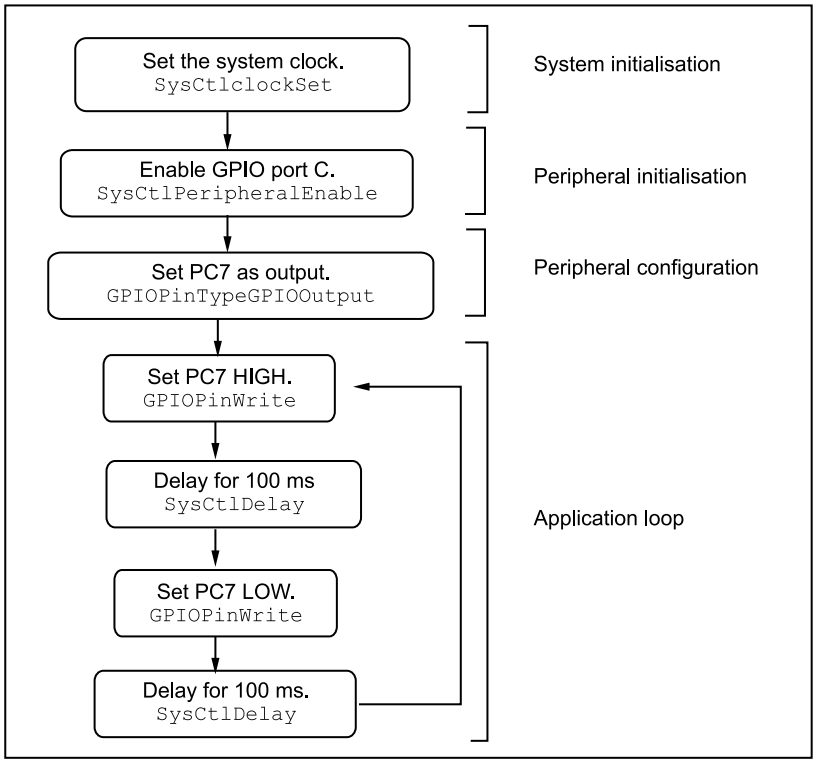

Figure 7.2 *Program flow for Experiment 3*

7.1.3 Suggested StellarisWare API Function Calls

The functions used to drive the GPIO port are contained in `driverlib/gpio.c`, with `driverlib/gpio.h` containing the API definitions for use by applications.

- **GPIOPinTypeGPIOOutput -** Configures the pin(s) for use as GPIO Outputs.

  Prototype: `void GPIOPinTypeGPIOOutput`(unsigned long `ulPort`, unsigned `char ucPins`)

  Parameters: `ulPort` is the base address of the GPIO port. `ucPins` is the bit-packed representation of the pin(s).

  Description: The GPIO pins must be properly configured in order to function correctly as GPIO outputs. The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

  Returns: None

- **GPIOPinWrite -** Writes a value to the specified pin(s)

  Prototype: `void GPIOPinWrite` (unsigned long `ulPort`, unsigned `char ucPins`, unsigned `char ucVal`)

  Parameters: `ulPort` is the base address of the GPIO port. `ucPins` is the bit-packed representation of the pin(s). `ucVal` is the value to write to the pin(s).

  Description: Writes the corresponding bit values to the output pin(s) specified by `ucPins`. Writing to a pin configured as an input pin has no effect.

  Returns: None

7.1.4 Program Code

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

```
//
// 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 clocking to directly run from the crystal at 8MHz */
//
SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN
| SYSCTL_XTAL_8MHZ);
//
// Peripheral Initialization Statement
// Set the clock for the GPIO Port C
//
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOC);

//
// Peripheral Configuration Statement
// Set the type of the GPIO Pins
//
GPIOPinTypeGPIOOutput(GPIO_PORTC_BASE, GPIO_PIN_7);

//
// GPIO Pin 7 on PORT C initialized to High, LD5 Off
//
GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_7, GPIO_PIN_7);

//
// Application Loop
//
while(1)
{

    //
    // Make Pin Low
```

```
    //
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_7, 0);

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

    //
    // Make Pin High
    //
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_7, GPIO_PIN_7);

    //
    // Delay for 100ms
    //
    SysCtlDelay(SysCtlClockGet()/100);
}
}
```

7.2 Experiment 4

7.2.1 Objective

Use switch S2 connected to PE0 as an input and make LD5 on PC7 blink whenever the switch is asserted.

7.2.2 Program Flow

Lighting up an LED whenever a switch is asserted is the aim of this experiment. Whenever we press switch S2 on PE0, the LED connected to PC7 should light up; it should turn off when the switch is de-asserted. The block diagram for the experiment is shown in Figure 7.3. Building on the algorithm from the previous experiment, we can write a new algorithm as:

1. Enable the system clock and the GPIO ports C and E.
2. Set the direction of the data register to input or output. In our case, it will be output for port C and input for E.
3. Set individual bits which are to be on both the ports.
4. Wait for PE0 to be asserted.
5. When asserted, turn on LED on PC7 and when de-asserted, turn off LED on PC7 by writing to the data register of GPIO port C.

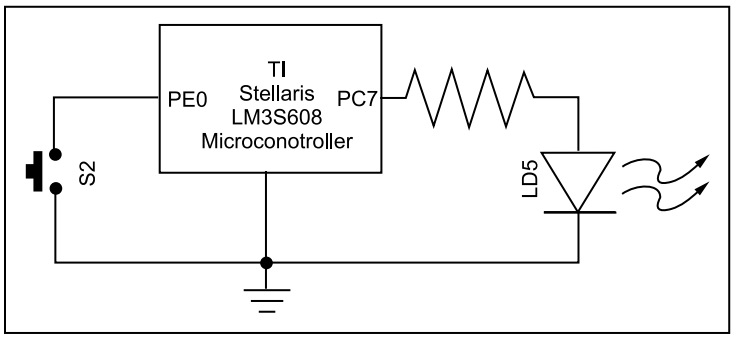

Figure 7.3 *Block diagram for Experiment 4. This method is also known as the Polling Method.*

The program flow for this experiment is shown in Figure 7.4.

7.2.3 Suggested StellarisWare API Function Calls

- **GPIOPinTypeGPIOInput** Configures the pin(s) as GPIO inputs.

  Prototype: `void GPIOPinTypeGPIOInput` (unsigned long `ulPort`, unsigned `char ucPins`);

  Parameters: `ulPort` is the base address of the GPIO port. `ucPins` is the bit-packed representation of the pin(s).

  Description: The GPIO pins must be properly configured in order to function correctly as GPIO inputs.

  Returns: None

- **GPIOPadConfigSet** Sets the pad configuration for the specified pin(s).

  Prototype:

  `void GPIOPadConfigSet` (unsigned long `ulPort`, unsigned `char ucPins`, unsigned long `ucStrength`, unsigned long `ulPinType`)

  Parameters: `ulPort` is the base address of the GPIO port. `ucPins` is the bit-packed representation of the pin(s). `ulStrength` specifies the output drive strength. `ulPinType` specifies the pin type.

  Description: This function sets the drive strength, type and pull-up/pull-down configuration for the specified pin(s) on the selected GPIO port. The parameter `ulStrength` can be one of the following values:

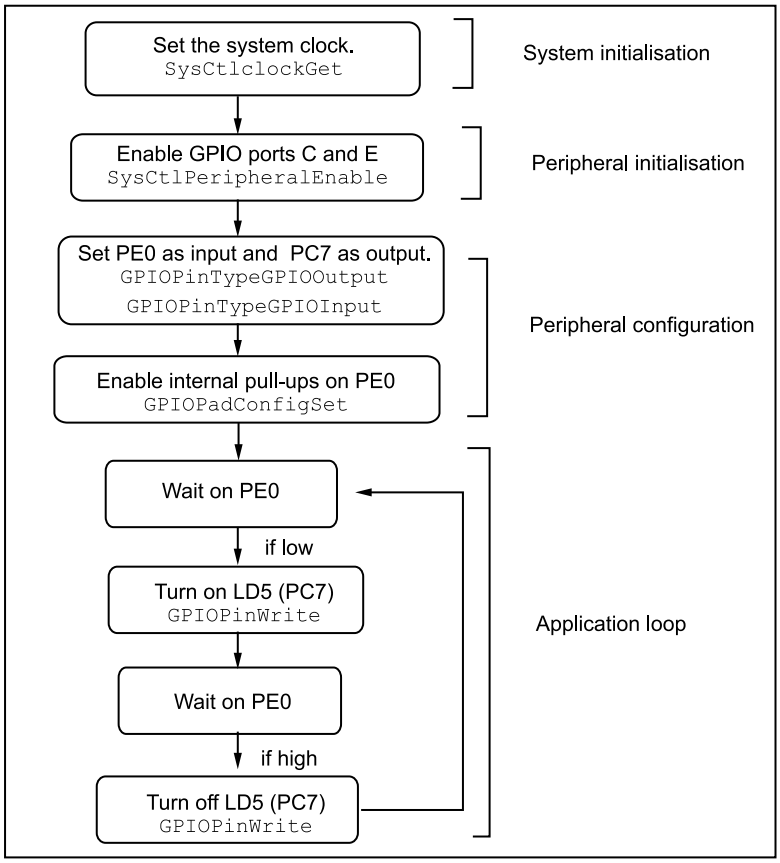

Figure 7.4 *Program flow for Experiment 4*

```
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.

The parameter ulPinType can take one of the following values:

```
GPIO_PIN_TYPE_STD_WPD,= GPIO_PIN_TYPE_OD
GPIO_PIN_TYPE_STD, GPIO_PIN_TYPE_STD_WPU
GPIO_PIN_TYPE_OD_WPU, GPIO_PIN_TYPE_OD_WPD
```

where,
`GPIO_PIN_TYPE_STD*` specifies a push-pull pin, `GPIO_PIN_TYPE_OD*` specifies an open-drain pin, `*_WPU` specifies a weak pull-up and `*_WPD` specifies a weak pull-down.

Returns: None

- **GPIOPinRead** Reads the value present of the specified pin(s).

  Prototype: `long GPIOPinRead`(unsigned long `ulPort`, unsigned char `ucPins`);

  Parameters: `ulPort` is the base address of the GPIO port. `ucPins` is the bit-packed representation of the pin(s).

  Description: The values at the specified pin(s) are read, as specified by `ucPins`.

  Returns: Returns a bit-packed byte providing the state of the specified pin, where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on. Any bit that is not specified by ucPins is returned as a 0. Bits 31:8 should be ignored.

7.2.4 Program Code

```
// 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
// Set the clocking to directly run from the crystal at 8MHz
SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN
                                      | SYSCTL_XTAL_8MHZ);
//
// Peripheral Initialization
// Set the clock for the GPIO Port C and Port E
//
```

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOC);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);

//
// Peripheral Configuration
// Set the type of the GPIO Pin
//
GPIOPinTypeGPIOOutput(GPIO_PORTC_BASE, GPIO_PIN_7);
GPIOPinTypeGPIOInput(GPIO_PORTE_BASE,GPIO_PIN_0);

//
// Configure GPIO pad with internal pull-up enabled for PE0
//
GPIOPadConfigSet(GPIO_PORTE_BASE, GPIO_PIN_0, GPIO_STRENGTH_2MA,
GPIO_PIN_TYPE_STD_WPU);

//
// GPIO Pin 7 on PORT C initialized to High, LD5 Off
//
GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_7, GPIO_PIN_7);

//
// Application Loop
//
while(1)
{
    //
    // Check if PE0 is high
    //
    if(GPIOPinRead(GPIO_PORTE_BASE, GPIO_PIN_0)==GPIO_PIN_0)
    {
    //
    // Make PC7 high, i.e. turn off LED
    //
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_7, GPIO_PIN_7);
    }

    else
    {
    //
    // Make PC7 Low, i.e., turn on LED
    //
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_7, 0);
    }
}
}
```

7.3 Experiment 5

7.3.1 Objective

To make LEDs LD5(PC7), LD6(PC6) and LD7(PC5) blink in a controlled pattern.

7.3.2 Program Flow

In this experiment, let us play with the LEDs. The experiment requires us to make the LEDs blink in a pattern. Unlike the previous experiments where we controlled only a single LED by making it blink continuously, this experiment will allow us to control three LEDs together. The choice of pattern is up to the reader. Figure 7.5 shows the block diagram of the setup required for this experiment. A suggested algorithm for the experiment is:

1. Enable the system clock and GPIO port C.
2. Set data direction as output for PC5, PC6 and PC7.
3. Set PC5 High for 1 second.
4. Set PC6 High and make PC5 Low for 1 second.
5. Set PC7 High and make PC6 Low for 1 second.
6. Repeat to generate a beautiful pattern.

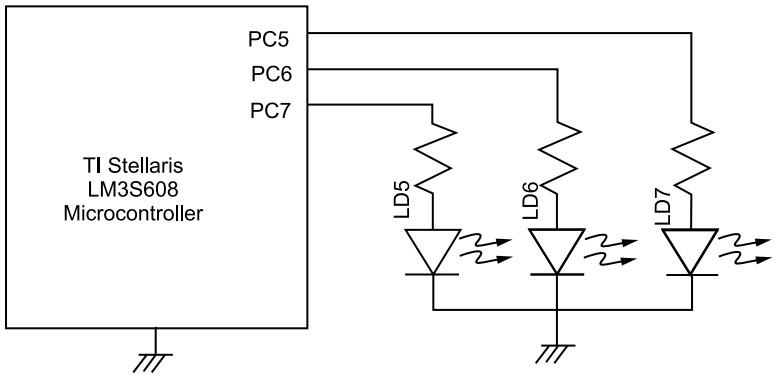

Figure 7.5 *Block diagram for Experiment 5*

A suggested program flow for the experiment is shown in Figure 7.6.

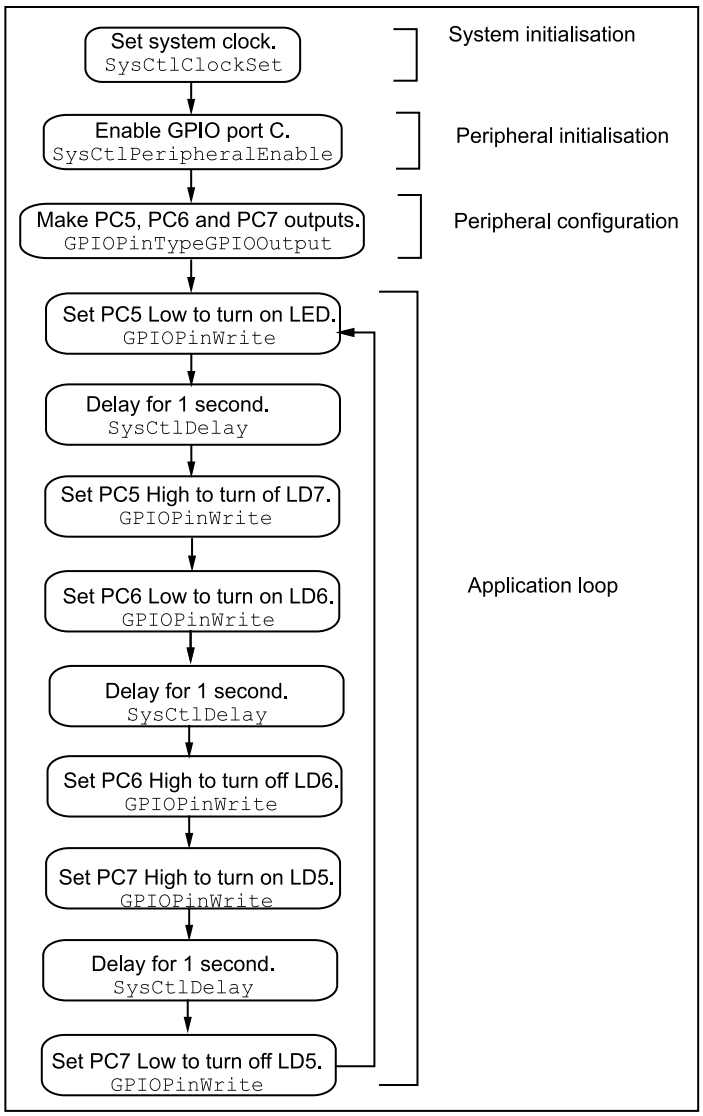

Figure 7.6 *Program flow for Experiment 5*

7.4 Experiment 6

7.4.1 Objective

To control the intensity of LED LD5 connected to PC7 using PWM implemented in software.

7.4.2 Program Flow

In all the experiments we have encountered so far, either the LED is ON or it is OFF. You may have seen an LED turn ON and OFF gradually over a prolonged period of time with its intensity varying from a minimum to a maximum (like Cylon's eyes in Battlestar Galactica for one!). Pulse width modulation allows us to control intensity by varying the on and off periods of a square waveform. For example, we achieve a 25% duty cycle, i.e., one-fourth the maximum intensity when the high period is one-third the off period or the high period is one-fourth the time period of the waveform. To achieve 50% duty cycle, an equal high and low periods or, a high period half the time period of the waveform is suitable. In this experiment, we control the intensity of LD5 using PWM techniques implemented in software by using variable delays to achieve different duty cycles. Hardware implementation of PWM is explained later in Chapter 9. We turn on the LED until it achieves its full intensity and then gradually dim it. Figure 7.7 shows the block diagram of the setup required for this experiment. The algorithm we follow for this experiment is:

1. Enable the system clock and GPIO port C.
2. Set data direction as output for PC7.
3. Create a variable whose value is incremented in every iteration of the loop, and the delay generated is proportional to the delay for the ON period.
4. Set PC7 High for a duration proportional to the value in the variable.

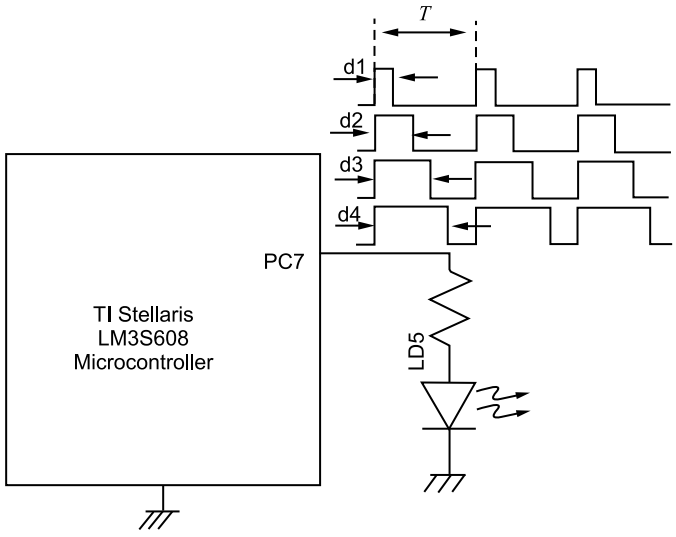

Figure 7.7 *Block diagram for Experiment 6*

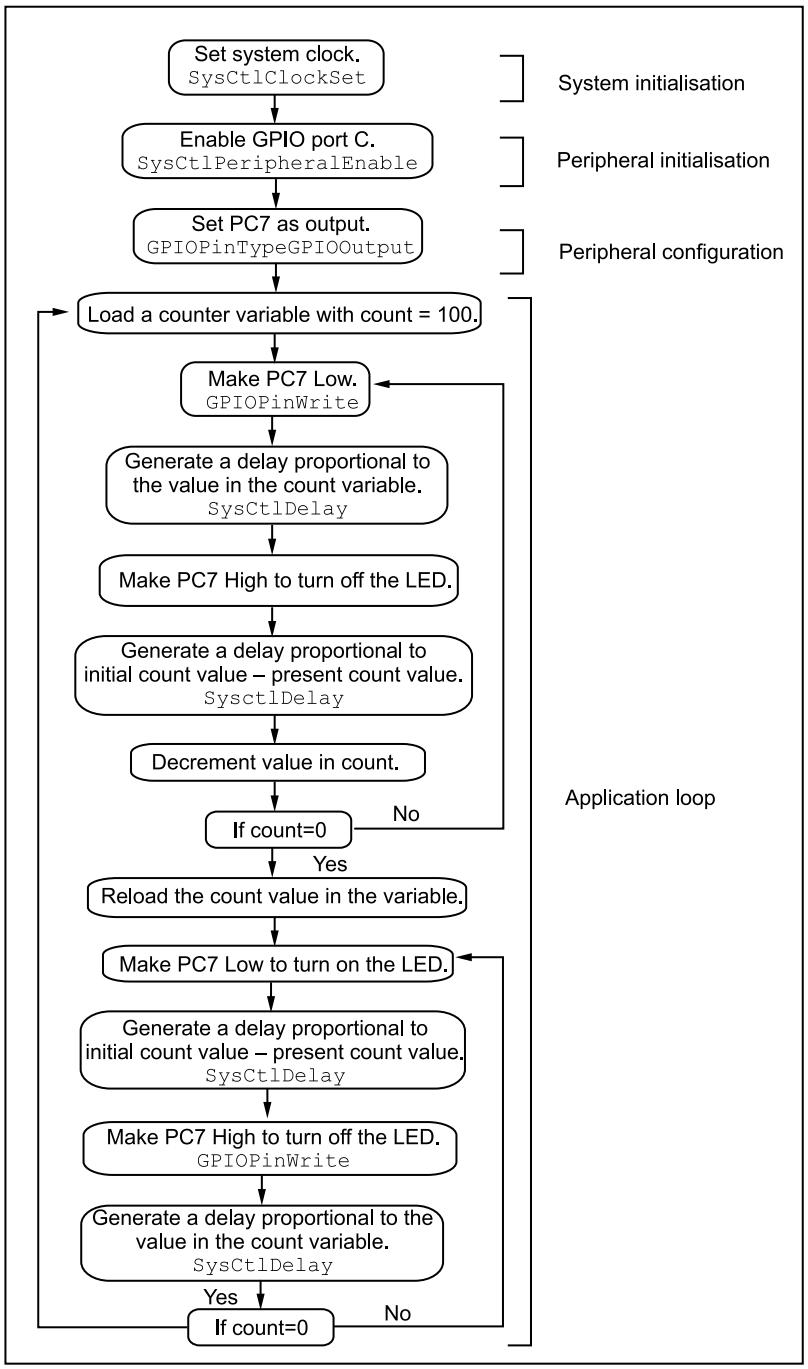

Figure 7.8 *Program flow for Experiment 6*

5. Set PC7 Low for a duration equal to OFF period (= Time period - ON period).
6. Increment the counter and repeat until OFF period is equal to zero.
7. Now decrement the counter and decrease the delay time.
8. Toggle the LEDs in the same manner until the ON period is equal to zero.

