MORGAN & CLAYPOOL PUBLISHERS

Steven F. Barrett, Daniel J. Pack

Microcontroller Programming and Interfacing with Texas Instruments MSP430FR2433 and MSP430FR5994

Second Edition

Synthesis Lectures on Digital Circuits and Systems

Mitchell A. Thornton, *Series Editor*

Microcontroller Programming and Interfacing with Texas Instruments MSP430FR2433 and MSP430FR5994

Second Edition

Synthesis Lectures on Digital Circuits and Systems

Editor
Mitchell A. Thornton, ***Southern Methodist University***

The *Synthesis Lectures on Digital Circuits and Systems* series is comprised of 50- to 100-page books targeted for audience members with a wide-ranging background. The Lectures include topics that are of interest to students, professionals, and researchers in the area of design and analysis of digital circuits and systems. Each Lecture is self-contained and focuses on the background information required to understand the subject matter and practical case studies that illustrate applications. The format of a Lecture is structured such that each will be devoted to a specific topic in digital circuits and systems rather than a larger overview of several topics such as that found in a comprehensive handbook. The Lectures cover both well-established areas as well as newly developed or emerging material in digital circuits and systems design and analysis.

Microcontroller Programming and Interfacing with Texas Instruments MSP430FR2433 and MSP430FR5994, Second Edition
Steven F. Barrett and Daniel J. Pack
2019

Synthesis of Quantum Circuits vs. Synthesis of Classical Reversible Circuits
Alexis De Vos, Stijn De Baerdemacker, and Yvan Van Rentergen
2018

Boolean Differential Calculus
Bernd Steinbach and Christian Posthoff
2017

Embedded Systems Design with Texas Instruments MSP432 32-bit Processor
Dung Dang, Daniel J. Pack, and Steven F. Barrett
2016

Fundamentals of Electronics: Book 4 Oscillators and Advanced Electronics Topics
Thomas F. Schubert and Ernest M. Kim
2016

Fundamentals of Electronics: Book 3 Active Filters and Amplifier Frequency
Thomas F. Schubert and Ernest M. Kim
2016

Bad to the Bone: Crafting Electronic Systems with BeagleBone and BeagleBone Black, Second Edition
Steven F. Barrett and Jason Kridner
2015

Fundamentals of Electronics: Book 2 Amplifiers: Analysis and Design
Thomas F. Schubert and Ernest M. Kim
2015

Fundamentals of Electronics: Book 1 Electronic Devices and Circuit Applications
Thomas F. Schubert and Ernest M. Kim
2015

Applications of Zero-Suppressed Decision Diagrams
Tsutomu Sasao and Jon T. Butler
2014

Modeling Digital Switching Circuits with Linear Algebra
Mitchell A. Thornton
2014

Arduino Microcontroller Processing for Everyone! Third Edition
Steven F. Barrett
2013

Boolean Differential Equations
Bernd Steinbach and Christian Posthoff
2013

Bad to the Bone: Crafting Electronic Systems with BeagleBone and BeagleBone Black
Steven F. Barrett and Jason Kridner
2013

Introduction to Noise-Resilient Computing
S.N. Yanushkevich, S. Kasai, G. Tangim, A.H. Tran, T. Mohamed, and V.P. Shmerko
2013

Atmel AVR Microcontroller Primer: Programming and Interfacing, Second Edition
Steven F. Barrett and Daniel J. Pack
2012

Representation of Multiple-Valued Logic Functions
Radomir S. Stankovic, Jaakko T. Astola, and Claudio Moraga
2012

Arduino Microcontroller: Processing for Everyone! Second Edition
Steven F. Barrett
2012

Advanced Circuit Simulation Using Multisim Workbench
David Báez-López, Félix E. Guerrero-Castro, and Ofelia Delfina Cervantes-Villagómez
2012

Circuit Analysis with Multisim
David Báez-López and Félix E. Guerrero-Castro
2011

Microcontroller Programming and Interfacing Texas Instruments MSP430, Part I
Steven F. Barrett and Daniel J. Pack
2011

Microcontroller Programming and Interfacing Texas Instruments MSP430, Part II
Steven F. Barrett and Daniel J. Pack
2011

Pragmatic Electrical Engineering: Systems and Instruments
William Eccles
2011

Pragmatic Electrical Engineering: Fundamentals
William Eccles
2011

Introduction to Embedded Systems: Using ANSI C and the Arduino Development Environment
David J. Russell
2010

Arduino Microcontroller: Processing for Everyone! Part II
Steven F. Barrett
2010

Arduino Microcontroller Processing for Everyone! Part I
Steven F. Barrett
2010

Digital System Verification: A Combined Formal Methods and Simulation Framework
Lun Li and Mitchell A. Thornton
2010

Progress in Applications of Boolean Functions
Tsutomu Sasao and Jon T. Butler
2009

Embedded Systems Design with the Atmel AVR Microcontroller: Part II
Steven F. Barrett
2009

Embedded Systems Design with the Atmel AVR Microcontroller: Part I
Steven F. Barrett
2009

Embedded Systems Interfacing for Engineers using the Freescale HCS08 Microcontroller II: Digital and Analog Hardware Interfacing
Douglas H. Summerville
2009

Designing Asynchronous Circuits using NULL Convention Logic (NCL)
Scott C. Smith and JiaDi
2009

Embedded Systems Interfacing for Engineers using the Freescale HCS08 Microcontroller I: Assembly Language Programming
Douglas H.Summerville
2009

Developing Embedded Software using DaVinci & OMAP Technology
B.I. (Raj) Pawate
2009