The program flow for the experiment is shown in Figure 7.8.

7.5 Experiment 7

7.5.1 Objective

To mimic light intensity sensed by the LED light sensor LD2(PD1) by varying the blinking rate of LED LD5(PC7).

7.5.2 Program Flow

We use an LED to measure ambient light intensity. Depending on the magnitude of the ambient light, we make LD5 blink at a rate proportional to it. For example, when the ambient light intensity is low, the LED blinks at a rate of 5 blinks/second, say, and when the intensity is higher, the LED blinks at 50 blinks/second. Figure 7.9 shows the block diagram of the setup required for this experiment. An algorithm which can be followed for the experiment is:

1. Enable the system clock and the GPIO ports C and D.
2. Set data direction of PC7 and PD1 as output.
3. Make PD1 High.

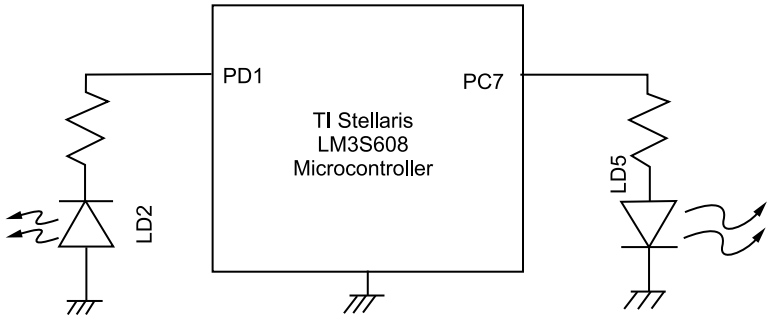

Figure 7.9 *Block diagram for Experiment 7*

4. After a short delay, change data direction of pin PD1 to input.
5. Start a counter and count until voltage on PD1 goes Low.
6. The value in the counter is proportional to the ambient light intensity.
7. Make LD5 blink at a rate proportional to the ambient light.

A suggested program flow for the experiment is shown in Figure 7.10.

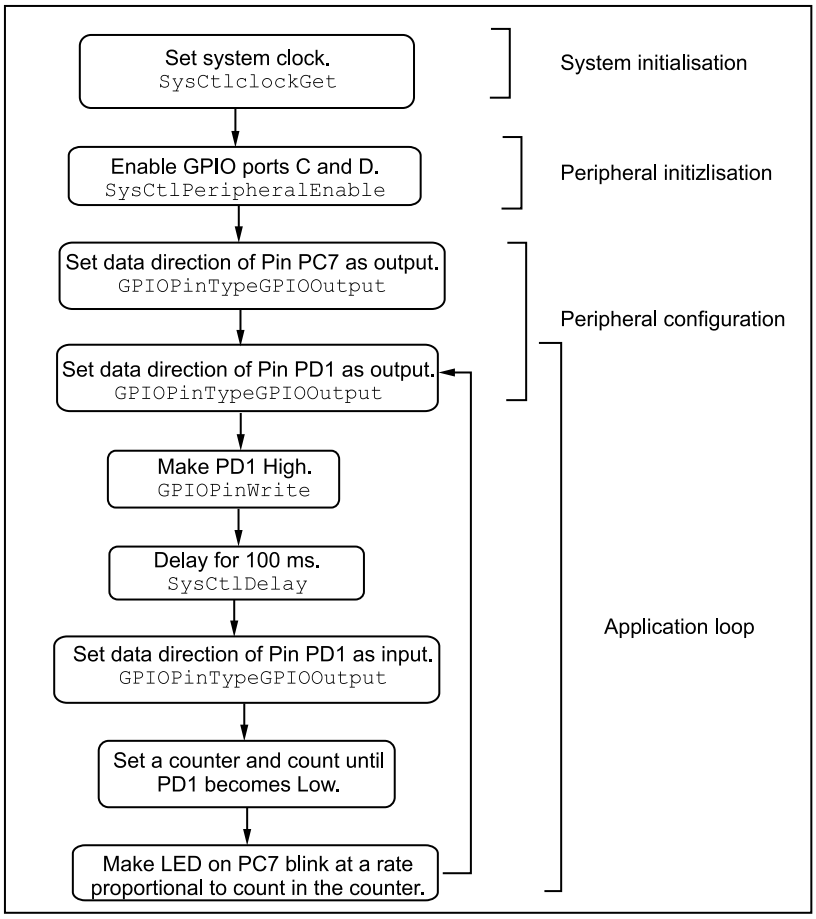

Figure 7.10 *Program flow for Experiment 7*

7.6 Generating Random Numbers

Many applications require random numbers. While it is almost impossible to generate a truly random number, one can generate a pseudo-random number using a variety of methods.

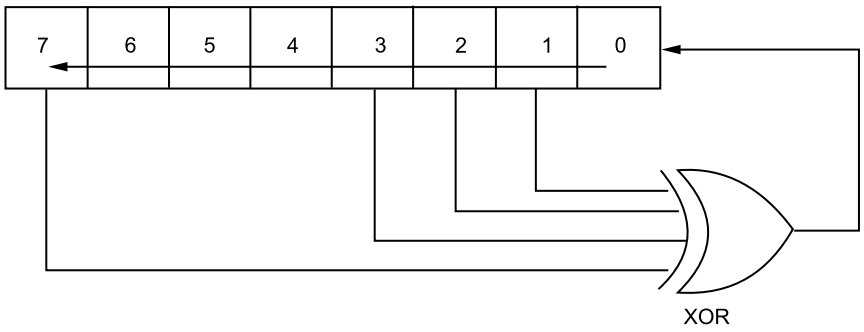

Figure 7.11 *An 8-bit linear feedback shift register with taps at bit positions 1, 2, 3 and 7*

One popular way to generate a pseudo-random number is to read the contents of a free-running counter. This is a simple scheme and can be easily implemented on the ARM® controller with no extra hardware. One of the available timers (or Timer1) can be clocked at a certain clock frequency derived from the system clock. Then, whenever a random number is required, the contents of the timer register can be read and the value that you get is a a pseudo-random number. This scheme is quite suitable in situations where the reading process is asynchronous to the activity on the microcontroller, such as a user pressing a switch to read a random number.

Another way to generate a pseudo-random number is to use the concept of Linear Feedback Shift Register (LFSR). LFSRs are ordinary shift registers with some outputs (called taps) feeding the input. LFSRs have an interesting property that if the feedback taps are chosen carefully, the outputs cycle through $2^n - 1$ sequences for an n-bit LFSR. The sequence then repeats after $2^n - 1$ instances. If the output sequences are observed, they appear to be random. An 8-bit LFSR is illustrated in Figure 7.11. An 8-bit LFSR will have a sequence length of 255. Similarly a 16-bit LFSR would have a length of 65535, and so on. (See Table 7.1 for a schema for pseudo-random number generation using LFSRs with bit values varying from 9 to 20.)

LFSRs can be be easily implemented on ARM® controllers. The LFSR must be initialised with a non-zero seed value. After the LFSR is initialised, it is clocked by shifting the values to the left and loading a new bit into the bit0 of the shift register. The new bit that is loaded into the bit0 of the shift register is calculated by XORing the bits at the selected taps of the LFSR.

This discussion on random number generation will be used in the exercises.

Table 7.1 *Scheme for pseudo random number generation using LFSR*

Bits	Sequence length	Taps
9	511	3,8
10	1023	2,9
11	2047	1,10
12	4095	0,3,5,11
13	8191	0,2,3,12
14	16,383	0,2,4,13
15	32,767	0,14
16	65535	1,2,4,15
17	131,071	2,16
18	262,143	6,17
19	524,287	0,1,4,18
20	1,048,575	2,19

7.7 Exercises

1. Toggle the state of LED LD5 by pressing and releasing switch S2.
2. Toggle the state of LED LD5 after Switch S2 is pressed and released twice.
3. Turn on LED LD5 for 5 seconds. During this 5 second window, count the number of times switch S2 is pressed and released. After the 5 second window is over, make LED LD6 blink the number of times switch S2 was pressed.
4. Use a random number generator based on a linear feedback shift register (LFSR). Output the value of the random number generator, one bit at a time on LED LD5, every 100 ms. Also try this experiment with 10 ms.
5. Make a coin-tossing machine using switch S2 and LEDs LD5 and LD6. If LD5 glows, it is 'heads' and if LD6 glows, it is 'tails'. Use an LFSR-based random number generator. When switch S2 is pressed the LFSR is clocked internally and when it is released, the output of the LFSR is used to determine if LD5 or LD6 should be turned on. If the output of the LFSR is '1', then LD5 is turned on and if it is '0', LD6 is turned on.

8 Interrupts

This chapter introduces the reader to the interrupt and exception handling features of the Stellaris® Guru. The Cortex™-M3 family of ARM® microcontrollers uses NVIC for managing and servicing interrupts requests.

8.1 Exception Handling

The LM3S608 microcontroller based on the Cortex™-M3 family uses a nested vectored interrupt controller (NVIC) for handling and prioritising exceptions and interrupts. All exceptions raised are handled when the CPU is in the handler mode. The processor state is saved onto the stack prior to this, and is returned when the Interrupt Service Routine (ISR) finishes. The vector table contains the reset value of the stack pointer and the start address, also called exception vectors, for all exception handlers. The vector address (of the exception raised) is fetched from the vector table in parallel to the state-saving process, which enables efficient interrupt entry. The processor also supports tail-chaining, which enables back-to-back interrupts to be performed without incurring the overhead of state-saving and restoration processes. Besides this, the NVIC also handles the priorities of exceptions. Software can set 8 priority levels on 7 exceptions (system handlers) and 23 interrupts (raised by peripherals). Priorities on the system handlers are set with the NVIC System Handler Priority n (SYSPRIn) registers. Interrupts are enabled through the NVIC Interrupt Set Enable n (ENn) register and prioritised with the NVIC Interrupt Priority n (PRIn) registers.

8.1.1 Exception Types

Exceptions can be classified into various different types as shown in Table 8.1.

- **Reset** - The reset is a special kind of an exception wherein the processor stops everything it is doing when reset is asserted. When de-asserted, the processor starts execution from the address given in the vector table.

Table 8.1 *Exception types*

Exception type	Vector number	Priority	Vector address or offset
–	0	–	0x0000.0000
Reset	1	−3 (highest)	0x0000.0004
Non-maskable interrupt (NMI)	2	−2	$0x$ 0000.0008
Hard fault	3	−1	0x0000.000C
Memory management	4	programmable	0x0000.0010
Bus fault	5	programmable	0x0000.0014
Usage fault	6	programmable	0x0000.0018
–	7 – 10 (Reserved)	–	–
SVCall	11	programmable	
Debug monitor	12	programmable	0x0000.0030
–	13	–	–
PendSV	14	programmable	0x0000.0038
SysTick	15	programmable	0x0000.003C
Interrupts	16 and above	programmable	0x0000.0040 and above

- **NMI** - The non-maskable interrupt (NMI) has the second highest priority after reset. It can be signalled using either the NMI signal or triggered by software using the Interrupt Control and State (INTCTRL) register.
- **Hard Fault** - Hard fault exceptions arise because of an error in exception handling or because the exception could not be handled by any of the handlers. They have a priority above all exceptions with configurable priorities.
- **Memory Management Fault** - Memory management fault exceptions are raised when the processor tries to violate memory access rules.
- **Bus Fault** - A bus fault is an exception that occurs because of a memory related fault for an instruction or data memory transaction such as prefetch fault or memory access fault. Bus fault exception is user-controlled and can be enabled or disabled.
- **Usage Fault** - A usage fault is an exception that occurs because of a fault related to instruction execution, such as
  - an undefined instruction
  - an illegal unaligned access
  - invalid state of instruction execution
  - an error on execution return.
- **SVCall** - A Supervisor Call (SVC) is made in an operating system framework. Applications use the SVC instructions to access the OS kernel related functions.

- **Debug Monitor** - This exception is caused by the debug monitor (when not halting). This exception is active only when enabled.
- **PendSV** - It is a pendable, interrupt-driven request for system-level service. It is used in an OS environment for context-switching. It is triggered using the Interrupt Control and State (INTCTRL) register.
- **SysTick** - The system timer, Systick, is capable of generating exceptions whenever it counts down to zero. It helps in generating regular ticks and can be used in an OS environment as a system tick.
- **Interrupt (IRQ)**- An interrupt is an exception signalled by a peripheral or generated by a software request. It can be prioritised and is asynchronous to instruction execution. Peripherals capable of generating interrupts include GPIO, UART, SSI, I2C, Timers and ADC. A complete detailed list of interrupts, including vector address offsets, can be found in the datasheet of LM3S608.

8.1.2 Exception States

Any exception is always in one of the following states:

- **Inactive** - The exception is neither active nor pending.
- **Pending** - The exception has occurred and is waiting to be serviced by the microcontroller. Another interrupt request can change the state of the corresponding exception to pending.
- **Active** - An exception that is being serviced by the processor but has not been completed.
- **Active and Pending** - The exception is being serviced, and in the meanwhile another exception is raised from the same source.

8.1.3 Exception Handler

The processor handles the various exceptions using:

- **Interrupt Service Routines (ISRs)** - Interrupts are handled by ISRs.
- **Fault Handlers** - Hard fault, memory management fault, usage fault, and bus fault are the fault exceptions handled by fault handlers.
- **System Handlers** - NMI, PendSV, SVCall, SysTick, and all the fault exceptions are system exceptions that are handled by system handlers.

8.1.4 Exception Priorities

All exceptions have a priority associated with them, with a lower priority number indicating a higher priority and a configurable priority for all exceptions except Reset, Hard Fault and NMI. By default, configurable priorities have a priority of 0. Reset, Hard Fault and NMI have negative priorities and are not configurable. If multiple pending exceptions have the same priority, the pending exception with the lowest exception number (refer to Table 8.1) takes precedence. When the processor is executing an exception handler, the exception handler is pre-empted if a higher priority exception occurs. The NVIC also supports priority grouping that divides each interrupt priority register into two fields:

- An upper field that defines the group priority
- A lower field that defines the subpriority within the group

Only the group priority defines the pre-emption of interrupt exceptions. If multiple pending interrupts have the same group priority, the subpriority field determines the order in which they would be processed.

8.2 Experiment 8

8.2.1 Objective

Control an LED using a switch by interrupt method and flash the LED once in every five switch-presses.

8.2.2 Program Flow

In this experiment, it is required to get the LED blink at every five button presses. When the button is pressed, it interrupts the processor and increments a variable counter. When counter becomes equal to five, the LED connected to the GPIO port blinks. The block diagram for the experiment is shown in Figure 8.1. The algorithm to be followed is:

1. Enable the system clock and the GPIO ports C and E.
2. Set direction of GPIO pins and enable internal pull-ups for the push button.
3. Configure the interrupt sequence and the type of triggering.
4. When the button is pressed and the ISR called, increment a counter.
5. If counter is equal to five, blink the LED and reset the counter.
6. Increment counter in subsequent button asserts and repeat.

A suggested program flow for the experiment is shown in Figure 8.2.

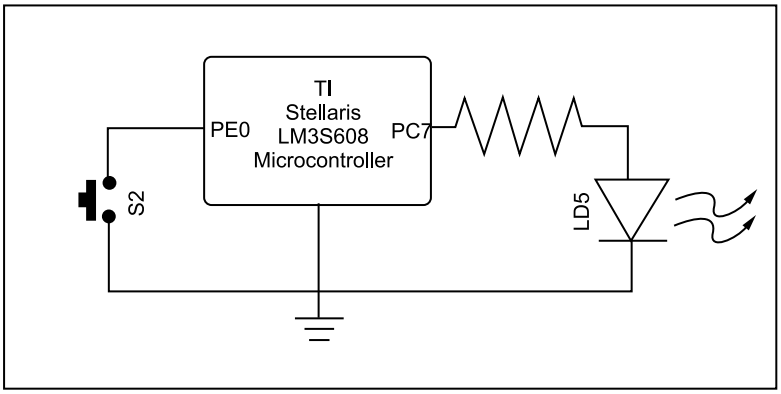

Figure 8.1 *Block diagram for Experiment 8*

8.2.3 Suggested StellarisWare API Function Calls

- **GPIOPortIntRegister** an interrupt handler for a GPIO port.

  Prototype: `void GPIOPortIntRegister`(unsigned long `ulPort, void (*pfnIntHandler)(void))`

  Parameters: `ulPort` 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 at the selected GPIO port. This function also enables the corresponding GPIO interrupt in the interrupt controller; individual pin interrupts and interrupt sources must be enabled with `GPIOPinIntEnable()`.

  Returns: None.

- **GPIOPinIntClear** Clears the interrupt for the specified pin(s).

  Prototype: `void GPIOPinIntClear`(unsigned long `ulPort`, unsigned `char ucPins`)

  Parameters: `ulPort` is the base address of the GPIO port. `ucPins` is the bit-packed representation of the pin(s).