Mismatch and Noise in Modern IC Processes
Andrew Marshall
2009

Asynchronous Sequential Machine Design and Analysis: A Comprehensive Development of the Design and Analysis of Clock-Independent State Machines and Systems
Richard F. Tinder
2009

An Introduction to Logic Circuit Testing
Parag K. Lala
2008

Pragmatic Power
William J. Eccles
2008

Multiple Valued Logic: Concepts and Representations
D. Michael Miller and Mitchell A. Thornton
2007

Finite State Machine Datapath Design, Optimization, and Implementation
Justin Davis and Robert Reese
2007

Atmel AVR Microcontroller Primer: Programming and Interfacing
Steven F. Barrett and Daniel J. Pack
2007

Pragmatic Logic
William J. Eccles
2007

PSpice for Filters and Transmission Lines
Paul Tobin
2007

PSpice for Digital Signal Processing
Paul Tobin
2007

PSpice for Analog Communications Engineering
Paul Tobin
2007

PSpice for Digital Communications Engineering
Paul Tobin
2007

PSpice for Circuit Theory and Electronic Devices
Paul Tobin
2007

Pragmatic Circuits: DC and Time Domain
William J. Eccles
2006

Pragmatic Circuits: Frequency Domain
William J. Eccles
2006

Pragmatic Circuits: Signals and Filters
William J. Eccles
2006

High-Speed Digital System Design
Justin Davis
2006

Introduction to Logic Synthesis using Verilog HDL
Robert B.Reese and Mitchell A.Thornton
2006

Microcontrollers Fundamentals for Engineers and Scientists
Steven F. Barrett and Daniel J. Pack
2006

Microcontroller Programming and Interfacing with Texas Instruments MSP430FR2433 and MSP430FR5994, Second Edition

Steven F. Barrett and Daniel J. Pack

www.morganclaypool.com

ISBN: 9781681736242 paperback
ISBN: 9781681736273 hardcover

A Publication in the Morgan & Claypool Publishers series
SYNTHESIS LECTURES ON DIGITAL CIRCUITS AND SYSTEMS

Lecture #54
Series Editor: Mitchell A. Thornton, *Southern Methodist University*
Series ISSN
Print 1932-3166 Electronic 1932-3174

Microcontroller Programming and Interfacing with Texas Instruments MSP430FR2433 and MSP430FR5994

Second Edition

Steven F. Barrett
University of Wyoming, Laramie, WY

Daniel J. Pack
University of Tennessee Chattanooga, TN

SYNTHESIS LECTURES ON DIGITAL CIRCUITS AND SYSTEMS #54

MORGAN & CLAYPOOL PUBLISHERS

ABSTRACT

This book provides a thorough introduction to the Texas Instruments MSP430™ microcontroller. The MSP430 is a 16-bit reduced instruction set (RISC) processor that features ultra-low power consumption and integrated digital and analog hardware. Variants of the MSP430 microcontroller have been in production since 1993. This provides for a host of MSP430 products including evaluation boards, compilers, software examples, and documentation. A thorough introduction to the MSP430 line of microcontrollers, programming techniques, and interface concepts are provided along with considerable tutorial information with many illustrated examples. Each chapter provides laboratory exercises to apply what has been presented in the chapter. The book is intended for an upper level undergraduate course in microcontrollers or mechatronics but may also be used as a reference for capstone design projects. Also, practicing engineers already familiar with another microcontroller, who require a quick tutorial on the microcontroller, will find this book very useful. This second edition introduces the MSP–EXP430FR5994 and the MSP430–EXP430FR2433 LaunchPads. Both LaunchPads are equipped with a variety of peripherals and Ferroelectric Random Access Memory (FRAM). FRAM is a nonvolatile, low-power memory with functionality similar to flash memory.

KEYWORDS

MSP430 microcontroller, microcontroller interfacing, embedded systems design, Texas Instruments

To our families

Contents

Preface

Texas Instruments is well known for its analog and digital devices, in particular, Digital Signal Processing (DSP) chips. Unknown to many, the company quietly developed its microcontroller division in the early 1990s and started producing a family of controllers aimed mainly for embedded meter applications, which require an extended operating time without intervention for power companies. It was not until the mid 2000s that the company began serious effort to present the MSP430 microcontroller family, its flagship microcontroller, to the academic community and future engineers. Their efforts have been quietly attracting many educators and students due to the MSP430's cost and the suitability of the controller for capstone design projects requiring microcontrollers. In addition, Texas Instruments offers many compatible analog and digital devices that can expand the range of the possible embedded applications of the microcontroller. Texas Instruments has continually added new innovation to the MSP430 microcontroller line. The second edition introduces the MSP–EXP430FR5994 and the MSP–EXP430FR2433 LaunchPads. Both LaunchPads are equipped with a variety of peripherals and Ferroelectric Random Access Memory (FRAM). FRAM is a nonvolatile, low-power memory with functionality similar to flash memory.

This book is about the MSP430 microcontroller family. We have three goals in writing this book. The first is to introduce readers to microcontroller programming. The MSP430 microcontrollers can be programmed either using assembly language or a high–level programming language such as C. The second goal of the book is to teach students how computers work. After all, a microcontroller is a computer within a single integrated circuit (chip). Finally, we present the microcontroller's input/output interface capabilities, one of the main reasons for developing embedded systems with microcontrollers.

Background

This book provides a thorough introduction to the Texas Instruments MSP430 microcontroller. The MSP430 is a 16-bit reduced instruction set (RISC) processor that features ultra-low power consumption and integrated digital and analog hardware. Variants of the MSP430 microcontroller have been in production since 1993 with a host of MSP430-related products including evaluation boards, compilers, software examples, and documentation.