  Description: Clears the interrupt for the specified pin(s).The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

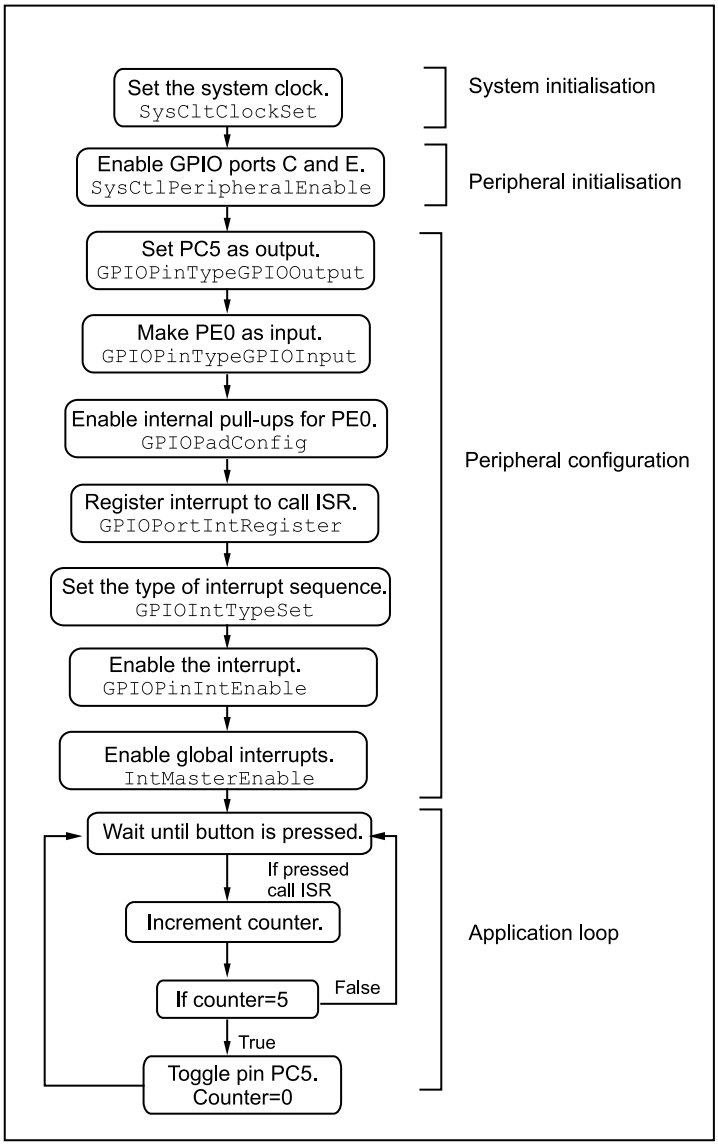

Figure 8.2 *Program flow for Experiment 8*

Returns: None.

- **GPIOIntTypeSet** Sets the interrupt type for the specified pin(s).

  Prototype: `void GPIOIntTypeSet`(unsigned long `ulPort`, unsigned `char ucPins`, unsigned long `ulIntType`)

  Parameters: `ulPort` is the base address of the GPIO port. `ucPins` is the bit-packed

representation of the pin(s). `ulIntType` 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 `ulIntType` is an enumerated data type that can have 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 also 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, so consult the data sheet to ensure that the device and the GPIO port support discrete interrupts. The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns: None.

- **GPIOPinIntEnable** Enables interrupts for the specified pin(s).

  Prototype: `void GPIOPinIntEnable`(unsigned long `ulPort`, unsigned `char ucPins`)

  Parameters: `ulPort` is the base address of the GPIO port. `ucPins` is the bit-packed representation of the pin(s).

  Description: Unmasks the interrupt for the specified pin(s). The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

  Returns: None.

- **IntEnable** Enables an interrupt.

  Prototype: `void IntEnable`(unsigned long `ulInterrupt`)

  Parameters: `ulInterrupt` specifies the interrupt to be enabled.

  Description: The specified interrupt is enabled in the interrupt controller. Other enables for the interrupt (such as at the peripheral level) are unaffected by this function.

  Returns: None.

- **IntMasterEnable** Enables the processor interrupt.

  Prototype:

  tBoolean IntMasterEnable(void)

  Description: This function allows the processor to respond to 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 interrupt is enabled.

8.3 Exercises

1. Turn on LED LD5 for 5 seconds. During this 5-seconds window, count the number of times switch S2 is pressed and released using the interrupt method. After the 5-seconds window is over, make LED LD6 blink the number of times S2 was pressed.
2. Play a game between two players. Player 1 has to press switch S2 and player 2 has to press S3. The two switches must be read using interrupt method. Turn on LED LD5 for 5 seconds. During this window count the number of times switches S2 and S3 are pressed. At the end of the 5-seconds interval, display the winner on LD6 if Player 1 was able to press his switch more number of times than Player 2, or LD7 if Player 2 wins by pressing his switch more number of times than Player 1.
3. Play fastest-finger-first game between two players using switches S2 and S3 which must be read using the interrupt method. Indicate the start of the game by turning on LED LD5 after a random period of time. Thereafter if S2 is pressed first, then turn on LD6, else if S3 is pressed first, then turn on LD7. In the event of a tie, turn both the LEDs, LD6 and LD7, on.
4. Create electronic dice using LEDs LD5, LD6 and LD7. These LEDs are used to represent numbers between 1 and 6 in binary form. Create a software loop to count between 1 to 6 over and over again. The user has to press switch S2 in interrupt method. When the switch is pressed, the current count in the loop is displayed on the LEDs.

9 Timer and Counter

This chapter deals with the timing and counting features of the LM3S608 on the Stellaris® Guru evaluation kit. The basic concepts of 16, 32 and 24 bit timers and the watchdog timer are explained in this chapter with the help of several experiments.

9.1 Introduction

The LM3S608 contains several timers, viz. general purpose timers, and two system control blocks, viz., SysTick and the watchdog timer. We cover each timer and system block in detail starting with the general-purpose timers.

9.1.1 General-Purpose Timers

Programmable timers can be used to count or time external events that drive the timer input pins. The Stellaris® General-Purpose Timer Module (GPTM) consists of three GPTM blocks: Timer0, Timer1 and Timer2. Each timer block consists of two 16-bit timers that can be operated as two individual 16-bit timers, or as one 32-bit timer. In addition, the GPTM can be used to trigger the ADC, generate PWM for motion control, and as a 32-bit timer for real time clock operation. Figure 9.1 shows the block diagram of the general-purpose timer module. We summarise some of the features of the GPTM below.

- Three general-purpose timer modules each of which provides two 16-bit timers/counters. Each can be configured to operate independently in the following ways.
  - As a single 32-bit timer
  - As one 32-bit real time clock
  - For pulse width modulation
  - To trigger analog-to-digital conversions

- 32-Bit and 16-bit timer modes
  - Programmable one-shot mode
  - Programmable Periodic timer
  - ADC event trigger
- 16-bit PWM mode with software programmable output inversion of the PWM signal.

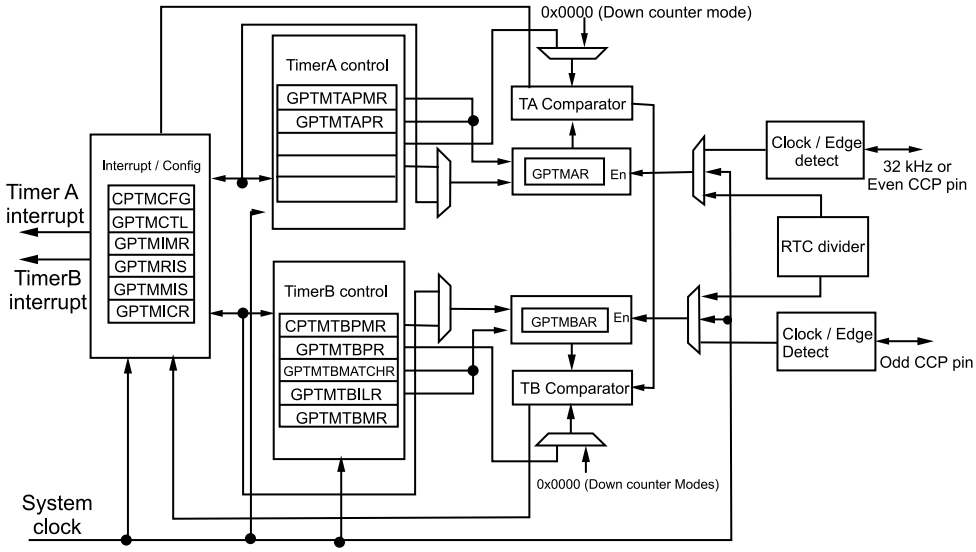

Figure 9.1 *General-purpose timer block diagram (Stellaris® LM3S608 microcontroller datasheet)*

9.1.2 SysTick Timer

The Cortex™-M3 includes an in-built system timer called SysTick which provides a 24-bit, clear-on-write, decrementing and wrap-on-zero counter with a flexible control mechanism. The counter can be used in the following ways.

- As an RTOS tick timer that fires at a programmable rate and invokes a SysTick routine
- As a high-speed alarm timer using the system clock
- As a simple counter used for measuring time to completion and run time.

When enabled, the timer counts down on each clock from the reload value to zero, reloads or wraps on the next clock edge, and then decrements on subsequent clocks. The SysTick timer runs on the system clock, and so if this clock signal is stopped in low power mode, the SysTick counter stops. Moreover, when the system is halted for debugging, the timer does not decrement.

9.1.3 Watchdog Timer

A watchdog timer can generate a non-maskable interrupt or an interrupt with highest priority when a time-out value is reached. The interrupt can either reset the system or call any other interrupt service routine. The watchdog timer is used to regain control of the system in case of failure due to a software error or due to the failure of an external device responding in an unexpected way.

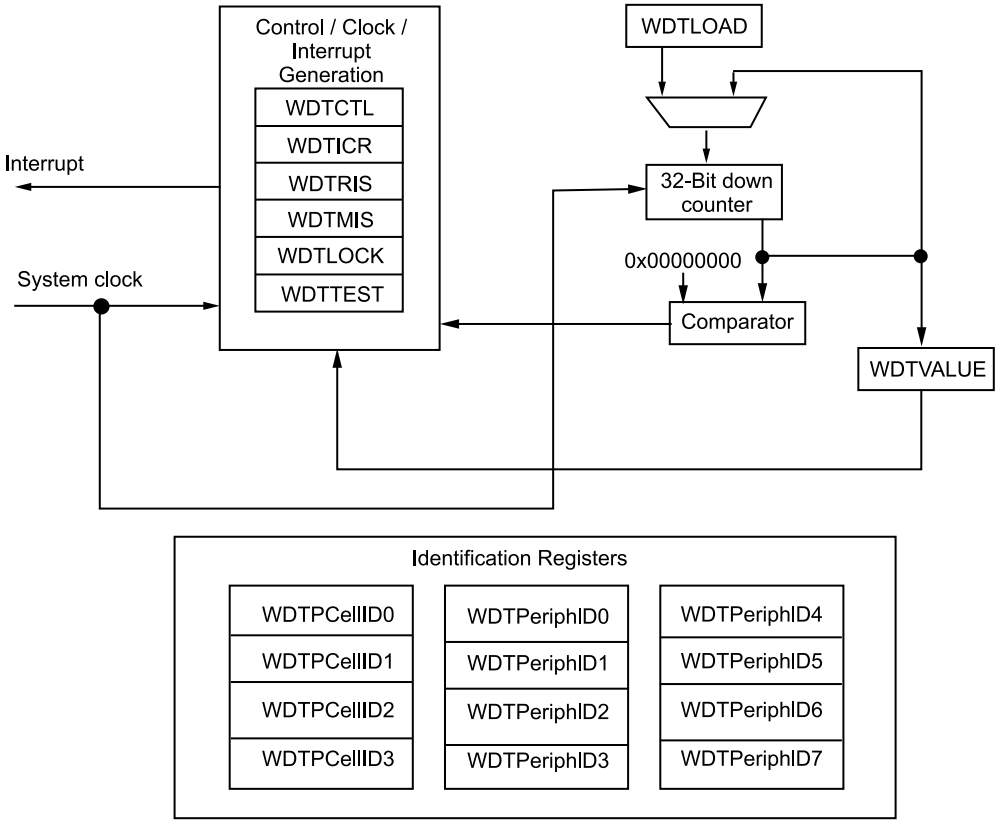

Figure 9.2 *Watchdog timer block diagram (Stellaris® LM3S608 microcontroller datasheet)*

The Stellaris® watchdog timer module has the following features.

- 32-bit down counter with a programmable load register
- Separate watchdog clock with an enable.
- Programmable interrupt generation with interrupt masking
- Reset generation logic with an enable/disable

The Watchdog timer can be configured to generate an interrupt on its first time-out and

a reset on its second time-out. The block diagram for the Watchdog module is shown in Figure 9.2.

9.2 Functional Description

9.2.1 General-Purpose Timer Module

The timer module provides two half-width timers/counters that can be configured to operate individually as timers or event counters, or can be combined to operate as one full-width timer or real-time clock (RTC). The two half-width timers are referred to as TimerA and TimerB and can be used together as a 32-bit full width timer, also referred to as TimerA. When configured as half-width or full-width, the timers can be used either in one-shot or continuous mode. In one-shot mode, when the timer reaches zero it will cease to count. In continuous mode, when the timer reaches zero it will reload the count and continue on the subsequent clock edges. It can also be used for event capture or as a pulse width modulation (PWM) generator in the half-width configuration. Control is also provided over interrupt sources and events. Interrupts can be generated to indicate that an event has been captured, or that a certain number of events have been captured. Interrupts can also be generated when the timer has counted down to zero, or when the RTC matches a certain value.

9.2.2 SysTick Timer

SysTick timer is a countdown timer and can be used to provide an interrupt whenever it wraps on to zero. The timer consists of three registers,

- **SysTick Control and Status (STCTRL):** A control and status register to configure its clock, enable the counter, enable the SysTick interrupt and determine the counter status.
- **SysTick Reload Value (STRELOAD):** The reload value for the counter, used to provide the counter's wrap value.
- **SysTick Current Value (STCURRENT):** The current value of the decrementing counter.

The SysTick interrupt handler does not need to clear the SysTick interrupt source. This will be done automatically by the NVIC when the SysTick interrupt handler is called.

9.2.3 Watchdog Timer

The Watchdog timer module generates the first time-out signal when the 32-bit counter reaches the zero state after being enabled; enabling the counter also enables the watchdog timer interrupt. After the first 32-bit time-out, the counter is reloaded again and resumes counting down from the value. If the timer counts down to its zero state again and the

reset signal has been enabled, the Watchdog timer asserts its reset signal to the system. The Watchdog module interrupt and reset generation can be enabled or disabled as required. When the interrupt is re-enabled, the 32-bit counter is preloaded with the load register value and not its last state. Some of the important registers for controlling the Watchdog timer are:

- **Watchdog Load (WDTLOAD):** When this register is written, the value is immediately loaded and the counter restarts counting down from the new value.
- **Watchdog Value (WDTVALUE):** This register contains the current count value of the timer.
- **Watchdog Control (WDTCTL):** This register is the watchdog control register. The watchdog timer can be configured to generate a reset signal (on second time-out) or an interrupt on time-out.

9.3 Experiment 9

9.3.1 Objective

To make an LED connected to PC5 blink using delays generated with the SysTick Timer.

9.3.2 Program Flow

The aim of this experiment is similar to the experiment encountered in the chapter on Digital I/O. It requires us to make an LED blink with a delay. However, the only difference between the two experiments is in the way the delay is generated. The block diagram for the experiment is shown in Figure 9.3. In this experiment, we generate delays using the SysTick timer. We load a count in the timer and check it periodically until it reaches zero. A suitable algorithm for this program is:

1. Enable the system clock, GPIO port C and the SysTick timer.
2. Configure the GPIO port and load an initial count into the SysTick timer.
3. Begin counting and check when the counter reaches zero.
4. When it reaches zero, toggle the LED connected to PC5.
5. The count gets reloaded into the SysTick timer, which starts counting again.
6. Toggle the LED again when counter touches zero.

A suggested program flow for the experiment is shown in Figure 9.4.

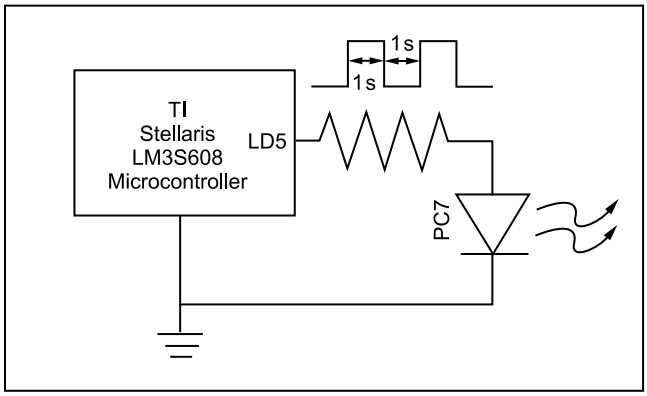

Figure 9.3 *Block diagram for Experiment 9*

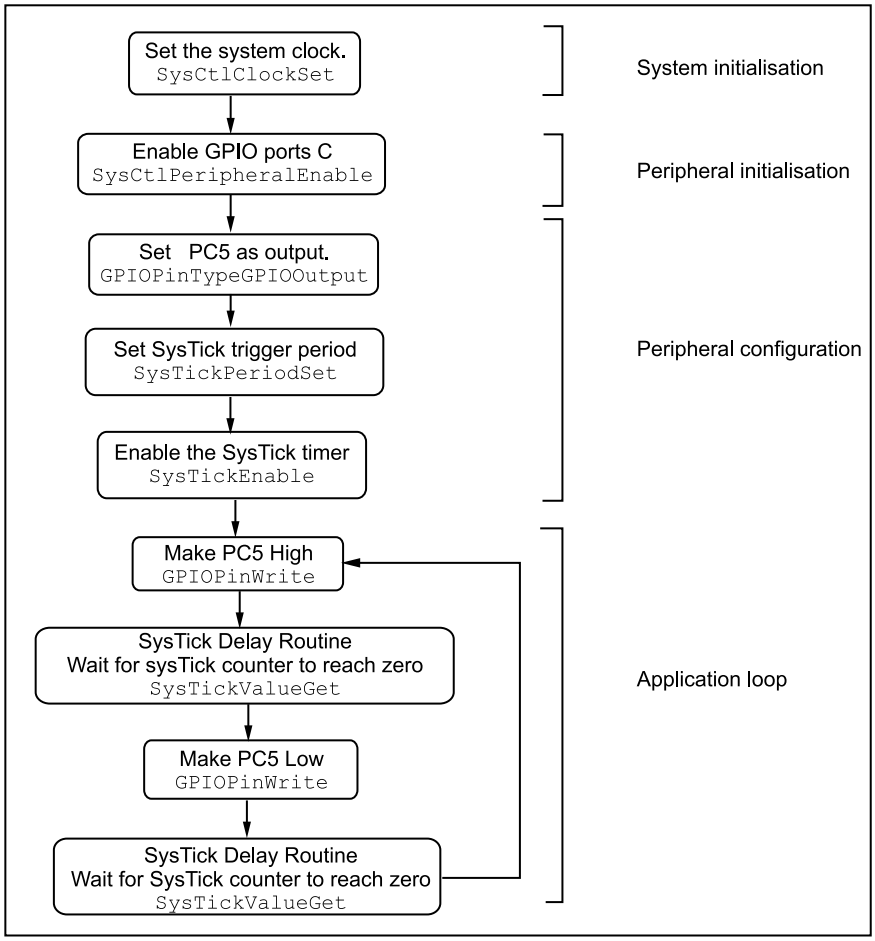

Figure 9.4 *Program flow for Experiment 9*

9.3.3 Suggested StellarisWare API Function Calls

The functions required to drive the SysTick timer module are listed in `driverlib/systick.c` with `driverlib/systick.c` containing the API definitions that are useful in applications.

- **SysTickPeriodSet** Sets the period of the SysTick counter.

  Prototype: `void SysTickPeriodSet`(unsigned long `ulPeriod`)

  Parameters: `ulPeriod` is the number of clock ticks in each period of the SysTick counter; must be between 1 and 2^{24} $(16,777,216)$ inclusive.

  Description: This function sets the rate at which the SysTick counter wraps; this is equal to the number of processor clocks between interrupts.

  Returns: None

- **SysTickEnable** Enables the SysTick counter.

  Prototype: `void SysTickEnable`(void)

  Parameters: None

  Description: This function will start the SysTick counter. If an interrupt handler has been registered, it is called when the SysTick counter rolls over.

  Returns: None

- **SysTickValueGet** Gets the current value of the SysTick counter.

  Prototype: unsigned long `SysTickValueGet`(void)

  Parameters: None

  Description: This function returns the current value of the SysTick counter, which is a value between Period -1 and Period-0, inclusive.

  Returns: Returns the current value of the SysTick timer.

9.4 Experiment 10

9.4.1 Objective

To control the intensity of an LED connected to PC5 using PWM implemented in hardware using Timer 0 in the GPTM module. Dim and brighten the LED in steps alternatively.

9.4.2 Program Flow

The block diagram for the experiment is shown in Figure 9.5. A suggested program flow is shown in Figure 9.6.

9.4.3 Suggested StellarisWare API Function Calls

The functions required to drive the timers in the GPTM are contained in `driverlib/timer.c` and defined for use in applications in `driverlib/timer.h`.

- **TimerConfigure** Configures the timer(s).

  Prototype: `void TimerConfigure`(unsigned long `ulBase`, unsigned long `ulConfig`)

  Parameters: `ulBase` is the base address of the timer module. `ulConfig` is the configuration of the timer.

  Description: This function configures the operating mode of the timer. Note that the timer module should be disabled before being configured. The configuration is defined by one of the following macros:

  `TIMER_CFG_32_BIT_OS` 32-bit one-shot timer, counts down

  `TIMER_CFG_32_BIT_OS_UP` 32-bit one shot timer, counts up

  `TIMER_CFG_32_BIT_PER` 32-bit periodic timer, counts down

  `TIMER_CFG_32_BIT_PER_UP` 32-bit periodic timer, counts up

  `TIMER_CFG_32_RTC` 32-bit real time clock.

  `TIMER_CFG_16_BIT_PAIR` two 16-bit timers.

  When configured for use as two 16-bit timers, each timer is configured independently. The first timer is configured by setting `ulConfig` to be the result of a logical OR operation between one of the following values and ulConfig.

  `TIMER_CFG_A_ONE_SHOT` 16-bit one-shot timer

  `TIMER_CFG_A_ONE_SHOT_UP` 16-bit one-shot timer that counts up instead of down

  `TIMER_CFG_A_PERIODIC` 16-bit periodic timer

  `TIMER_CFG_A_PERIODIC_UP` 16-bit periodic timer that counts up instead of down

  `TIMER_CFG_A_CAP_COUNT` 16-bit edge count capture

  `TIMER_CFG_A_CAP_COUNT_UP` 16-bit edge count capture that counts up instead of down

  `TIMER_CFG_A_CAP_TIME` 16-bit edge time capture

  `TIMER_CFG_A_CAP_TIME_UP` 16-bit edge time capture that counts up instead of down

  `TIMER_CFG_A_PWM` 16-bit PWM output

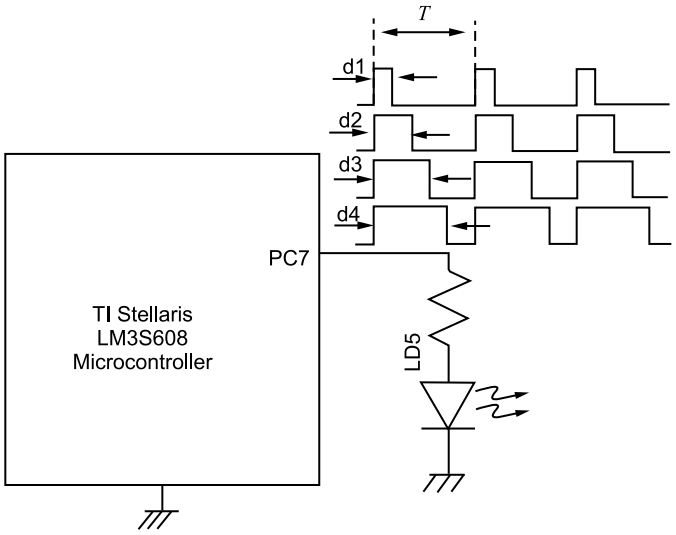

Figure 9.5 *Block Diagram for Experiment 10*

Similarly, the second timer is configured by setting `ulConfig` to the result of a logical OR operation between one of the corresponding `TIMER_CFG_B_*` macros and `ulConfig`.

Returns: None

- **TimerLoadSet** Sets the timer load value.

  Prototype: void TimerLoadSet(unsigned long `ulBase`, unsigned long `ulTimer`, unsigned long `ulValue`)

  Parameters: `ulBase` is the base address of the timer module. `ulTimer` specifies the timer that need to be adjusted—must be one of `TIMER_A`, `TIMER_B`, or `TIMER_BOTH`. `ulValue` is the load value.

  Description: This function sets the timer load value. If the timer is already running when the function is called, the value is loaded immediately. Otherwise, the value is loaded into the counter when it is triggered with `TimerEnable` (explained below).

  Returns: None

- **TimerLoadGet** Gets the timer load value.

  Prototype: unsigned long `TimerLoadGet`(unsigned long `ulBase`, unsigned long `ulTimer`);

  Parameters: `ulBase` is the base address of the timer module. `ulTimer` specifies the timer.

  Description: The function returns the currently programmed load value for the specified timer.

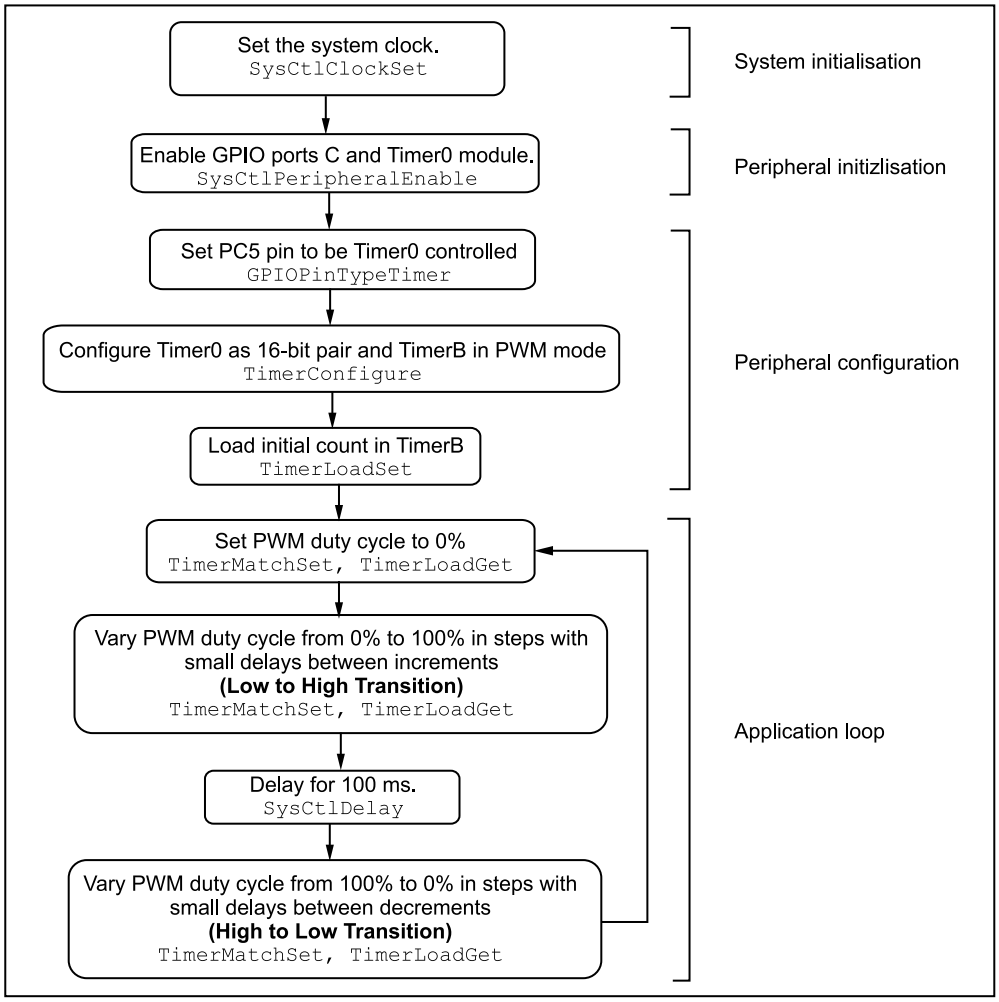

Figure 9.6 *Program Flow for Experiment 10*

Returns: The load value of the timer.

- **TimerMatchSet** Sets the timer match value.

  Prototype: `void TimerMatchSet`(unsigned long `ulBase`, unsigned long `ulTimer`, unsigned long `ulValue`

  Parameters: `ulBase` is the base address of the timer module. `ulTimer` specifies the timer to be adjusted. `ulValue` is the match value.

  Description: This function sets the match value for a timer. When the timer count value of the counter matches the `ulValue`, an interrupt is triggered. However, the

timer continues to count until it wraps onto zero. This is used in 'capture count' mode to determine when to interrupt the processor, and in PWM mode to determine the duty cycle of the output signal.

Returns: None

- **TimerEnable** Enables the timer.

  Prototype: `void TimerEnable`(unsigned long `ulBase`, unsigned long `ulTimer`)

  Parameters: `ulBase` is the base address of the timer module. `ulTimer` specifies the timer to be enabled.

  Description: This function enables the operation of the timer module. The timer must be configured before it is enabled.

  Returns: None

- **TimerDisable** Disables the timer.

  Prototype: `void TimerDisable`(unsigned long `ulBase`, unsigned long `ulTimer`)

  Parameters:`ulBase` is the base address of the timer module. `ulTimer` specifies the timer to be disabled.

  Description: This function disables the operation of the timer module.

  Returns: None

9.5 Experiment 11

9.5.1 Objective

To evaluate the functioning of the Watchdog timer.

9.5.2 Program Flow

In this experiment we see the working of a Watchdog timer. The timer is programmed to reset itself after a specific period of time. The LED LD7 is set to blink and the 'Reset' is transmitted serially every time the system resets. On pressing the button S2, the reset and the interrupt on the Watchdog timer are disabled. The Watchdog timer keeps resetting the system after a specific interval of time. This is seen on the terminal application as the blinking LED (LD7). The algorithm for this experiment is:

1. Enable the system clock, GPIO ports A, C, E and the Watchdog timer module.
2. Configure the GPIO pins for the LEDs and the button.
3. Enable the interrupts on GPIO Port E.

4. Make PA0 and PA1 pins UART-controlled.
5. Set the Watchdog timer configuration.
6. Enable the Watchdog timer and its interrupt.
7. Blink the LED LD7 (PC5).
8. If button S2 is pressed, the Watchdog reset and interrupt is disabled and the LEDs are turned on.

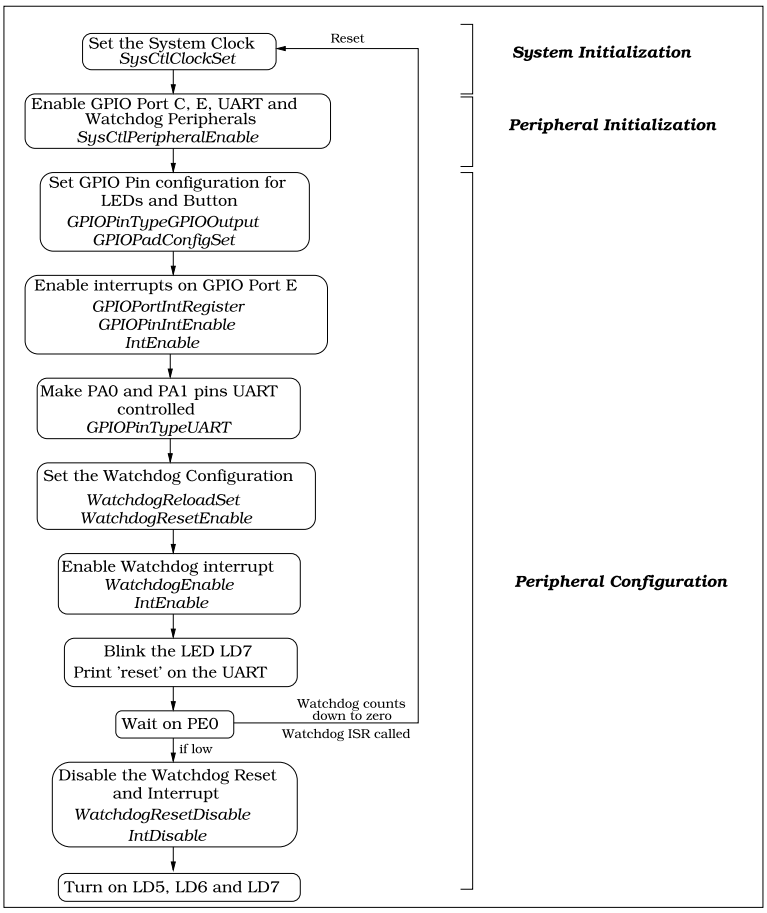

Figure 9.7 *Program flow for Experiment 11*

9.5.3 Suggested StellarisWare API Function Calls

This driver is contained in `driverlib/watchdog.c`, with `driverlib/watchdog.h` containing the API definitions for use by applications.

- **WatchdogReloadSet** Sets the Watchdog timer reload value.

Prototype:`void WatchdogReloadSet`(unsigned long `ulBase`, unsigned long `ulLoadVal`)

Parameters:`ulBase` is the base address of the watchdog timer module. `ulLoadVal` is load value for the watchdog timer.

Description: This function sets the value to load into the watchdog timer when the count reaches zero for the first time; if the watchdog timer is running when this function is called, then the value will be immediately loaded into the watchdog timer counter. If the ulLoadVal parameter is 0, then an interrupt is immediately generated.

Returns: None

- **WatchdogResetEnable** Enables the Watchdog timer reset.

  Prototype: `void WatchdogResetEnabe`(unsigned long `ulBase`)

  Parameters: `ulBase` is the base address of the Watchdog timer module.

  Description: Enables the capability of the watchdog timer to issue a reset to the processor upon a second time-out condition.

  Returns: None

- **WatchdogEnable** Enables the watchdog timer.

  Prototype:`void WatchdogEnable`(unsigned long `ulBase`)

  Parameters: `ulBase` is the base address of the Watchdog module.

  Description: This will enable the watchdog timer counter and interrupt.

  Returns: None

- **WatchdogIntClear** Clears the watchdog timer interrupts.

  Prototype: `void WatchdogIntClear`(unsigned long `ulBase`)

  Parameters: `ulBase` is the base address of the Watchdog module.

  Description: The watchdog timer interrupt source is cleared so that it no longer asserts.

  Returns: None

9.6 Exercises

1. Turn on LED LD5 for 5 seconds using a timer. During this 5-seconds Window, count the number of times switch S2 is pressed and released. After the 5-seconds window is over, make LED LD6 blink the number of times S2 was pressed.

2. Use a random number generator based on a linear feedback shift register (LFSR). Output the value of the random number generator, one bit at a time on LED LD5, every 100 ms using a timer. Also try this experiment with 10 ms and compare the outputs.

3. Use a timer as a random number generator for electronic dice. The dice number is displayed on LEDs, LD5, LD6 and LD7, in binary form. Poll switch S2. When S2 is pressed, capture the value of the timer and use mod-6 function to get a random number between 1 and 6 and display it on the LEDs.

10 Serial Communication with the UART

10.1 Introduction

Computers transfer data in two ways, serial and parallel. In parallel communication, often one or more data lines are required to transmit data to devices that are placed only a few meters away. Devices which use parallel mode of communication are printers, hard disks, etc. If data is to be transferred over longer distances, serial communication is used. In contrast to parallel communication, data is sent on a single line, one bit at a time, in serial communication. Serial communication using the Stellaris® Guru is the topic of this chapter. The Stellaris® LM3S608 has serial communication capability built into it, thereby making fast data transfer using a few wires possible.

The Stellaris® Universal Asynchronous Receiver/Transmitter (UART) has the following features:

- Two fully programmable 16C550-type UARTs
- Programmable baud rate generator
- Separate 16× 8 transmit (Tx) and receive (Rx) FIFO buffers
- Automatic generation and stripping of start, stop, parity bits
- Programmable serial interface
  - 5, 6, 7 or 8 data bits
  - Even, odd, stick or no parity bit generation
  - 1 or 2 stop-bit generation
  - Baud rate generator
- Modem/Flow control
- 9-bit operation
- Line-break generation and detection

Serial communication on the Stellaris® Guru is achieved using a FTDI FT232 UART-to-USB bridge that allows the board to be detected as a standard virtual COM port on the computer. Data is sent and received in a standard manner as would have been done when using a RS232 interface. Figure 10.1 shows the block diagram of the UART module.

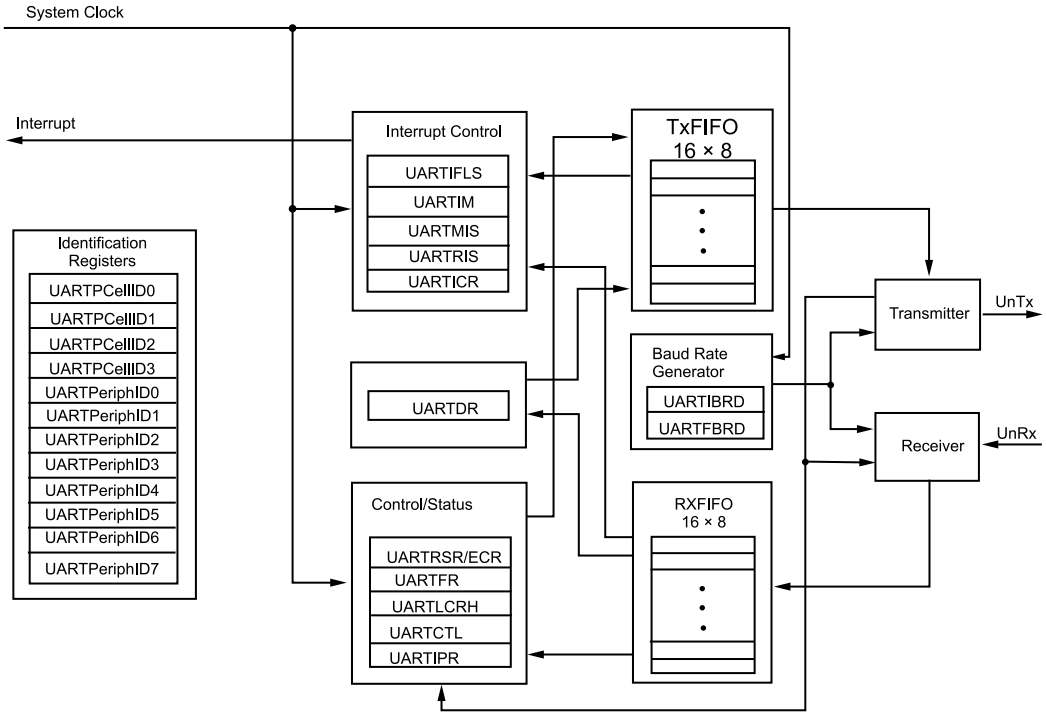

Figure 10.1 *UART block diagram (Stellaris® LM3S608 microcontroller datasheet)*

10.2 Functional Description

10.2.1 Half and Full Duplex Transmission

Serial communication can be further classified on the basis of the direction of data on the communication line. If data can be both transmitted and received, it is duplex transmission. If data can only be transmitted or received, it is simplex transmission, as in printers where data is only sent. Duplex transmission can be further classified into half duplex and full duplex. In half duplex, at a particular instant of time, data is either being transmitted or being received, in contrast to full duplex transmission where bidirectional communication takes place. Hence, full duplex requires two conductors for communication whereas simplex and half duplex require only one conductor. Figure 10.2 highlights the several modes of serial communication.