This book is intentionally tutorial in nature with many worked examples, illustrations, and laboratory exercises. An emphasis is placed on real-world applications such as smart home concepts, mobile robots, an unmanned underwater vehicle, and a DC motor controller to name a few.

Intended Readers

The book is intended for an upper level undergraduate course in microcontrollers or mechatronics but may also be used as a reference for capstone design projects. Also, practicing engineers who are already familiar with another line of microcontrollers, but require a quick tutorial on the MSP430 microcontroller, will find this book beneficial.

Approach and Organization

This book provides a thorough introduction to the MSP430 line of microcontrollers, programming techniques, and interface concepts. Each chapter contains a list of objectives, background tutorial information, and detailed information on the operation of the MSP430 system under study. Furthermore, each chapter provides laboratory exercises to apply what has been presented in the chapter and how the concepts are employed in real applications. Each chapter concludes with a series of homework exercises divided into Fundamental, Advanced, and Challenging categories. The reader will get the most out of the book by also having the following references readily available:

- MSP430FR2433 Mixed–Signal Microcontroller, SLASE59B;
- MSP430FR4xx and MSP430FR2xx Family User's Guide, SLAU445G;
- MSP430FR599x, MSP430FR596x Mixed–Signal Microcontrollers, SLASE54B; and
- MSP430FR58xx, MSP430FR59xx, and MSP430FR6xx Family User's Guide, SLAU367O.

This documentation is available for download from the Texas Instruments website [www.ti.com].

Chapter 1 provides a brief review of microcontroller terminology and a short history followed by an overview of the MSP430 microcontroller. The overview surveys systems onboard the microcontroller and also various MSP430 families. The chapter concludes with an introduction to the hardware and software development tools that will be used for the remainder of the book. Our examples employ the MSP–EXP430FR5994 and the MSP430FR2433 LaunchPads, the Energia rapid prototyping platform, and the Texas Instruments' Code Composer Studio Integrated Development Environment (IDE). The information provided can be readily adapted to other MSP430 based experimenter boards.

Chapter 2 provides a brief introduction to programming in C. The chapter contains multiple examples for a new programmer. It also serves as a good review for seasoned programmers. Also, software programming tools including Energia, Code Composer Studio IDE, and debugging tools are explored. This chapter was adapted from material originally written for the Texas Instruments MSP432, a 32-bit processor that has close ties to the 16-bit MSP430.[1] Embed-

[1]This chapter was adapted with permission from *Arduino Microcontroller Processing for Everyone*, S. Barrett, 3rd ed., Morgan & Claypool Publishers, San Rafael, CA, 2013.

ded system developers will find a seamless transition between the MSP430 and MSP432 line of processors.

Chapter 3 introduces the MSP430 hardware architecture, software organization, and programming model. The chapter also presents an introduction to the MSP430 orthogonal instruction set, including its 27 instructions and 9 emulated instructions.

Chapter 4 describes a wide variety of input and output devices and how to properly interface them to the MSP430 microcontroller. We believe it is essential for the embedded system designer to understand the electrical characteristics of the processor so a proper interface to peripheral components may be designed. We have included a chapter on these concepts for the books we have written for the Synthesis Lecture Series. We continue to add material as new microcontroller peripherals are developed. The chapter begins with a review of the MSP430 electrical operating parameters followed by a discussion of the port system. The chapter includes a description of a wide variety of input device concepts including switches, interfacing, debouncing, and sensors. Output device concepts are then discussed including light-emitting diodes (LEDs), tri–state LED indicators, liquid crystal displays (LCDs), high-power DC and AC devices, motors, and annunciator devices.

Chapter 5 provides an in–depth discussion of the MSP430 power management system. The power management system provides for ultra-low power operation and practices.

Chapter 6 is dedicated to the different memory components onboard the MSP430 including the new FRAM nonvolatile memory, RAM, EEPROM and the associated memory controllers. The Direct Memory Access (DMA) controller is also discussed.

Chapter 7 discusses the clock and timer systems aboard the MSP430. The chapter begins with a detailed discussion of the flexible clock system, followed by a discussion of the timer system architecture. The timer architecture discussion includes the Watchdog timers, timers A and B, real-time clocks, and pulse width modulation (PWM).

Chapter 8 provides an introduction to the concepts of resets and interrupts. The various interrupt systems associated with the MSP430 are discussed, followed by detailed instructions on how to properly configure and program them.

Chapter 9 discusses the analog systems aboard the MSP430. The chapter discusses the analog–to–digital converters (ADCs), the digital–to–analog converters (DACs), and the comparators.

Chapter 10 is designed for a detailed review of the complement of serial communication systems resident onboard the MSP430, including the universal asynchronous receiver transmitter (UART), the serial peripheral interface (SPI), the I2C system, the radio frequency (RF) link, USB, and the IrDA infrared link. The systems are contained within the MSP430 universal serial communication interfaces eUSCI_A and eUSCI_B subsystems.

Chapter 11 provides a detailed introduction to the data integrity features aboard the MSP430 including a discussion of noise and its sources and suppression, an Advanced Encryption Standard (AES) 256 accelerator module, and a 16- or 32-bit cyclic redundancy check

(CRC) engine. This chapter was adapted from material originally written for the Texas Instruments MSP432, a 32-bit processor that has close ties to the 16-bit MSP430.[2] Embedded system developers will find a seamless transition between the MSP430 and MSP432 line of processors.

Chapter 12 discusses the system design process followed by system level examples. We view the microcontroller as the key component within the larger host system. It is essential the embedded system designer has development, design, and project management skills to successfully complete a project. This chapter provides an introduction some of the skills used for project development. We have included a chapter on these concepts for the books we have written for the Synthesis Lecture Series. The examples have been carefully chosen to employ a wide variety of MSP430 systems discussed throughout the book.