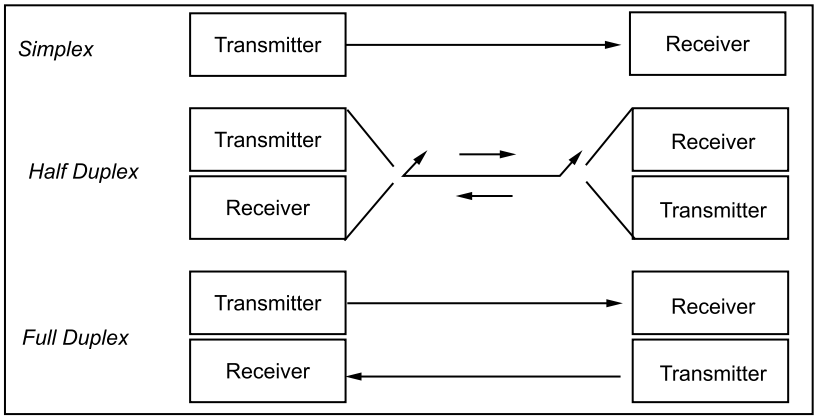

Figure 10.2 *Serial modes*

10.2.2 Serial Communication and Data Framing

In asynchronous serial communication, each character data is placed between the start and stop bits. This is called framing. The start bit is always one bit but the stop bit can be one or two bits. The start bit is 0 (low) and the stop bit is 1 (high). The control logic outputs the serial bit stream beginning with a start bit, and followed by the data bits (LSB first), parity bit, and the stop bits according to the programmed configuration in the control registers. Figure 10.3 shows an example of a data frame.

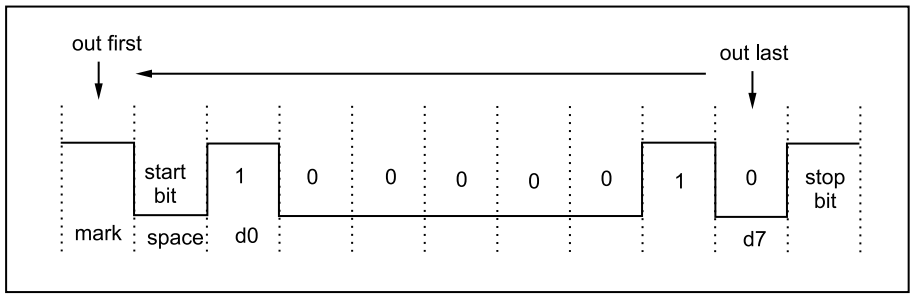

Figure 10.3 *Data frame*

10.3 Experiment 12

10.3.1 Objective

Receive a byte over UART, increment it by one and transmit it back. The configuration of the UART module is 115200, 8, N, 1 (baud rate = 115200 kbps, data length = 8 bits, parity = none and stop bits = one).

10.3.2 Program Flow

Serial mode of data transfer is one of the many modes in which a microcontroller can communicate. The block diagram for the experiment is shown in Figure 10.4. This experiment requires us to receive a byte over UART, increment the byte and then transmit the new byte over UART. For example, if A is received the reply should be B or if 3 is received the reply should be 4. The algorithm to be followed is:

1. Enable the System Clock, GPIO port A and the UART module.
2. Set the direction of the GPIO port and configure the UART module for settings 115200, 8, N, 1.
3. Recieve a byte over UART.
4. Increment the byte.
5. Transmit the byte back over UART and view on serial monitor.

Figure 10.5 shows a suggested program flow for this experiment.

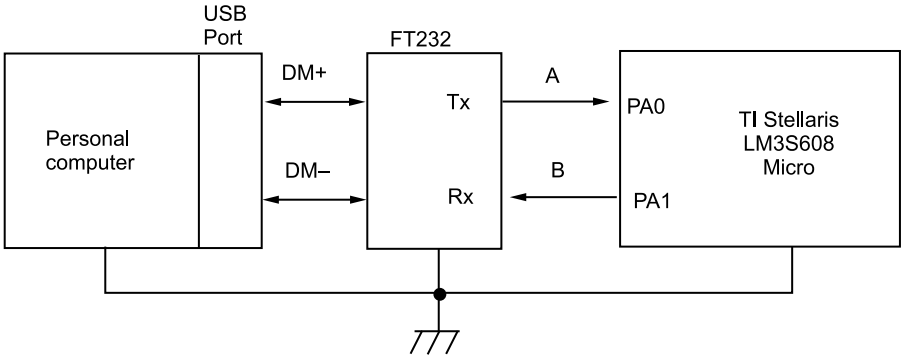

Figure 10.4 *Block diagram for Experiment 12*

10.3.3 Suggested StellarisWare API Function Calls

The functions used to drive the UART module are contained in `driverlib/uart.c` with `driverlib/uart.h` containing the API definitions for use by application.

- **UARTConfigSetExpClk** Sets the configuration of the UART module.

  Prototype: `void UARTConfigSetExpClk`(unsigned long `ulBase`, unsigned long `UARTClk`, unsigned long `ulBaud`, unsigned long `ulConfig`);

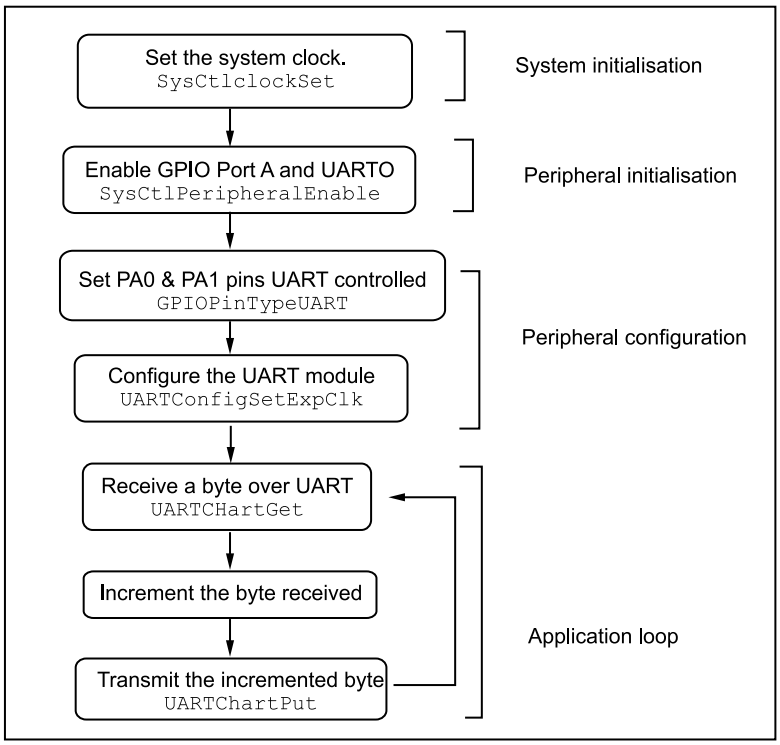

Figure 10.5 *Program flow for Experiment 12*

Parameters: `ulBase` is the base address of the UART port. `ulUARTClk` is the rate of the clock supplied to the UART module. `ulBaud` is the desired baud rate. `ulConfig` is the data format for the port.

Description: This function configures the UART for operation in the specified data frame format. The baud rate is specified in the ulBaud parameter and the data format in the ulConfig parameter.

The `ulConfig` is the logical OR of three parameters, the word length, 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.
`UART_CONFIG_STOP_ONE` & `UART_CONFIG_STOP_TWO` select one or two stop bits.
`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).

The peripheral clock is the same as the processor clock and can be obtained by using `SysCtlClockGet()`.

Returns: None

- **UARTCharGet** Waits for a character from the specified port.

  Prototype: `long UARTCharGet`(unsigned long `ulBase`)

  Parameters: `ulBase` is the base address of the UART port.

  Description: This function gets a character from the receive FIFO for the specified port. If there are no characters available, this function waits until a character is received before returning.

  Returns: Returns the character read from the specified port, cast as a long.

- **UARTCharPut** Waits for a character from the specified port.

  Prototype: `void UARTCharGet`(unsigned long `ulBase`, unsigned char `ucData`)

  Parameters: `ulBase` 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 FIFO for the specified port. If there is no space available in the transmit FIFO, this function waits until there is space available before returning.

  Returns: None

10.4 Experiment 13

10.4.1 Objective

To control the intensity of LED, LD5, connected on PC7 using software PWM using parameters received over UART. Configuration of the UART is 115200, 8, N, 1.

10.4.2 Program Flow

This experiment allows us to control the intensity of an LED using parameters received over UART. The block diagram is shown in Figure 10.6. The parameter is a number between 0 to 10, where 0 signifies the least intensity and 10 signifies the highest intensity. Drawing on the algorithm from the previous experiment with the addition of software PWM, the new algorithm is:

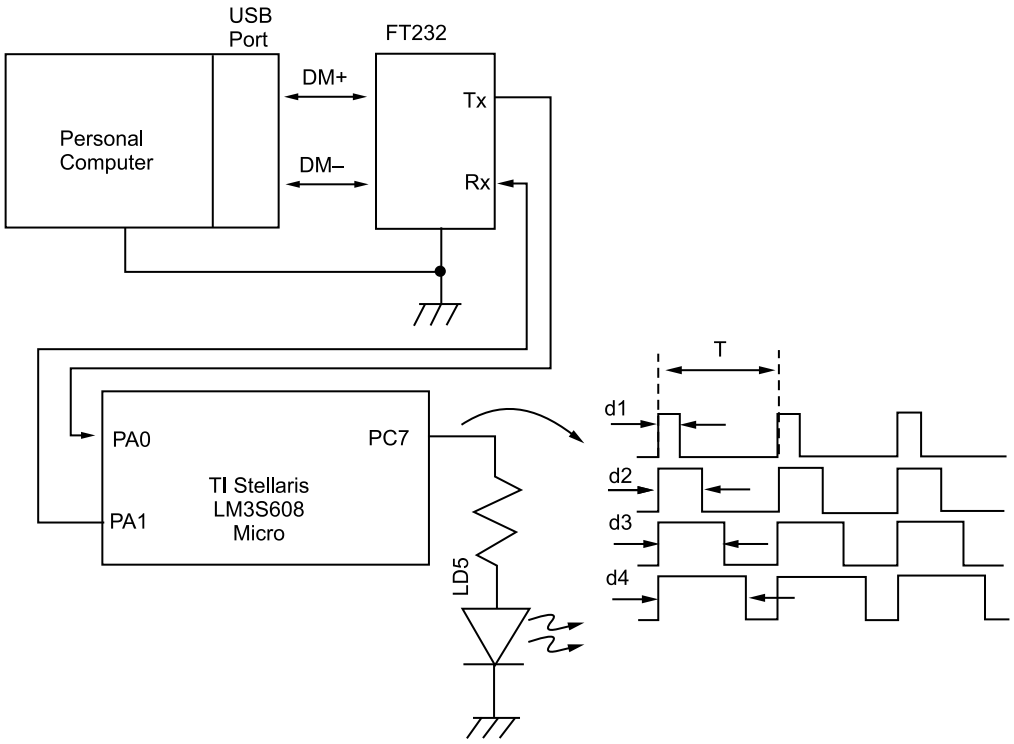

Figure 10.6 *Block diagram for Experiment 13*

1. Enable the system clock, GPIO ports C and A and the UART module.
2. Set the direction of GPIO ports A and C and configure the UART module for setting 115200, 8, N, 1.
3. Receive the parameter over UART.
4. Depending on the magnitude of the parameter, vary the intensity of LD5 using PWM.
5. Continue PWM until a new parameter is received.

The program flow for this experiment is straightforward and is shown in Figure 10.7.

10.5 Experiment 14

10.5.1 Objective

Measure the light intensity using the LED light sensor on the Stellaris® Guru and send measured values to a PC to plot them in a graphing utility with respect to time. You may also view the intensity values on a serial monitor. Configuration of the UART is 115200, 8, N, 1.

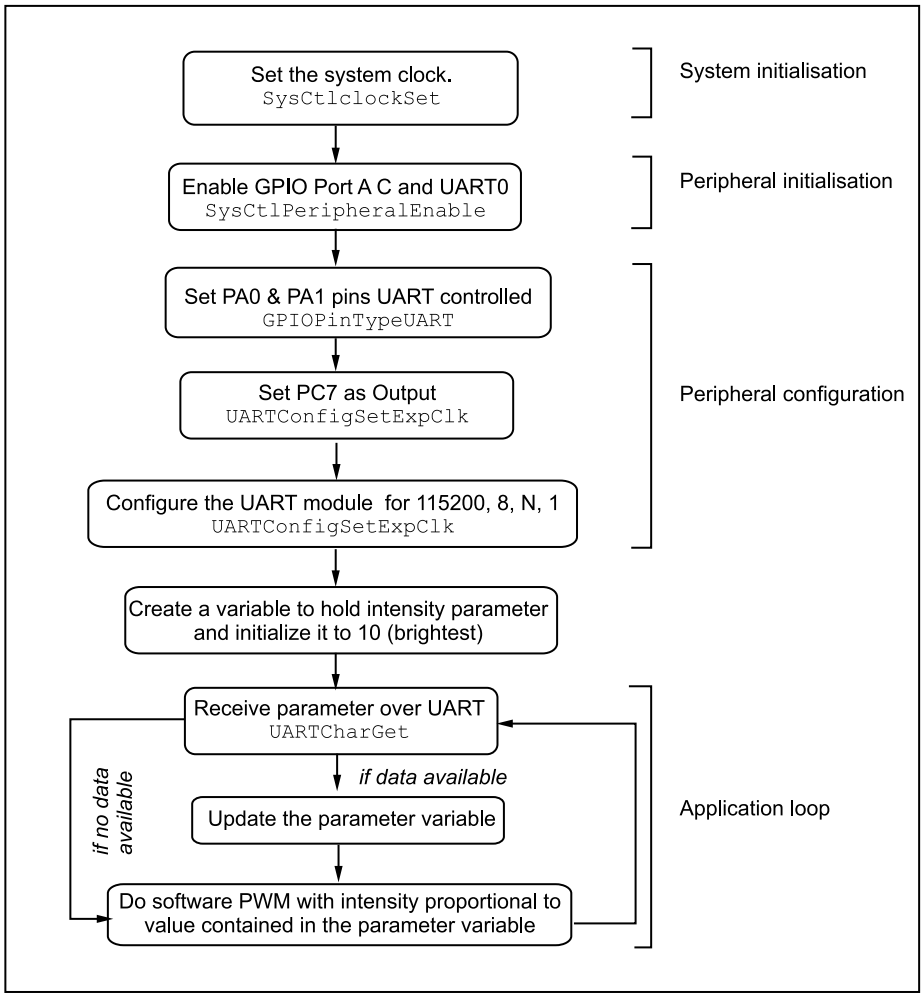

Figure 10.7 *Program flow for Experiment 13*

10.5.2 Program Flow

In this experiment, we use an LED as an ambient light sensor. The block diagram for the experiment is shown in Figure 10.8. An LED can be used as a light sensor in reverse bias. The algorithm for this experiment is:

- Enable the system clock, GPIO ports A and D and the UART module.
- Set the direction of GPIO port A and configure the UART module for 115200, 8, N, 1.

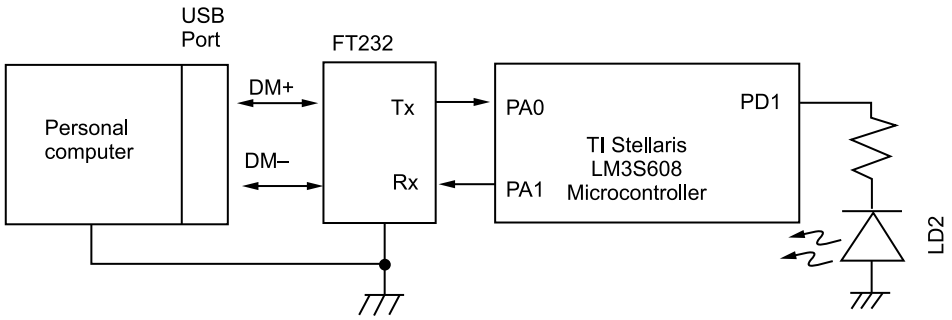

Figure 10.8 *Block diagram for Experiment 14*

- Set the direction of GPIO port D pin 1 as output.
- Make PD1 high for 100 ms and then change the data direction of pin to input.
- Start a counter and wait for PD1 to go low.
- Once PD1 is low, the count is proportional to the ambient light intensity.
- Scale counter value accordingly and transmit over UART.
- Use a plotting application to plot a real time graph of ambient light intensity.

There are many ways to plot waveforms on a computer. It is up to you what method you want to employ for this experiment. Some of the methods are:

1. Store incoming value to a text file and then plot using Microsoft Excel.
2. Develop a custom PC application communicating with the serial port.

Figure 10.9 shows a suggested program flow for the experiment.

10.6 Experiment 15

10.6.1 Objective

To simulate a real-time clock using the general purpose timer module and output time over UART in hh:mm:ss format.

10.6.2 Program Flow

In this experiment we maintain time using the Guru kit. The time keeping is done by the microcontroller and the current time can be viewed on a serial monitor. The current time is set by sending parameters over UART. We set up a timer to generate an interrupt every second which increments variables suitably to maintain time. In every interrupt routine, the current time is sent over UART for display on the terminal screen. The block diagram for the experiment is shown in Figure 10.10.

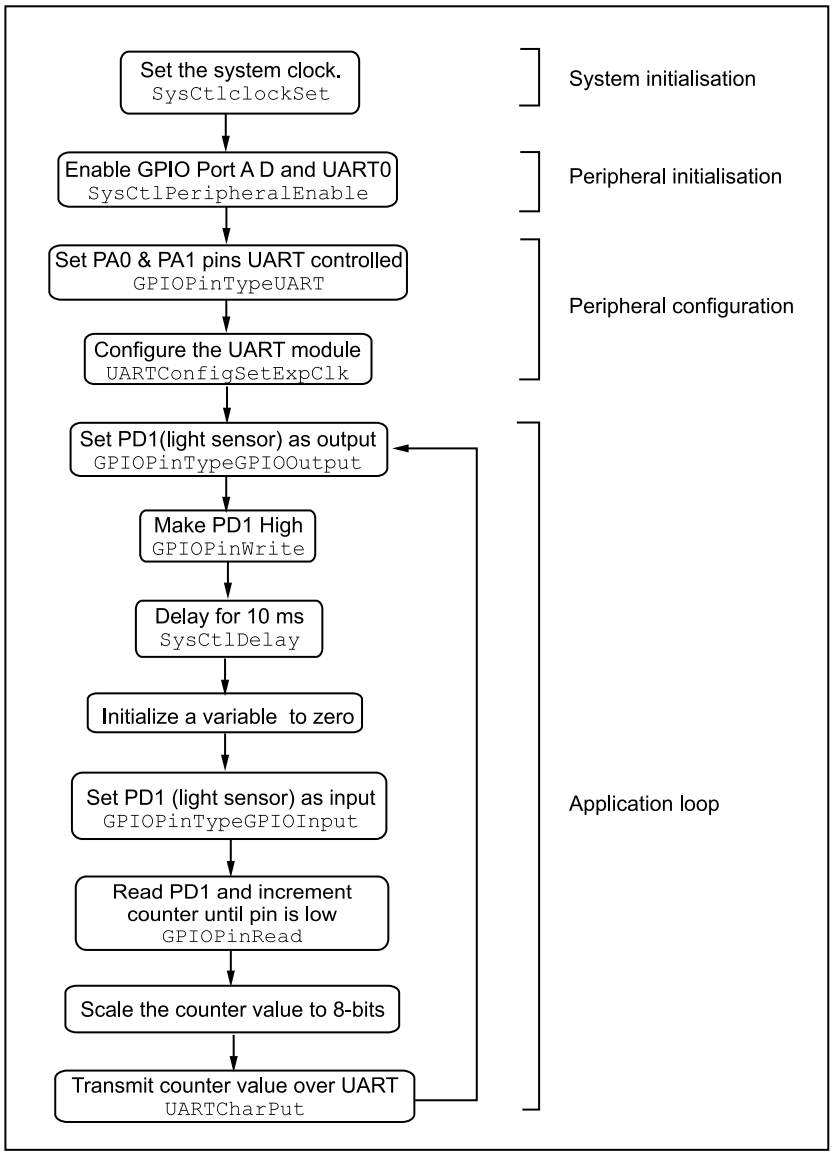

Figure 10.9 *Program flow for Experiment 14*

An algorithm for this experiment is:

1. Enable the system clock, GPIO port A and Timer0 module.
2. Configure PA0 and PA1 as UART and configure for 115200, 8, N, 1.
3. Enable 32-bit operation in Timer0 module and set the trigger period to 1 second.
4. Receive the current time parameters over UART and store the variables.