Table 1 provides a summary of chapter contents and related MSP430 subsystems.

Steven F. Barrett and Daniel J. Pack
July 2019

[2] *Embedded Systems Design with the Texas Instruments MSP432 32-bit Processor*, Dung Dang, Daniel J. Pack, and Steven F. Barrett, Morgan & Claypool Publishers, San Rafael, CA, 2017.

Table 1: MSP-EXP430FR5994 and the MSP-EXP430FR2433 LaunchPad subsystems.

Chapter	MSP- EXP430FR2433	MSP-EXP430FR5994
Ch. 1: Introduction		
Ch. 2: Programming	MSP430 port system	MSP430 port system
Ch. 3: HW and SW	Joint Test Action Group (JTAG) serial debug port, Enhanced Emulation Module (EEM) onboard debug tool, serial Spy-Bi-Wire (SBY) JTAG	Joint Test Action Group (JTAG) serial debug port, Enhanced Emulation Module (EEM) on-board debug tool, serial Spy-Bi-Wire (SBY) JTAG
Ch. 4: Interfacing	MSP430 port system	MSP430 port system
Ch. 5: Power Mgt	Power Mgt Module	Power Mgt: LDO, SVS, Brownout
Ch. 6: Memory	FRAM: 15KB + 512B RAM: 4KB	FRAM: 256KB RAM: 4 KB + 4 KB DMA Controller Memory Protection Unit (MPU) IP Encapsulation Segment (IPE)
Ch. 7: Timer Systems - Clock - Timers	Clock system (CS), LFXT Timer_A3(2), Timer_A2(2) Watchdog, Real-Time Clock	Clock system (CS), TB0: Timer_B, TA0: Timer_A, TA1: Timer_A, TA4: Timer_A, Watchdog, Real-Time Clock
Ch. 8: Resets and Interrupts		
Ch. 9: Analog Peripherals	ADC: 8 ch, SE, 10-bit, 200 ksps	Comp_E: 16 ch, Ref_A ADC 12_B: 16 ch SE/8 DE, 12-bit
Ch. 10: Comm Sys	eUSCI_A(2) - UART, IrDA, SPI eUSCI_B0 - SPI, I2C	eUSCI_A(4) (A0 to A3) - UART, IrDA, SPI eUSCI_B(4) (B0 to B3) - SPI, I2C
Ch. 11: System Integrity	CRC16: 16-bit cyclic redundancy check	CRC16: CRC-16-CCITT CRC32: CRC-32-ISO-3309 AES 256: security encryption/decryption
Ch. 12: System Design		

Acknowledgments

There have been many people involved in the conception and production of this book. We especially want to thank Doug Phillips, Mark Easley, and Franklin Cooper of Texas Instruments. The future of Texas Instruments is bright with such helpful, dedicated engineering and staff members. In 2005, Joel Claypool of Morgan & Claypool Publishers, invited us to write a book on microcontrollers for his new series titled "Synthesis Lectures on Digital Circuits and Systems." The result was the book *Microcontrollers Fundamentals for Engineers and Scientists.* Since then we have been regular contributors to the series. Our goal has been to provide the fundamental concepts common to all microcontrollers and then apply the concepts to the specific microcontroller under discussion. We believe that once you have mastered these fundamental concepts, they are easily transportable to different processors. As with many other projects, he has provided his publishing expertise to convert our final draft into a finished product. We thank him for his support on this project and many others. He has provided many novice writers the opportunity to become published authors. His vision and expertise in the publishing world made this book possible. We thank Sara Kreisman of Rambling Rose Press, Inc. for her editorial expertise. We also thank Dr. C.L. Tondo of T&T TechWorks, Inc. and his staff for working their magic to convert our final draft into a beautiful book. Finally, we thank our families who have provided their ongoing support and understanding while we worked on books over the past fifteen plus years.

Steven F. Barrett and Daniel J. Pack
July 2019

CHAPTER 1

Introduction to Microcontroller Technology

Objectives: After reading this chapter, the reader should be able to:

- describe the key technological accomplishments leading to the development of the microcontroller;
- define microprocessor, microcontroller, and microcomputer;
- identify examples of microcontroller applications in daily life;
- list key attributes of the MSP430 microcontroller;
- describe different features that differentiate MSP430 microcontroller family members;
- list the subsystems onboard the MSP430FR2433 and the MSP430FR5994 microcontrollers;
- provide an example application for each subsystem onboard the MSP430 microcontrollers;
- describe the hardware, software, and emulation tools available for the MSP430 microcontrollers; and
- employ the development tools to load and execute simple programs on the MSP-EXP430FR2433 and the MSP-EXP430FR5994 evaluation boards.

In every chapter, we start with a motivation and background followed by a section on theory. After the theory section, an example application is used to demonstrate the operational use of chapter concepts. Each chapter includes a hands-on laboratory exercise and a list of chapter references, which you can use to explore further areas of interest. Each chapter concludes with a series of practice exercises, divided into Fundamental, Advanced, and Challenging levels.

1.1 MOTIVATION

This book is about microcontrollers! A microcontroller is a self-contained processor system in a single integrated circuit (IC or chip) that contains essential functional units of a general-purpose computer such as a central processing unit (CPU), a memory, and input/output (I/O) units.

Microcontrollers provide local computational resources to many products, requiring a limited amount of processing power to perform their functions. They are everywhere! In the routine of daily life, we use multiple microcontrollers. Take a few minutes and jot down a list of microcontroller equipped products, sometimes called embedded systems, you have used today.

This chapter introduces the Texas Instruments MSP430 line of microcontrollers. We begin with a brief history of computer technology followed by an introduction to the MSP430FR2433 and the MSP430FR5994 microcontrollers. After a review of these MSP430 microcontrollers, we introduce you to the powerful and user-friendly development tools.

1.2 BACKGROUND THEORY: A BRIEF HISTORY AND TERMINOLOGY

The development of microcontrollers can be traced back to the time of early computing with the first generation of computers. The generations of computer development are marked by breakthroughs in hardware and architecture innovation. The first generation of computers employed vacuum tubes as the main switching element. Mauchly and Eckert developed the electronic numerical integrator and calculator (ENIAC) in the mid 1940s. This computer was large and consumed considerable power due to its use of 18,000 vacuum tubes. The computer, funded by the U.S. Army, was employed to calculate ordnance trajectories in World War II. The first commercially available computer of this era was the UNIVAC I [Bartee, 1972].

The second generation of computers employed transistors as the main switching element. The transistor was developed in 1947 by John Bardeen and Walter Brattain at Bell Telephone Laboratories. Bardeen, Brattain, and William Schockley were awarded the 1956 Nobel Prize in Physics for development of the transistor [Nobel.org]. The transistor reduced the cost, size, and power consumption of computers.

The third generation of processors started with the development of the integrated circuit. The integrated circuit was developed by Jack Kilby at Texas Instruments in 1958. The integrated circuit revolutionized the production of computers, greatly reducing their size and power consumption. Computers employing integrated circuits were first launched in 1965 [Bartee, 1972], and Kilby was awarded the Nobel Prize in Physics in 2002 for his work on the integrated circuit [Nobel.org]. The first commercially available minicomputer of this generation was the digital equipment corporation's (DEC) PDP-8 [Osborne, 1980].

The fourth generation of computers was marked by the advancement of levels of integration, leading to very large-scale integration (VLSI) and ultra-large scale integration (ULSI) production techniques. In 1969, the Data Point Corporation of San Antonio, Texas had designed an elementary CPU. The CPU provides the arithmetic and control for a computer. Data Point contracted with Intel and Texas Instruments to place the design on a single integrated circuit. Intel was able to complete the task, but Data Point rejected the processor as being too slow for their intended application [Osborne, 1980].

Intel used the project as the basis for their first general-purpose 8-bit microprocessor, the Intel 8008. The microprocessor chip housed the arithmetic and control unit for the computer. Other related components such as memory (ROM), random access memory (RAM), I/O components, and interface hardware were contained in external chips. From 1971–1977, Intel released the 8008, 8080, and 8085 microprocessors which significantly reduced the number of system components and improved upon the number of power supply voltages required for the chips. Some of the high visibility products of this generation were the Apple II personal computer, developed by Steve Jobs and Steve Wozniak and released in 1977, and the IBM personal computer, released in 1981 [MCS 85, 1977, Osborne, 1980].

The first single-chip microcontroller was developed by Gary Boone of Texas Instruments in the early 1970s. A microcontroller contains all key elements of a computer system within a single integrated circuit. Boone's first microcontroller, the TMS 1000, contained the CPU, ROM, RAM, and I/O and featured a 400 kHz clock [Boone, 1973, 1978]. From this early launch of microcontrollers, an entire industry was launched. There are now over 35 plus companies manufacturing microcontrollers worldwide offering over 250 different product lines [Wendt]. The MSP430 line of microcontrollers was first developed in 1992 and became available for worldwide release in 1997.

1.3 MICROCONTROLLER SYSTEMS

Although today's microcontrollers physically bear no resemblance to their earlier computer predecessors, they all have a similar architecture. All computers share the basic systems shown in Figure 1.1. The processor or CPU contains both datapath and control hardware. The datapath is often referred to as the arithmetic logic unit (ALU). As its name implies, the ALU provides hardware to perform the mathematical and logic operations for the computer. The control unit provides an interface between the computer's hardware and software. It generates control signals to the datapath and other system components such that operations occur in the correct order and within an appropriate time to execute the desired actions of a software program.

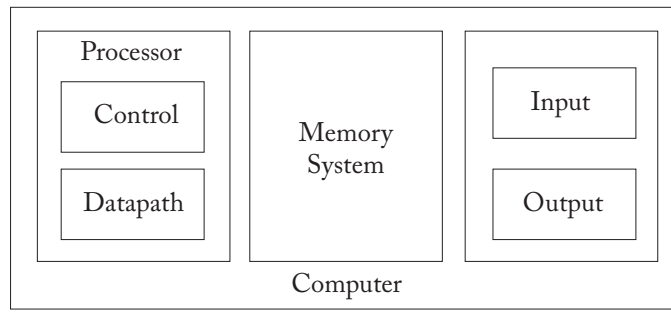


Figure 1.1: Basic computer architecture. (Adapted from Patterson and Hennessy [1994].)

The memory system contains a variety of memory components to support the operation of the computer. Typical memory systems aboard microcontrollers contain RAM, ROM, and electrically erasable programmable read only memory (EEPROM) components. RAM is volatile. When power is unavailable, the contents of RAM memory is lost. RAM is typically used in microcontroller operations for storing global variables, local variables, which are required during execution of a function, and to support heap operations during dynamic allocation activities. In contrast, ROM memory is nonvolatile. That is, it retains its contents even when power is not available. ROM memory is used to store system constants and parameters. If a microcontroller application is going to be mass produced, the resident application program may also be written into ROM memory at the manufacturer.