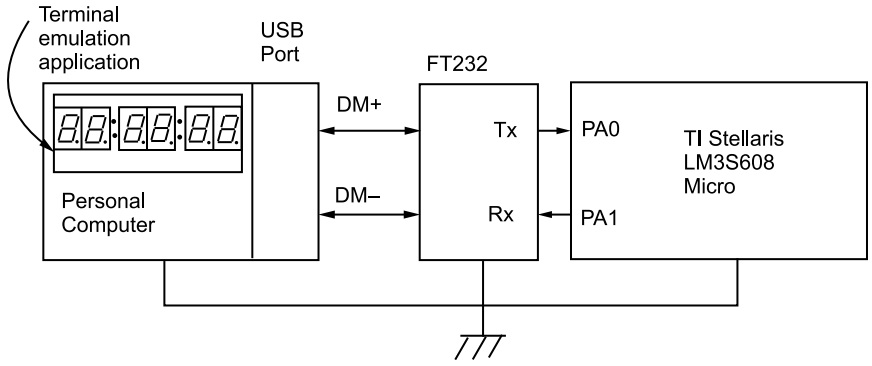

Figure 10.10 *Block diagram for Experiment 15*

5. Enable interrupt generation by the timer module.
6. Whenever an interrupt occurs, increment the seconds variable.
7. When the seconds variable is sixty, increment the minutes variable and reset the seconds. Do the same when the minutes variable overflows.
8. Send the new time over UART for display on the terminal application.

A suggested program flow is shown in Figure 10.11 on the following page.

10.7 Exercises

1. Instead of hardware UART, design a software-driven UART (also called bit-bang UART) to transmit data to a computer using the 115200, 8, N, 1 format.
2. Design a software-driven UART receiver to receive data from a computer using the 115200, 8, N, 1 format. The computer should transmit the 8-bit intensity values that the program on the Guru should receive and set the LED LD5 intensity accordingly.
3. Design an interrupt-driven UART communication protocol in which the blinking rate of an LED is changed with respect to a parameter received over UART. Instead of polling the UART module, an interrupt is to be generated when data is received by the module.
4. Maintain an up/down counter using Guru. The count is to be maintained using timers and displayed on a terminal emulation program.
5. Connect two Guru kits together, one as the master and the other as slave. The master should send instructions to vary the blinking pattern of LEDs on the slave board.
6. Implement dice using LFSR random number generator and output a new value over UART every time switch S2 is pressed.
7. Connect a computer and two Guru kits in series. Using a terminal emulation program running on the computer, send commands to light up LEDs on either of the kits.

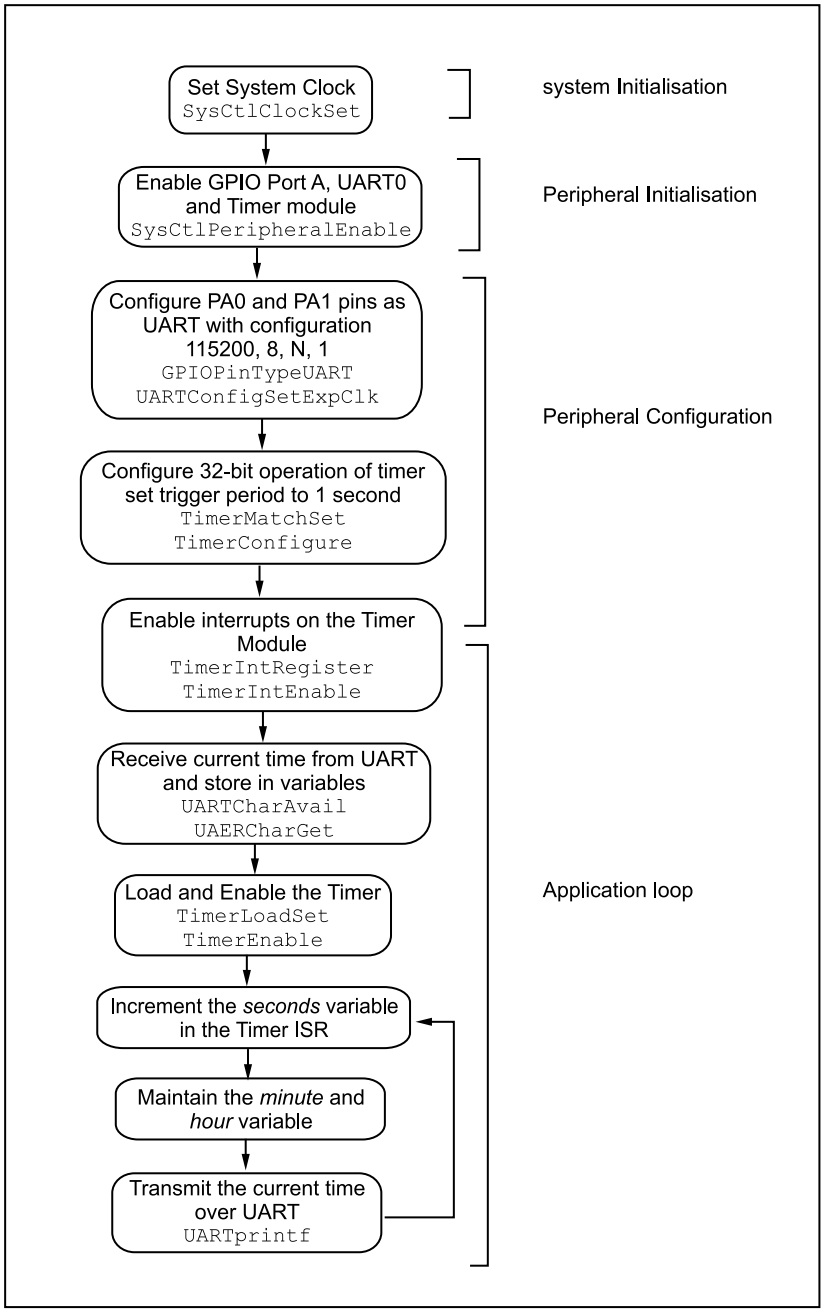

Figure 10.11 *Program flow for Experiment 15*

11 Analog-to-Digital Converter

An analog-to-digital converter (ADC) is a peripheral that converts a continuous analog voltage to a discrete digital number. This chapter deals with the ADC module on the Stellaris® family of microcontrollers and explains its usage through many experiments.

11.1 Introduction

The Stellaris ADC module features 10-bit conversion resolution, supports eight input channels, and comes with an internal temperature sensor. The ADC module contains four programmable sequencers that allow for the sampling of multiple analog input sources without any controller intervention. Each sample sequence provides flexible programming with fully configurable input source, trigger events, interrupt generation, and sequence priority.

The Stellaris® ADC module has the following features:

- Eight analog input channels
- Single ended and differential input configurations
- On-chip internal temperature sensor
- Sample rate of 500 thousand samples per second
- Four programmable sample conversion sequences from one to eight entries long, with corresponding conversion result FIFOs
- Hardware-averaging of up to 64 samples for improved accuracy
- Converter using an internal 3-V reference

Four sampling sequences, each with configurable trigger events, can be captured. The first sequence will capture up to eight samples, the second and third sequences will capture up to four samples, and the fourth sequence will capture a single sample. Each sample can be the same channel, different channels, or any combination in any order. Hardware oversampling of the ADC improves accuracy. An oversampling factor of 2x, 4x, 8x, 16x, 32x and 64x is supported but it reduces the throughput of the ADC by a commensurate factor. Hardware

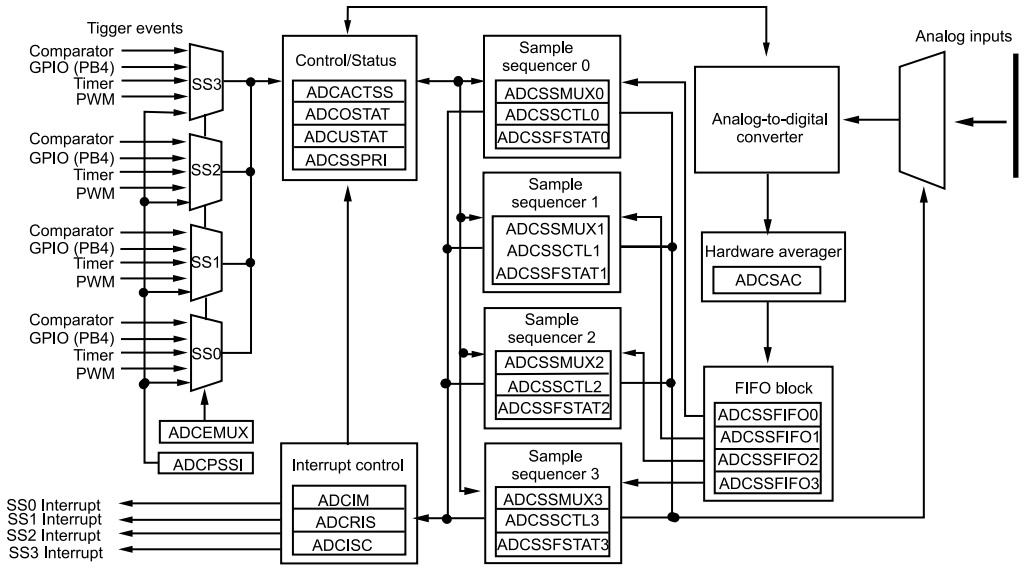

Figure 11.1 *ADC module block diagram (Stellaris® LM3S608 microcontroller datasheet)*

oversampling is applied uniformly across all sample sequences. Software oversampling of the ADC is also available but reduces the depth of the sample sequences by a corresponding amount. The block diagram of the ADC module on the LM3S608 is shown in Figure 11.1.

11.2 Functional Description

The analog-to-digital converter (ADC) API provides a set of functions for dealing with ADC. Functions are provided to configure sample sequencers, read the captured data, register a sample sequence interrupt handler, and handle interrupt masking/clearing.

In order to build a simple ADC, two parts are required: a sample-and-hold circuit and a circuit that determines the voltage held by the sample-and-hold circuit. A simple way to implement a sample-and-hold circuit is to have an input fed to a capacitor through a switch. To sample the input, one closes the switch between the input and the capacitor, and to hold the sample value, one opens the switch.

The Stellaris® ADC collects sample data by using a programmable sequence-based approach instead of the conventional single or double sampling sequences found on many ADC modules. Each programmed sequence is a series of consecutive samples, allowing the ADC to collect multiple input sources without having to be reconfigured. The programming of each sample in the sample sequence includes parameters such as the input mode (differential or single-ended) and source; interrupt generation on sample completion and indicator for the last sample in the sequence.

11.3 Experiment 16

11.3.1 Objective

Take analog readings on the rotation of a rotary potentiometer connected to ADC7 of LM3S608 on the Stellaris® Guru, and output the reading over UART0 and display it on the terminal.

11.3.2 Program Flow

The world is analog. To use analog quantities in a digital system, they first need to converted to discrete values using sampling and quantization technqiues. This experiment demonstrates how we can convert movement of the rotary potentiometer to discrete digital values and then output them over the UART. Figure 11.2 shows the block diagram of the setup required for this experiment. The 10-bit ADC converts the voltage from the potentiometer to a value between 0 and 1023.

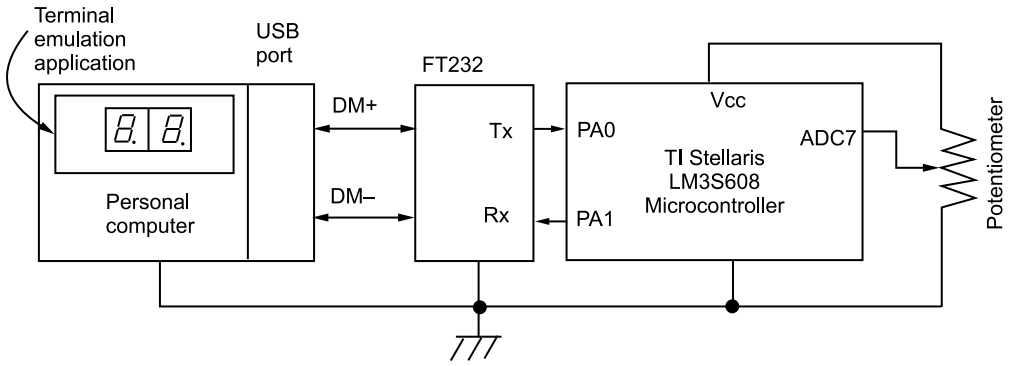

Figure 11.2 *Block diagram for Experiment 16*

The algorithm to be followed is:

1. Enable the system clock, GPIO port A, the UART and the ADC module.
2. Configure the UART and the ADC peripheral for appropriate function.
3. Read the analog value of the potentiometer through ADC Channel 7 and output on UART0 for display on a serial monitor.
4. Move the potentiometer and see changes in the sampled value.

A suggested program flow for the experiment is shown in Figure 11.3.

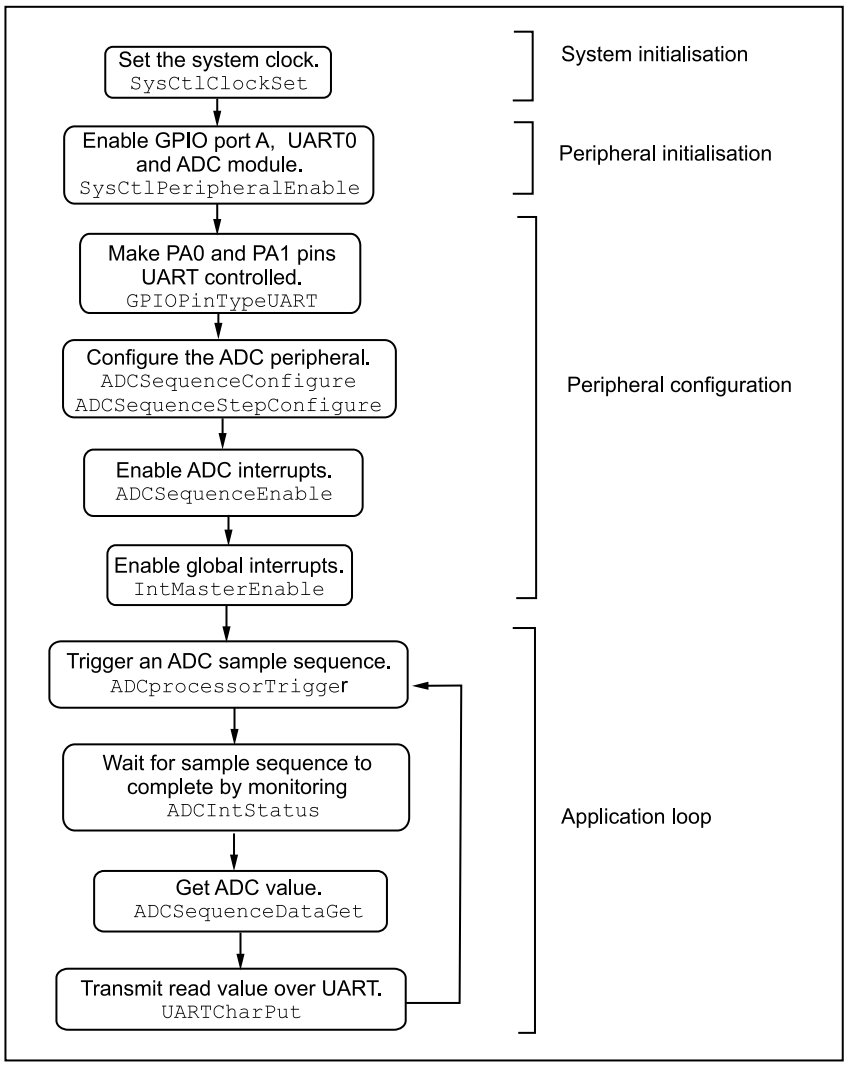

Figure 11.3 *Program flow for Experiment 16*

11.3.3 Suggested StellarisWare API Function Calls

The functions used to drive the ADC module are contained in `driverlib/adc.h` with `driverlib/adc.c` containing the API definitions for use by the application.

- **ADCSequenceConfigure** Configures the trigger source and priority of a sample sequence.

  Prototype: `void ADCSequenceConfigure`(unsigned long `ulBase`, unsigned

long `ulSequenceNum`, unsigned long `ulTrigger`, unsigned long `ulPriority`)

Parameters: `ulBase` is the base address of the ADC module. `ulSequenceNum` is the sample sequence number. `ulTrigger` is the trigger sequence that initiates the A/D conversion. `ulPriority` is the relative priority of the sample sequence.

Description: This function configures the initiation criteria for a sample sequence. Valid sample sequences range from zero to three; sequence zero will capture up to eight samples, sequences one and two will capture up to four samples, and sequence three will capture a single sample.

The `ulTrigger` parameter can take on the following values:

```
ADC_TRIGGER_PROCESSOR,   ADC_TRIGGER_COMP0,
ADC_TRIGGER_COMP1,       ADC_TRIGGER_EXTERNAL,
ADC_TRIGGER_TIMER,       ADC_TRIGGER_PWM0,
ADC_TRIGGER_PWM1,        ADC_TRIGGER_PWM2,
ADC_TRIGGER_PWM3 and     ADC_TRIGGER_ALWAYS
```

The `ulPriority` parameter is a value between 0 and 3, where 0 represents the highest priority and 3 the lowest.

Returns: None

- **ADCSequenceStepConfigure** Configure a step of the sample sequencer.

  Prototype: `void ADCSequenceStepConfigure`(unsigned long `ulBase`, unsigned long `ulSequenceNum`, unsigned long `ulStep`, unsigned long `ulConfig`)

  Parameters:`ulBase` is the base address of the ADC module. `ulSequenceNum` is the sample sequence number. `ulStep` is the step to be configured. `ulConfig` is the configuration of this step.

  Description: This function will set the configuration of the ADC for one step of a sample sequence. The ADC can be configured for single-ended or differential operation (the `ADC_CTL_D` bit selects differential operation when set), the channel to be sampled can be chosen (the `ADC_CTL_CH0` through `ADC_CTL_CH15` values), and the internal temperature sensor can be selected (the `ADC_CTL_TS` bit). Additionally, this step can be defined as the last in the sequence (the `ADC_CTL_END` bit) and it can be configured to cause an interrupt when the step is complete (the `ADC_CTL_IE` bit). If digital comparators are present on the device, this step may also be configured to send the ADC sample to the selected comparator using `ADC_CTL_CMP0` through `ADC_CTL_CMP7`. The ADC uses the configuration at the appropriate time when the trigger for this sequence occurs.

  The `ulStep` parameter determines the order in which the samples are captured by the ADC when the trigger occurs. It can range from zero to seven for the first sample

sequence, from zero to three for the second and third sample sequence, and can only be zero for the fourth sample sequence.

Returns: None

- **ADCSequenceEnable** Enables a sample sequence

  Prototype: `void ADCSequenceEnable`(unsigned long `ulBase`, unsigned long `ulSequenceNum`)

  Paramters: `ulBase` is the base address of the ADC module. `ulSequenceNum` is the sample sequence number.

  Description: Allows the specified sample sequence to be captured when its trigger is detected.

  Returns: None

- **ADCIntClear** Clears the sample sequence interrupt source.

  Prototype: `void ADCIntClear`(unsigned long `ulBase`, unsigned long `ulSequenceNum`)

  Parameters: `ulBase` is the base address of the ADC module. `ulSequenceNum` is the sample sequence number.

  Description: The specified sample sequence interrupt is cleared so that it no longer asserts. This is done in the interrupt handler to keep it from being called again immediately upon exit.

  Returns: None

- **ADCIntStatus** Gets the current interrupt status.

  Prototype: unsigned long `ADCIntStatus`(unsigned long `ulBase`, unsigned long `ulSequenceNum, tBoolean bMasked`)

  Parameters: `ulBase` is the base address of the ADC module. `ulSequenceNum` is the sampling sequence number. `bMasked` is false if raw interrupt status is required and true if masked interrupt status.

  Description: This returns the interrupt status for the specified sample sequence.

  Returns: The current raw or masked interrupt status.

- **ADCSequenceDataGet** Gets the captured data for a sample sequence.

  Prototype: long `ADCSequenceDataGet`(unsigned long `ulBase`, unsigned long `ulSequenceNum`, unsigned long `*pulBuffer`)

  Parameters: `ulBase` is the base address of the ADC module. `ulSequenceNum` is the sample sequence number. `pulBuffer` is the address where the data is required.

Description: This function copies data from the specified sample sequence output FIFO to a memory-resident buffer. The number of samples available in the hardware FIFO are copied into the buffer, which is assumed to be large enough to hold that many samples.

Returns: None

11.4 Experiment 17

11.4.1 Objective

Measure the ambient temperature using onboard LM35 and display the temperature magnitude by varying the intensity of the RGB LED.

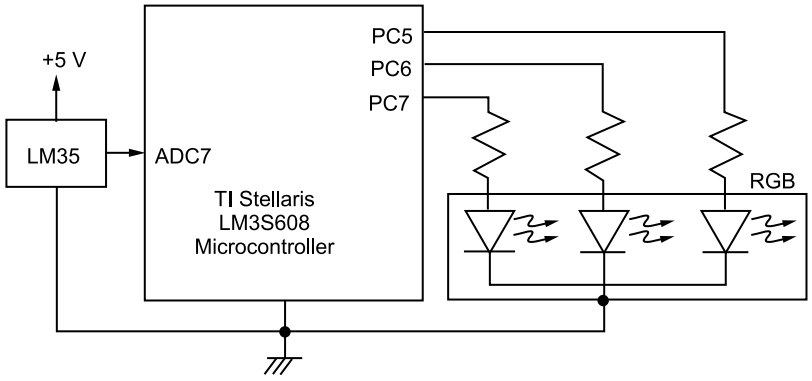

Figure 11.4 *Block diagram for Experiment 17*

11.4.2 Program Flow

In this experiment, we wish to simulate the magnitude of the ambient temperature using an RGB LED. We vary the intensity of the LED with changes in the ambient temperature. Figure 11.4 shows the block diagram of the setup required for this experiment. For example, in case the temperature is below 10 degrees Celsius the RGB LED is blue, and as the temperature rises to above 40 degrees Celsius, the RGB LED turns red. LM35 gives an output of 10 mV/Celsius. The algorithm is as follows:

1. Enable the system clock, GPIO and the ADC peripheral.
2. Sample temperature using the ADC Channel 7.
3. Scale the 10-bit value accordingly.
4. Depending on the temperature value, simulate it on the RGB LED using hardware/software PWM.

The program flow for the experiment is shown in Figure 11.5.

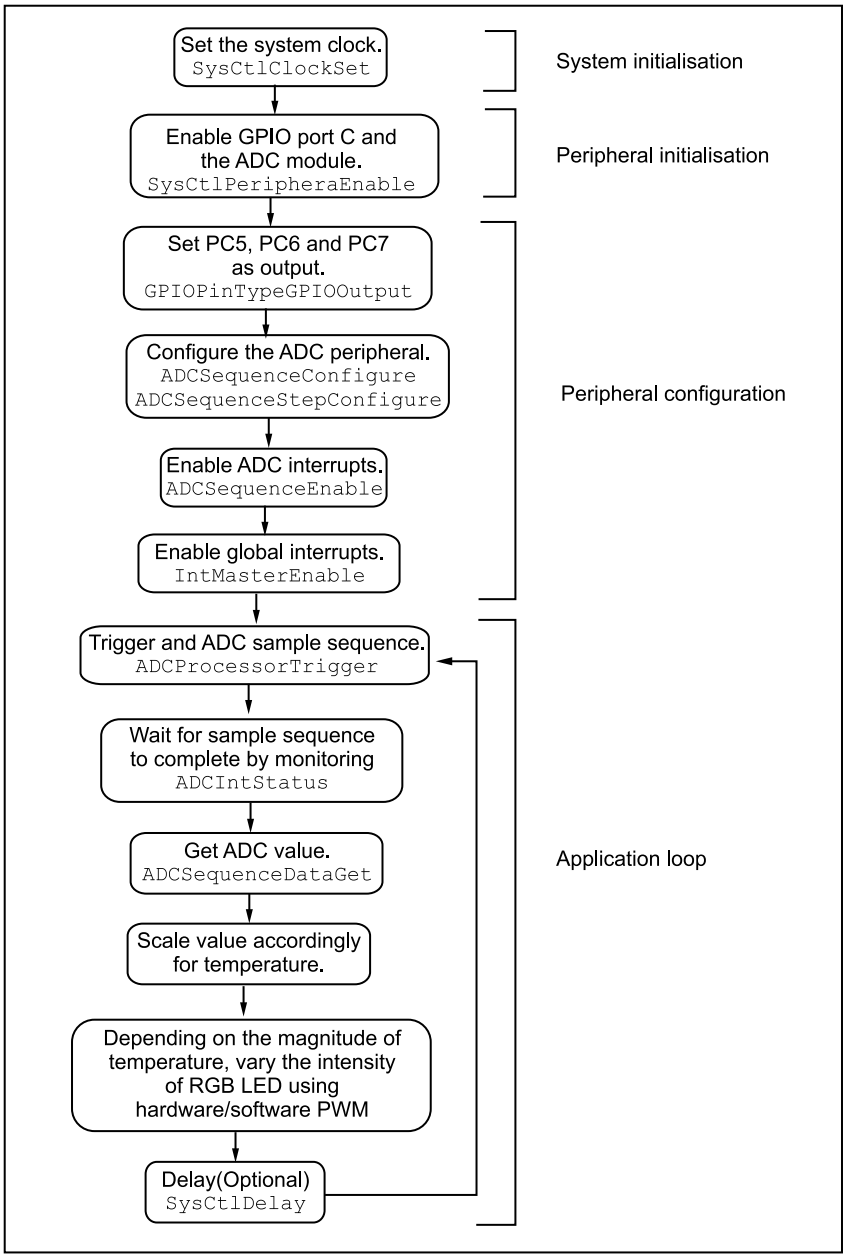

Figure 11.5 *Program flow for Experiment 17*

11.5 Experiment 18

11.5.1 Objective

Measure the ambient temperature using onboard LM35 and send the temperature values in degrees Celsius over the UART for display in the terminal program.

11.5.2 Program Flow

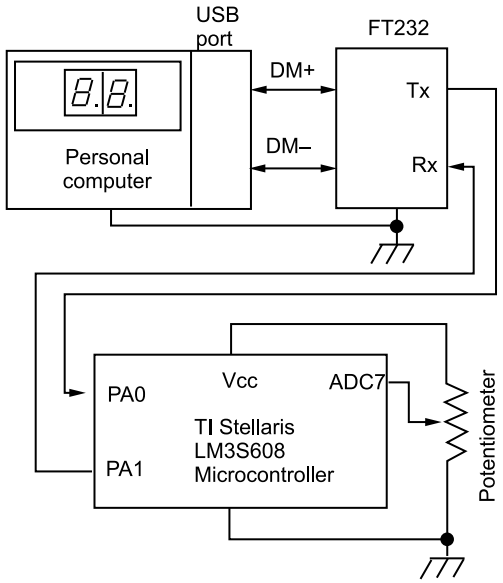

Figure 11.6 *Block diagram for Experiment 18*

The aim of this experiment is to sample the ambient temperature using LM35 sensor, scale it and send the value over UART0 for display on the serial monitor. This experiment also aims to evaluate the ADC module along with the UART module. Figure 11.6 shows to the block diagram of the setup required for this experiment. The algorithm to be followed for the program is:

- Enable the system clock, UART and the ADC module.
- Sample the temperature using ADC Channel 7.
- Scale the 10-bit value accordingly.
- Transmit the temperature value over the UART for display on Serial Monitor.

A suggested program flow is shown in Figure 11.7.

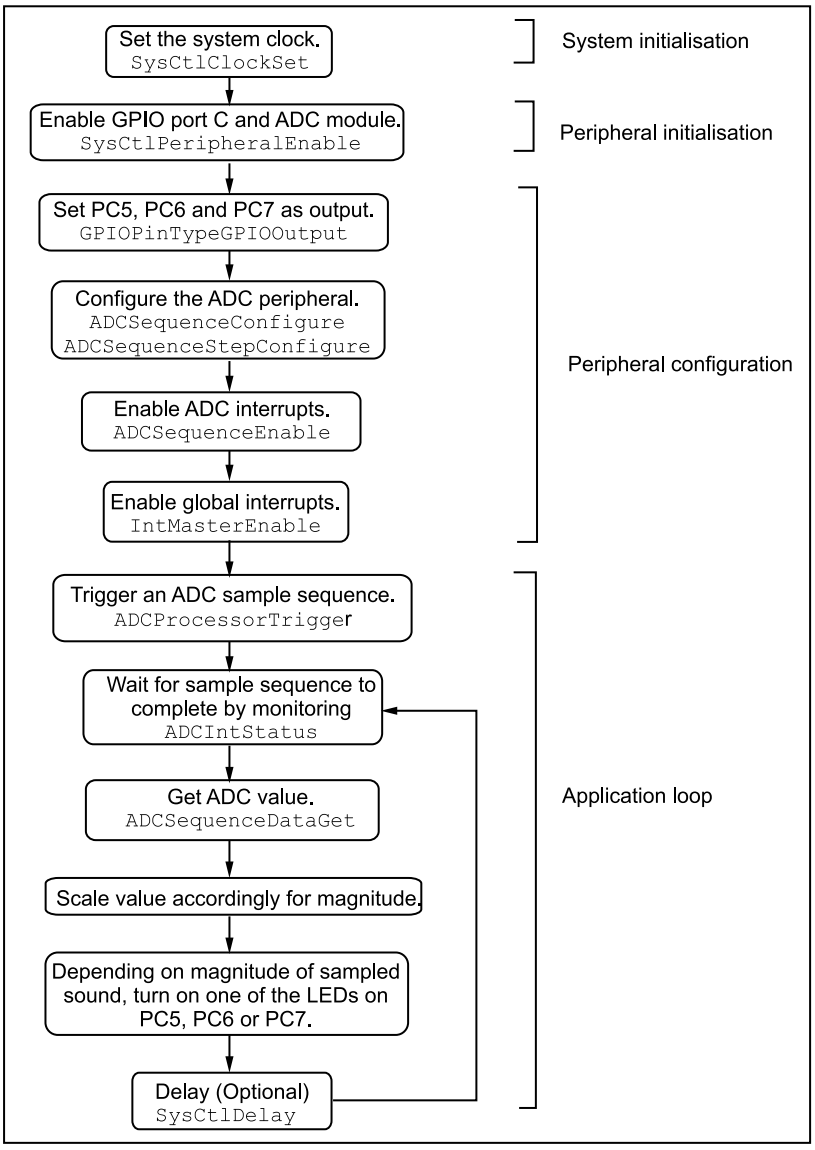

Figure 11.7 *Program flow for Experiment 18*

11.6 Experiment 19

11.6.1 Objective

Sample sound using a microphone connected to the ADC module and display the sound level on LEDs.

11.6.2 Program Flow

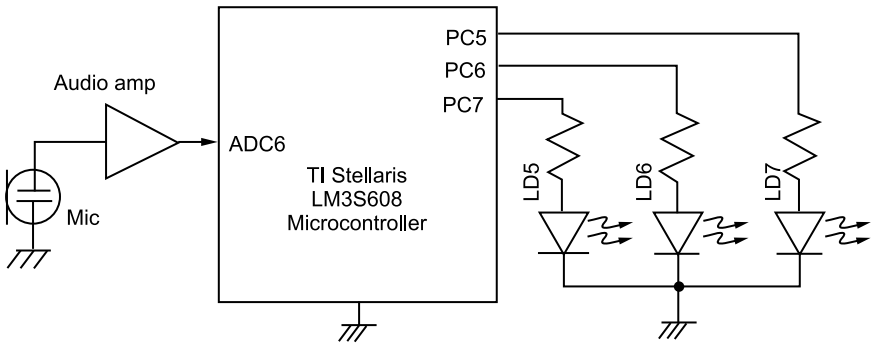

Figure 11.8 *Block diagram for Experiment 19*

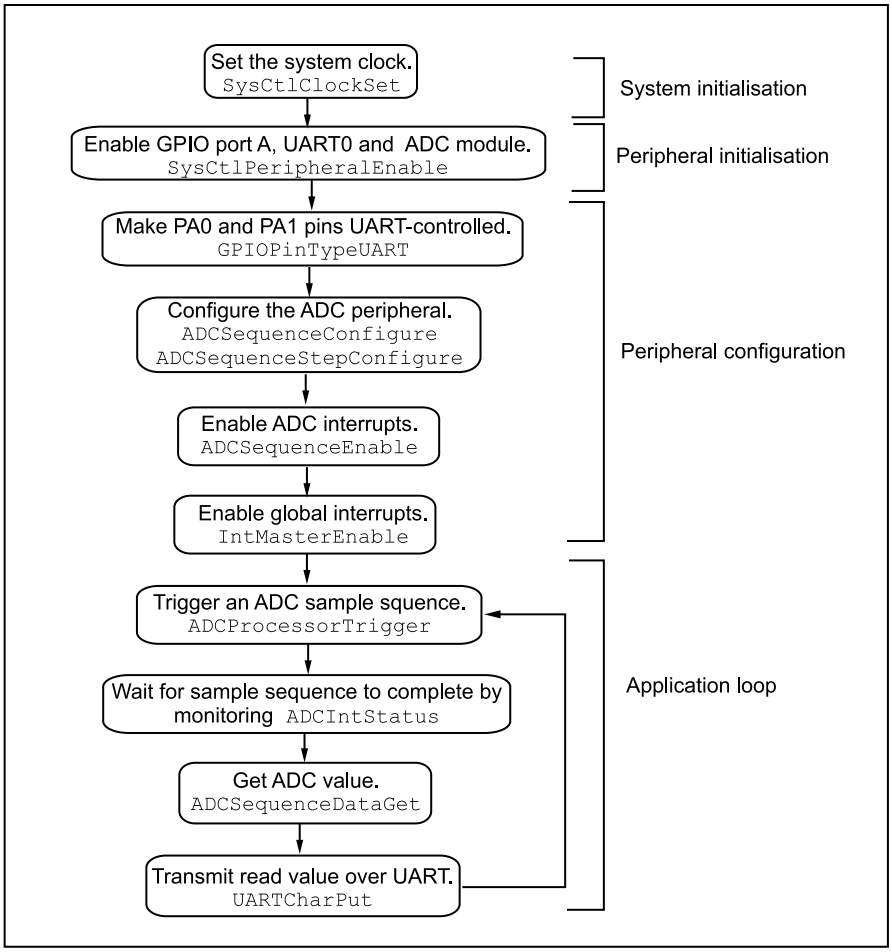

Figure 11.9 *Program flow for Experiment 19*

Let us play some music for this experiment!

This experiment requires you to sense audio and depict its magnitude using LEDs. We start by sensing audio using the onboard microphone and then, depending on its magnitude, turn on the corresponding LED. Figure 11.8 shows the block diagram of the setup required for this experiment. For example, if audio magnitude is high, turn on red; if it is low, turn on blue, and if it is somewhere in between, turn on green. The algorithm for the experiment is:

- Enable the system clock, the GPIO and the ADC module.
- Sample audio using the microphone connected to ADC Channel 6.
- Find the magnitude of the sample audio and turn on the corresponding LED.

The program flow for the experiment is shown in Figure 11.9.

11.7 Experiment 20

11.7.1 Objective

Sample sound using the onboard microphone and plot the amplitude vs. time waveform on a computer. Any graphing application available on the computer can be used for plotting.

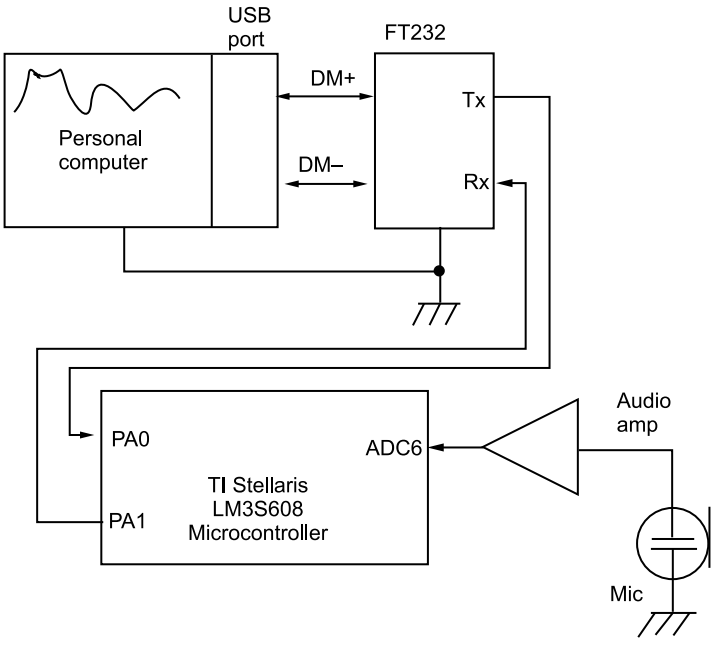

Figure 11.10 *Block diagram for Experiment 20*

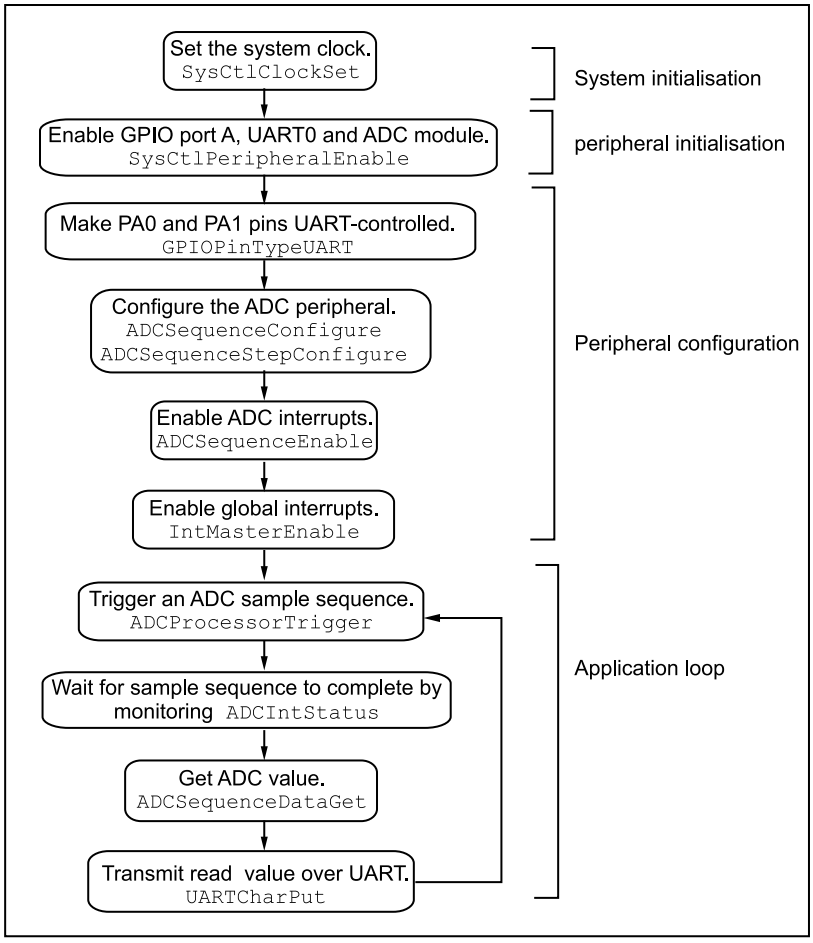

Figure 11.11 *Program flow for Experiment 20*

11.7.2 Program Flow

In this experiment we sample sound using a microphone and plot amplitude vs. time waveform for the sampled sound on a computer. Figure 11.10 shows the block diagram of the setup required for this experiment. We use the ADC module to sample the sound, scale it, and then send it over the UART to the computer. An application running on the computer then draws the waveform. The algorithm to be followed is:

1. Enable the system clock, the UART and the ADC module.
2. Sample sound using microphone connected to ADC Channel 6.
3. Scale the 10-bit analog signal.
4. Send the transformed value to the computer using UART.