EEPROM is available in two variants: byte-addressable and flash programmable. Byte-addressable memory EEPROM, as its name implies, allows variables to be stored, read, and written during program execution. The access time for byte-addressable EEPROM is much slower than RAM memory; however, when power is lost, the EEPROM memory retains its contents. Byte-addressable EEPROM may be used to store system passwords and constants. For example, if a microcontroller-based algorithm has been developed to control the operation of a wide range of industrial doors, system constants for a specific door type can be programmed into the microcontroller onsite when the door is installed. Flash EEPROM can be erased or programmed in bulk. It is typically used to store an entire program.

Ferroelectric Random Access Memory (FRAM) is a nonvolatile, ultra-low power (ULP) with access speeds similar to RAM. It has been termed a universal memory because it can be used for storing program code, variables, constants, and for stack operations. Note these functions are typically performed by nonvolatile ROM and volatile RAM. FRAM also has a high level of write endurance on the order of 10^{15} cycles [SLAA526A, 2014, SLAA628, 2014].

The input and output system of a microcontroller usually consists of a complement of ports. Ports are fixed sized hardware registers that allow for the orderly transfer of data in and out of the microcontroller. In most microcontroller systems, ports are equipped for dual use. That is, they may be used for general-purpose digital I/O operations or may have alternate functions such as input access for the analog-to-digital (ADC) system.

Our discussion thus far has been about microcontrollers in general. For the remainder of this chapter and the rest of the book, we concentrate on the Texas Instruments MSP430 microcontroller, specifically the MSP430FR2433 and the MSP430FR5994.

1.4 WHY THE TEXAS INSTRUMENTS MSP430?

The MSP430 line of microcontrollers began development in 1992. Since this initial start, there have been multiple families of the microcontroller developed and produced with a wide range of features. This allows one to choose an appropriate microcontroller for a specific application. Texas Instruments invests considerable resources in providing support documentation, development tools, and instructional aids for this processor family.

The various families of the MSP430 have the following traits [SLAB034AD, 2017]:

- low-power supply range,
- ultra-low power (ULP) consumption,
- 16-bit reduced instruction set (RISC) architecture,
- over 300 code compatible products,
- capability to integrate digital and analog components,
- compatible radio frequency (RF) peripheral components to provide wireless communications,
- onboard ADC and digital-to-analog converter (DAC) system, and
- full range of documentation and support for the student, design engineer, and instructor.

1.5 TARGET MICROCONTROLLER FEATURES

Figure 1.2 provides a summary of features for the MS430-EXPFR2433 (FR2433) and the MSP-EXP430FR5994 (FR5994) evaluation modules (EVM). We discuss features common to both, followed by features specific to the FR5994.

The FR2433 and FR5994 possess the following common features [SLAB034AD, 2017, SLASE59D, 2018, SLASE54C, 2018]:

- operating frequency up to 16 MHz,
- 16-bit RISC architecture,
- low supply voltage range: 1.8–3.6 VDC,
- ULP consumption,
- brown-out reset,
- large complement of general-purpose I/O pins,
- multi-channel, high-resolution analog-to-digital converter,
- hardware multiplier,
- multiple enhanced universal serial communication interfaces (eUSCI),
- multiple basic timers,
- memory system including FRAM for program and data memory and SRAM,

MSP-EXP430FR2433 Evaluation Module (EVM) FR2xx/FR4xx family	**MSP-EXP430FR5994 Evaluation Module (EVM) FRxx FRAM Family**

- Speed: 16 MHz
- Voltage: 1.8–3.6 VDC
- Ports
  - PA(16), PB(3)
 	
- ADC 10-bit, 8-channel conversion
- Hardware multiplier (MPY), 32 bit
- Universal serial comm interface (eUSCI)(3)
  - A0, A1:
  - UART, IrDA, SPI
  - B0:
  - I2C, SPI
- Basic timer
  - Timer_A2 (2)
  - Timer_A3 (2)
- Memory:
  - FRAM: 15.5 KB
  - 15K program
  - 512 B data
  - 4 KB SRAM
 	
- Power Mgt Module (PMM)
 	
- Clock System (CS) and distribution
- Cyclic Redundancy Check (CRC), 16-bit
- Embedded Emulation Module (EEM)
- Watch dog timer, 16-bit
- Real-time Clock, 16-bit
- JTAG
- Spy-Bi-Wire (SBW)
	- Speed: 16 MHz
	- Voltage 1.8–3.6 VDC
	- Ports
	  - PA(16), PB(16), PC(16) PD(16), PJ(8)
	- ADC 12-bit, 16-channel conversion
	- Hardware multiplier (MPY), 32 bit
	- Universal serial comm interface (eUSCI)(8)
	  - A0, A1, A2, A3:
	  - UART, IrDA, SPI
	  - B0, B1, B2, B3:
	  - I2C, SPI
	- Basic timer (4)
	  - TA0, 1, 4
	  - TB0
	- Memory:
	  - FRAM: 256 KB
	  - flexible configuration
	  - xxxK program
	  - xxxB info
	  - 8 KB SRAM
	- Supply voltage supervisor (SVS)
	- Clock System (CS) and distribution
	- Cyclic Redundancy Check (CRC), 16- or 32-bit
	- Embedded Emulation Module (EEM)
	- Watch dog timer
	- Real-time Clock
	- JTAG
	- Spy-Bi-Wire (SBW)
	- Analog comparator (Comp_A), 16-channel
	- Direct memory access (DMA), 6-channel
	- Onboard LCD controller
	- Capacitive touch
	- Hardware encryption (AES), 128- or 256-bit

Figure 1.2: MSP430 features [SLAB034AD, 2017, SLASE59D, 2018, SLASE54C, 2018].