There are many ways to plot waveforms on a computer. It is up to you what method you want to employ for this experiment. Some of the methods are:

1. Store incoming value to a text file and then plot using Microsoft Excel.
2. Develop a custom application for communicating with the serial port.

A suggested program flow for the experiment is shown in Figure 11.11.

11.8 Exercises

1. Vary the intensity of LD5 by moving the potentiometer.
2. Vary the intensity of LD5, LD6 and LD7 by moving the potentiometer. When the potentiometer is at its least resistance position, LD5 is dimly lit. When it is at the centre, LD5 is at its full brightness but LD6 is dimly lit, and when it is completely turned, all LEDs are lit at its brightest.
3. Filter the sound input using three filters of high, medium and low frequencies and show the output on LEDs, one for each filter.
4. Sample sound and analyse its spectrum using FFT, and plot The amplitude vs. frequency result on a computer.
5. Maintain time and sample the temperature values and output the current temperature with a timestamp over the UART.
6. Do a simple voice profiling for ON and OFF using a microphone by counting the number of zero crossings. If the user says ON, light up the LED, and in case of OFF, turn it off.
7. Measure audio input and vary the intensity of an LED depending on the magnitude of the voice sample.
8. Implement a two-player game where the person who shouts louder is the winner. Let LD5 denote the first player to be the winner and LD6 the second player.

12 Power Management and System Control

Power management and system control are very critical to the operation of Stellaris® Guru. This chapter deals with the various components involved with these functions.

12.1 System Control

System control determines the overall operation of the device. It provides information about the device, controls the clocking to the CPU and individual peripherals, and handles reset detection and reporting.

The system control module provides the following capabilities.

- Device identification
- Reset control, power control, clock control
- System control (modes of operation of the device)

12.1.1 Device Identification

The StellarisWare library provides a layer of abstraction and hides unnecessary detail from the end user. Various functions are pre-built in the library, which provide information present in several read-only registers about version, part number, SRAM size, Flash memory size, and other features. To display the Flash memory size and SRAM size reading over the UART on the terminal, use `SysCtlSRAMSizeGet`(void) and `SysCtlFlashSizeGet`(void) functions present in `/driverlib/sysctl.c`.

12.1.2 System Control

- Reset Control: The Stellaris® family of controllers has six sources of reset, namely:

1. **External reset button** The external reset pin resets the microcontroller including the CPU and all the on-chip peripherals. To improve noise immunity and/or to delay reset at power up, the reset input may be connected to an RC network.
2. **Power-on Reset** The internal power-on reset consists of a circuitry that generates a reset signal when the supply voltage reaches a threshold value.
3. **Internal brown-out reset** A drop in the supply voltage resulting in the assertion of the brown-out detector can be used to reset the controller.
4. **Software-initiated reset** Software reset can be used to reset a single peripheral or the entire system.
5. **Watchdog timer reset** The watchdog timer module's function is to prevent the system from hanging. Its job is to ensure the safe running of the code and it ensures that the system does not end up in an infinite loop leading to a hang-up.
6. **Internal low-drop out regulator** A reset can also be generated when the internal low drop-out (LDO) regulator output goes unregulated. The internal LDO integrated with the Stellaris® microcontroller is used to provide power to the majority of the controllers internal logic. The regulated value can be adjusted using software for power reduction.

- Clock Control: The Clock is the heart beat of any microcontroller system. There are multiple clock sources for use in the device:

1. **Internal Oscillator** The internal oscillator is an on-chip clock source. It does not require the use of external components. Its frequency is 12 MHz +/- 30%.
2. **Main Oscillator** The frequency-accurate clock rate is provided by the main oscillator. To use the PLL (phase locked loop) module with the main oscillator as the reference, it is imperative to use predefined external crystal of supported frequency.

The internal system clock (SysClk) is derived from either of these two sources or from the output of the main internal PLL, or the internal oscillator divided by 4. The run-mode clock configuration register allows control for the clocks in the run-time mode. It controls the clock source of the sleep and deep-sleep modes. The PLL sources, clock divisors and the input crystal selection are all controlled by the run-mode clock configuration register. The function SysCtlClockSet(unsigned long ulConfig) configures the clocking of the device. The function `SysCtlMOSCConfigSet`(unsigned long `ulConfig`) sets the configuration of the main oscillator control.

12.1.3 Modes of Operation

There are three levels of operation in the Stellaris® Guru.

- **Run Mode** This is the normal mode of execution of the program. The microcontroller executes code as normal and controls all the peripherals that are currently enabled. The system clock can be any of the available sources, including the PLL.
- **Sleep Mode** In the sleep mode the active peripherals remain clocked but the memory and the processor are not clocked, and hence do not execute code. A configured interrupt event can bring the device back from sleep mode to run mode.
- **Deep-Sleep Mode** In deep-sleep mode the clock frequency of the active peripherals may change (depending on the run-mode clock configuration) in addition to the processor clock being stopped. An interrupt returns the device to run mode from one of the sleep modes. Any properly configured interrupt event in the system will bring the processor back into the run mode.

12.2 Experiment 21

12.2.1 Objective

To evaluate the various sleep modes of the ARM® by putting the core in sleep and deep-sleep modes.

12.2.2 Program Flow

This experiment explains how to put the processor in sleep and deep-sleep modes. A button is pressed to generate an interrupt that puts the processor in sleep or deep-sleep modes. The same button is used to generate an interrupt to put the processor back in run mode. The algorithm to be followed is:

1. Enable the system clock and GPIO port E module.
2. Set internal pull ups on PE0 and enable interrupts for it.
3. When the pin is asserted, i.e., an interrupt is generated in the ISR, disable clock gating to GPIO Port E and enable the port for operation in sleep/deep-sleep modes. Put the processor to sleep/deep sleep.
4. When the pin is asserted again, reset the system to put the processor back into run mode.

In Figure 12.1, a suggested program flow for the experiment is given. Note that this program flow is drawn for putting the processor to sleep mode, although the same can be ported for putting the processor to deep-sleep mode.

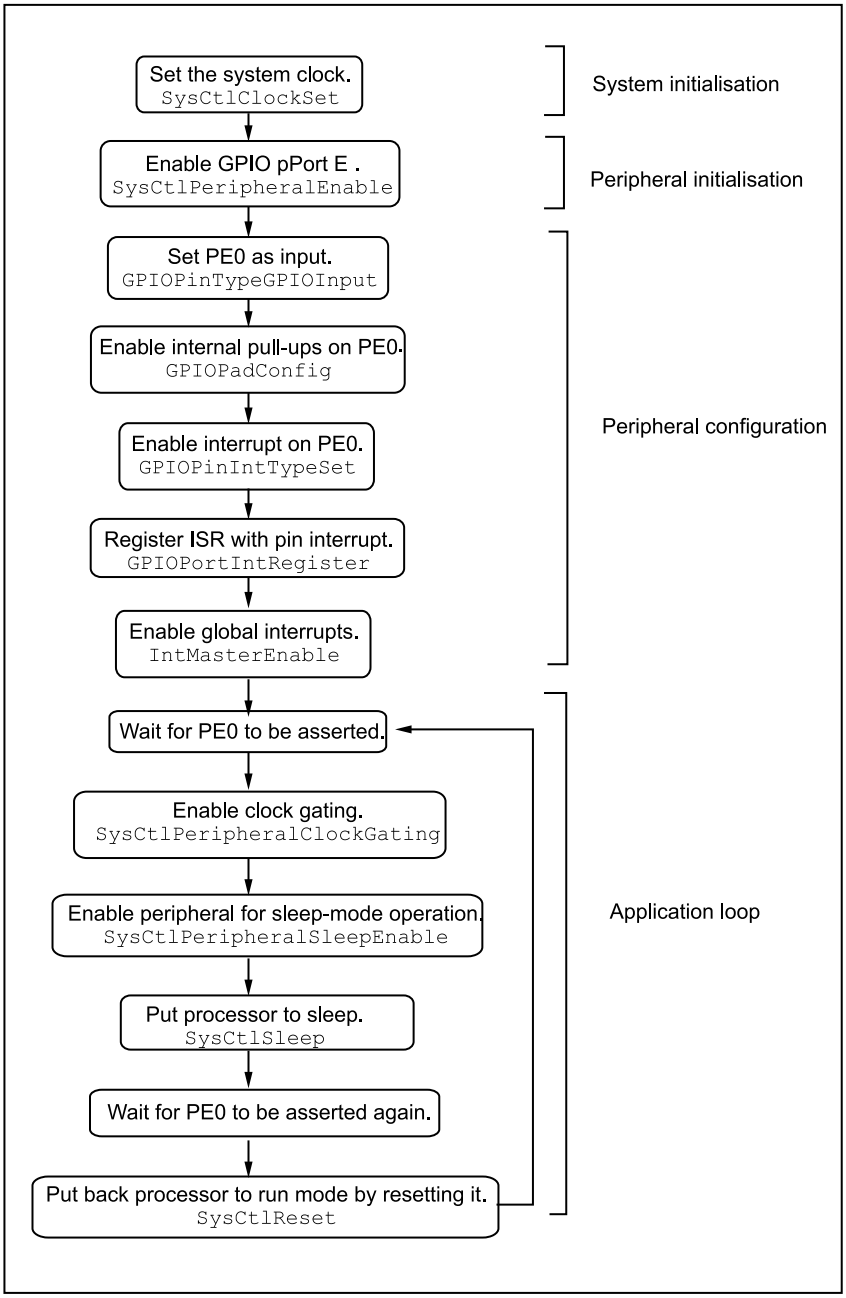

Figure 12.1 *Program flow for Experiment 21*

12.2.3 Suggested StellarisWare API Function Calls

- **SysCtlPeripheralClockGating** Controls the peripheral clock gating in sleep and deep-sleep modes.
  Prototype: `void SysCtlPeripheralClockGating`(tboolean 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 how peripherals are clocked when the processor goes into sleep- or deep-sleep modes. By default, the peripherals are clocked the same as in the run mode; if peripherals clock gating is enabled, they are clocked according to the configuration set by `SysCtlPeripheralSleepEnable()`, `SysCtlPeripheralSleepDisable()`, `SysCtlPeripheralDeepSleepEnable()`, and `SysCtlPeripheralDeepSleepDisable()`, which are explained further in this section.

  Returns: None

- **SysCtlPeripheralSleepEnable** Enable a peripheral in sleep mode.

  Prototype: `void SysCtlPeripheralSleepEnable`(unsigned long `ulPeripheral`)

  Parameters: `ulPeripheral` is the peripheral to be enabled in sleep mode.

  Description: This function allows a peripheral to be operating when the processor goes into sleep mode. The clock gating to a peripheral in sleep mode must be enabled before using this function.

  The `ulPeripheral` parameter can be any of the following macros,

```
SYSCTL_PERIPH_GPIOA,  SYSCTL_PERIPH_GPIOC,
SYSCTL_PERIPH_ADC0,   SYSCTL_PERIPH_UART0 etc.
```

  Returns: None

- **SysCtlSleep** Puts the processor into sleep mode.

  Prototype:`void SysCtlSleep`(void)

  Parameters: None

  Description: This function places the processor into sleep; it will not return until the processor returns to the run mode. The peripherals that are enabled using `SysCtlPeripheralSleepEnable` continue to function and can wake up the processor.

  Returns: None

- **SysCtlPeripheralDeepSleepEnable** Enables a peripheral in deep-sleep mode.

  Prototype: `void SysCtlPeripheralDeepSleepEnable`(unsigned long `ulPeripheral`)

  Parameters: `ulPeripheral` is the peripheral to be enabled in deep-sleep mode.

  Description: This function allows a peripheral to continue operating when the processor goes into deep-sleep mode. Since the clocking configuration of the device may change, not all peripherals can safely continue operating while the processor is in sleep mode. Deep-sleep-mode clocking of peripherals must be enabled using `SysCtlPeripheralClockGating`.

  The `ulPeripheral` parameter can be any of the following macros.

  ```
  SYSCTL_PERIPH_GPIOA,        SYSCTL_PERIPH_GPIOC,
  SYSCTL_PERIPH_ADC0,         SYSCTL_PERIPH_UART0 etc.
  ```

  Returns: None

- **SysCtlDeepSleep** Puts the processor in deep-sleep mode.

  Prototype: `void SysCtlDeepSleep`(void)

  Parameters: None

  Description: This function places the processor into deep-sleep mode; it will not return until the processor returns to the run mode. The peripherals that are enabled via `SysCtlPeripheralDeepSleepEnable()` continue to operate and can wake up the processor.

  Returns: None

- **SysCtlReset** Resets the device.

  Prototype: `void SysCtlReset`(void)

  Parameters: None

  Description: This function will perform a software reset of the entire device. The processor and all peripherals will be reset and all device registers will return to their default values.

  Returns: None

12.3 Experiment 22

12.3.1 Objective

Real-time alteration of system clock using the PLL module.

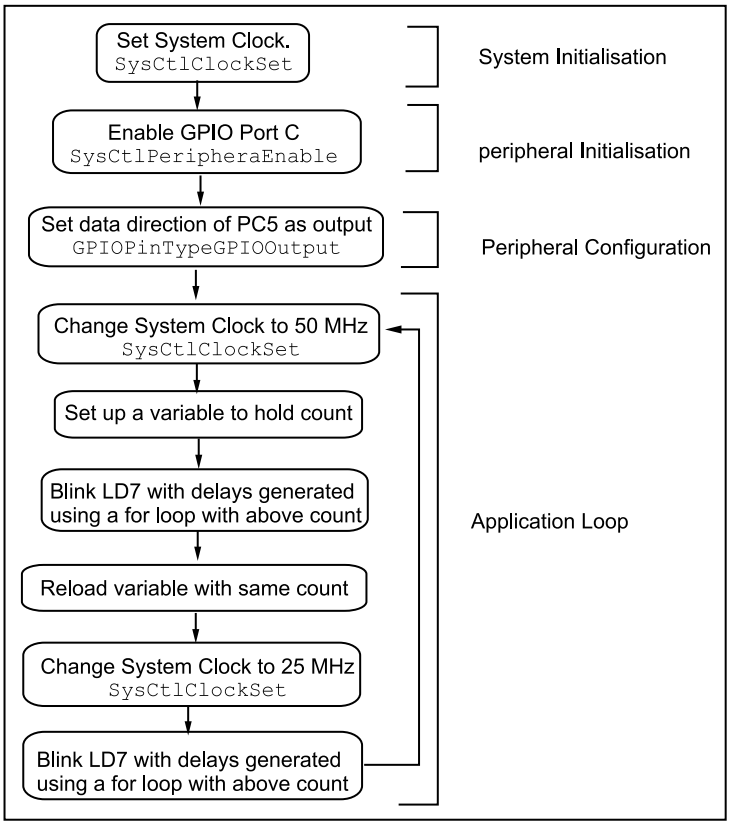

Figure 12.2 *Program flow for Experiment 22*

12.3.2 Program Flow

In all the experiments performed so far, setting up the system clock has been the fundamental step in every program. You must have noticed that we set up the system clock initially and then the rest of the program code followed. Can you think of any reason why `SysCtlClockSet` goes before other code? Is it possible to set up the system clock anywhere else in your program code? Can one alter the system clock later in application code? This experiment answers all of these questions.

The TI Stellaris® ARM® microcontroller uses phase locked loops (PLL) to generate several frequencies for use as system clock from a fixed-frequency external crystal. The answer to why `SysCtlClockSet` goes first is rather trivial. We must tell the processor as to which clock source we intend to use before it can start decoding instructions.

To answer the second question, think of a scenario when we start executing instructions without defining our clock source. Naturally, the processor will function at an indeterminate frequency. However, this is not the case. The architecture is designed to use a default clock

source if `SysCtlClockSet` is undefined. For the ARM®, the default is the external crystal of fixed frequency. So, whenever we call `SysCtlClockSet`, we are reconfiguring the processor to use the PLL module as the default source. Hence, it is possible to set up the system clock anywhere in the program code, even in the application loop section, as this way, we are just reconfiguring our clock settings and not tampering with the operation of the processor.

The answer to the third question is evident now. If we can set the system clock anywhere in our program code, we can do it so multiple times and vary our clock frequency. Indeed, this is the solution we employ when we want to put the processor in a low power state, as power is proportional to the square of the frequency. This experiment demonstrates how we can alter the system clock in real time in our program code. We know that the speed of execution of a for loop to generate delays is dependent on the system clock frequency. If we increase the system frequency, the for loop executes faster and vice versa. In the experiment, we alter the system clock frequency and see its effect on the blinking rate of an LED which uses a for loop for generating delays. The algorithm for the experiment is:

1. Enable the system Clock for a frequency of 50 MHz.
2. Enable GPIO port C and set the data direction of PC5 to output.
3. Set up a counter to hold an integer value.
4. Blink the LED, LD7(PC5), with delays generated using a `for` loop for decrementing the counter.
5. Make a call to `SysCtlClockSet` and change the system frequency to 25 MHz.
6. With the same count in the counter as before, make the LED blink with delays generated using a for loop.
7. Vary the system frequency and notice the effect on the blinking rate.

A suggested program flow is shown in Figure 12.2.

12.4 Exercises

1. Maintain an alarm using the timer module. The alarm and the current time setting are received over the UART. The processor is put into sleep mode with the timer running, and generating an interrupt every second which wakes up the processor. After incrementing the current time it goes back into sleep mode. When the current time equals the alarm time, LD5 is lit up and the processor goes back to sleep mode.
2. Evaluate the different reset modes and display the latest reset source using LEDs. In case of power-on-reset, LD5 is lit up. When reset is done using the Reset switch, LD6 is lit up. Switch S2 enables the watchdog timer and if reset is caused by the watchdog, LD7 is lit up. (Hint: use function `SysCtlResetCauseGet()`)

End Notes

1. Official Arduino website, http://www.arduino.cc
2. ARM® Cortex™-M3 Technical Reference Manual, Revision r2p1
3. Joint Test Action Group, http://www.jtag.com
4. Bray ++ Serial Terminal, https://sites.google.com/site/terminalbpp
5. Dhananjay V Gadre and Sheetal Vashist, LED senses and displays ambient-light intensity, EDN, 9 November 2006, www.edn.com/article/CA6387024
6. Shields are add-on boards mounted on top of the Arduino PCB for extending its capabilities.
   http: /arduino.cc/en/Main/Arduino_Shields
7. http://www.eclipse.org/
8. Sourcery CodeBench Lite Edition is free, unsupported version of Sourcery CodeBench provided by Mentor Graphics.
   http://www.mentor.com/embedded-software/sourcery-tools/sourcery -codebench/editions/ lite-edition/
9. This is an open source Eclipse CDT Managed Build Extensions for GNU ARM® toolchains.
   http://sourceforge.net/projects/gnuarmeclipse/
10. Future Technology Devices International, http://ftdichip.com
11. For an 8-bit port, if we want to set bits 2, 4, 6 as outputs and the rest as inputs, the bit-packed representation will be denoted by 0b01010100 or 0×54.
12. Keil Embedded Development Tools, http://www.keil.com
13. IAR Systems, http://www.iar.com
14. Code Red Technologies, http://www.code-red-tech.com
15. Code Composer Studio, http://www.ti.com/tool/ccstudio

Bibliography

If you wish to read more about the ARM® processor family and specifically, more about the Cortex™- M3, we recommend to you some good reading material below, in no specific order of preference. These books and articles have been used in the development of this manual also.

- Joseph Yiu, *The Definitive Guide to the ARM® Cortex™-M3*, Second Edition, Elsevier Inc., 2010.
- Steve Furber, *ARM® System-On-A-Chip Architecture*, Second Edition, Addison Wesley, 2000.
- Andrew N. Sloss, Dominic Symes and Chris Wright, *ARM® System Developers Guide: Designing and Optimizing System Software*, Elsevier Inc., 2004.
- William Hohl, *ARM® Assembly Language*, CRC Press, 2009.
- Jonathan M. Valvano, *Embedded System and Real Time Interfacing to the ARM® Cortex™-M3*, Create Space, 2011.
- StellarisWare Peripheral Driver Library User Guide, Build 8555, Texas Instruments.
- Wikipedia article on Linear Feedback Shift Register: `http://en.wikipedia.org/wiki/Linear_feedback_shift_register`

Index