- power management features,
- flexible clock system (CS),
- cyclic redundancy check (CRC),
- embedded emulation module (EEM),
- watch dog timer (WDT),
- real-time clock (RTC),
- joint test action group (JTAG) interface,
- spy-bi-wire (SBW) JTAG interface,
- onboard light crystal display (LCD) controller (FR5994), and
- low-energy accelerator (LEA) (FR5994).

In addition, the MSP-EXP430FR5994 EVM has the following features:

- analog comparators,
- six channels of internal direct memory access (DMA),
- onboard LCD controller,
- capacitive touch features, and
- hardware AES encryption.

Provided below is a brief summary of these features. More details are found throughout the remainder of the book [SLAB034AD, 2017, SLASE59D, 2018, SLASE54C, 2018]:

Maximum operating frequency: 16 MHz. Both the FR2433 and the FR5994 EVMs have a maximum operating frequency of 16 MHz. Generally speaking, a microcontroller's power consumption is linearly related to its operating frequency. Microcontrollers are typically used in remote applications sourced by battery power. To conserve energy the microcontroller is placed in low-power, low-frequency sleep mode when inactive. The microcontroller is awoken when needed. Therefore, a microcontroller needs a combination of both high- and low-frequency clock sources, different operating modes, and the ability to quickly transition between the modes. The MSP430 is equipped with a flexible and stable clock and distribution system to satisfy these requirements.

16-bit RISC architecture. Reduced instruction set computer (RISC) architecture is based on the premise of designing a processor that is very efficient in executing a basic set of building

block instructions. From this set of basic instructions more complex instructions may be constructed. The 16-bit data width establishes the range of numerical arguments that may be used within the processor. For example, a 16-bit processor can easily handle 16-bit unsigned integers. This provides a range of unsigned integer arguments from 0 to $(2^{16} - 1)$ or approximately 65,535. Larger arguments may be handled, but additional software manipulation is required for processing, which consumes precious execution time.

Low supply voltage range: 1.8–3.6 VDC. The MSP430 operates at very low voltages. Some operating voltage of interest include:

- 3.6 V: close to Li-ion battery supply range (rechargeable electronic battery);
- 1.8–3.6 V: 2x AA or AAA batteries, coin-cell applications, and energy harvesting applications. In energy harvesting techniques, energy is derived from sources external to the microcontroller; and
- 1.8 V: many modern sensors/consumer electronics operate at 1.8 V. Being able to run the microcontroller at this range means the whole system can operate at VCC = 1.8 V.

Ultra-low power consumption. The MSP430 has a variety of operating modes including an active mode (AM) and multiple low-power modes (LPM). In the active mode, the MSP430 draws 126 (FR2433)/118 (FR5994) microamps of current per MHz of clock speed. In the standby mode the MSP430 draws less than one microamp of current. In LPM 3.5 and operating from a RTC frequency of 32,768 Hz, the MSP430 draws 73 nA (FR2433) and 350 nA (FR5994). In LPM 4.5 shutdown mode, the MSP430 draws 16 nA (FR2433) and 45 nA (FR5994) of current [SLASE59D, 2018, SLASE54C, 2018].

Large complement of I/O ports. The FR2433 is equipped with a single 16-bit digital I/O port, designated as PA. This port may also be subdivided into two 8-bit ports, designated as P1 and P2. The FR2433 also has a 3-bit, port designated as PB. The FR5994 is equipped with a four 16-bit digital I/O ports, designated as PA through PD. These ports may also be subdivided into two 8-bit ports. For example, port PA may be as designated P1 and P2. The FR5994 also has an 8-bit port, designated as PJ.

Multi-channel, high-resolution analog-to-digital converter. The FR2433 is equipped with an 8-channel, 10-bit analog-to-digital converter. The FR5994 is equipped with 16 channels of 12-bit ADC. This feature provides for a large number of converters and very good resolution of analog signals converted.

Hardware multiplier. Many microcontrollers can perform mathematical multiplication operations. However, most perform these calculations using a long sequence of instructions that consume multiple clock cycles. That is, it takes a relatively long period of time to perform a multiplication operation. The MSP430 is equipped with a dedicated hardware multiplier that can

multiply 32-, 24-, 16-, and 8-bit signed and unsigned multiplication operations. The hardware multiplier can also perform the signed and unsigned multiply and accumulate operation. This operation is used extensively in digital signal processing (DSP) operations.

Multiple enhanced universal serial communication interfaces (eUSCI). The MSP430 microcontroller is equipped with the enhanced Universal Serial Communication Interface (eUSCI). The system is equipped with many different serial communication subsystems. The eUSCI consists of two different communication subsystems: eUSCI A type modules and eUSCI B modules. The FR2433 EVM is equipped with two A modules (A0, A1) and a single B module (B0). The FR5994 EVM is equipped with four A modules (A0 to A3) and four B modules (B0 to B3).

The eUSCI A modules provide support for the following.

- Universal asynchronous serial receiver and transmitter (UART). The UART supports a serial data link between a transmitter and a receiver. The transmitter and receiver pair maintains synchronization using start and stop bits that are embedded in the data stream.
- Infrared data association (IrDA). The IrDA protocol provides for a short-range data link using an infrared (IR) link. It is a standardized protocol for IR linked devices. It is used in various communication devices, personal area networks, and instrumentation.
- The serial peripheral interface (SPI) provides synchronous communications between a receiver and a transmitter. The SPI system maintains synchronization between the transmitter and receiver pair using a common clock provided by the master designated microcontroller. An SPI serial link has a much faster data rate than UART.

The eUSCI B modules also provide support for SPI communications and inter-integrated communication (I^2C) communications. The I^2C is one of prominent communication modes, used when multiple serial devices are interconnected through a serial bus. The I^2C bus is a two-wire bus with the serial data line (SDL) and the serial clock line (SCL). By configuring devices connected to the common I^2C line as either a master device or a slave device, multiple devices can share information using a common bus. The I^2C system is used to link multiple peripheral devices to a microcontroller or several microcontrollers together in a system that are in close proximity to one another [SLAU356A].

Multiple basic timers. The MSP430 employs timers for capturing the parameters of an incoming signal (period, frequency, duty cycle, pulse length), generating a precision output digital signal, or generating a pulse width modulated (PWM) signal. The FR2433 is equipped with four 16-bit timers designated as type Timer_A3 and Timer_A2. Each of the two Timer_A3 timers are equipped with three capture/compare registers. The two Timer_A2 timers are equipped with two capture/compare registers. The FR5994 is equipped with 6 different 16-bit registers with the number of capture/compare registers shown: Timer_TA0(3), Timer_TA1(3), Timer_TA2(3), Timer_TA3(2), Timer_TA4(2), and Timer_TB0(7).

Memory system. FRAM is a nonvolatile and operates on ULP with access speeds similar to RAM. It has been termed a universal memory because it can be used for storing program code, variables, constants, and for stack operations. The FR2433 is equipped with 15 KB (kilobytes) of FRAM for program storage and 512 bytes for data storage. The FR5994 is equipped with 256 KB of flexibly configurable FRAM for program and information storage.

Onboard RAM memory. The FR2433 hosts a 4 KB static RAM (SRAM) memory; whereas, the FR5994 is equipped with 8 KB of SRAM memory. The SRAM memory is used for global variables, local variables, and the dynamic allocation of user-defined data types during program execution.

Power management system. The power management module (PMM) supplies the core voltage for the microcontroller. It consists of an integrated voltage regulator to maintain a stable core voltage. It is also equipped with a supply voltage supervisor and monitoring system, which may be configured to reset the microcontroller when the core voltage falls below a preset value.

Flexible clock system. Microcontrollers are synchronous circuits. That is, all microcontroller operations are synchronized with a clock circuit. There are several clock source options available for the MSP430 including a 32 kHz crystal (XT1), an internal very low-frequency oscillator (VLO), an internal trimmed low-frequency oscillator (REFO), an integrated digitally controlled oscillator (DCO) employing a frequency logic loop (FLL) with a digital modulator, and a high-frequency crystal oscillator (XT1 or XT2).

Cyclic redundancy check (CRC) generator. Data stored aboard a microcontroller may be corrupted by noise sources flipping 1s to 0s and vice versa. The MSP430 is equipped with the CRC16 subsystem, which employs the CRC-CCITT standard to calculate a checksum for a block of data. The checksum is also stored with the data. When the data is used, a checksum is calculated again and compared to the stored value. If the values agree, the data is considered good for use. Alternately, if the checksums do not agree, the data is considered corrupted and not available for use. The CRC system onboard the FR5994 EVM may also be used in a 32-bit mode.

Embedded emulation module (EEM). The EEM is used to troubleshoot system operation of an embedded system. It allows events to be triggered based on memory access, CPU access, or hardware triggers.

Watch dog timer (WDT). The WDT is a timer that, if expired, results in a processor reset. It is used to reset the processor when a software malfunction has occurred. During normal program processing the WDT is reset by specific program steps. Should a software malfunction occur and the WDT timer is not reset, the WDT will timeout and result in a processor reset. In response to the processor reset, the software malfunction may clear.

Real-time clock (RTC). Microcontrollers keep time based on elapsed clock ticks. They do not "understand" the concepts of elapsed time in seconds, hours, etc. The RTC provides a general-purpose 16-bit counter while in the Counter Mode or an RTC in the Calendar Mode. Both timer modes can be used to read or write counters using software.

Joint test action group (JTAG) interface. JTAG is a four-wire standard interface to send and receive data from the microcontroller. It is defined within IEEE Standard 1149.1. The JTAG interface allows for programming and debugging programs.

Spy-bi-wire (SBW) JTAG interface. The SBW is a two-wire JTAG compatible interface.

In addition to the features common between the FR2433 and the FR5994 EVMs, the FR5994 is also equipped with the following subsystems.

Analog comparator. The FR5994 is equipped with a 16-channel analog comparator to monitor analog signals of interest. The comparator, as its name implies, compares an analog input signal with a pre-defined threshold.

Direct memory access (DMA). Memory transfer operations from one location to another typically requires many clock cycles involving the CPU. The FR5994 is equipped with six DMA channels that allow memory-to-memory location transfers without involving the CPU, freeing the CPU to perform other instructions.

Capacitive touch. The FR5994 is equipped with capacitive touch I/O features that support simple touch screen applications.

Hardware encryption. The FR5994 is also equipped with a hardware-based advanced encryption standard (AES) accelerator. The accelerator speeds up the encryption and decryption of data by one to two orders of magnitude over a software-based implementation. It can be used for 128- or 256-bit encryption.

Low-energy accelerator. The FR5994 is equipped with a 32-bit, fixed-point processor, the low energy accelerator (LEA) for vector-based operations. Vector-based operations are commonly used is signal processing applications. Operations include finite impulse response (FIR) filtering, infinite impulse response (IIR) filtering, fast fourier transforms (FFT), inverse fast fourier transforms (IFFT), and others. The LEA, once configured for operation, executes commands independent of the MSP430 CPU SLAU367O [2017].

1.6 INTRODUCTION TO THE EVALUATION MODULES (EVM)

Throughout the rest of the book we use the MSP-EXP430FR2433 and the MSP-EXP430FR5994 EVMs to illustrate operation of different systems aboard the MSP430.

MSP430FR2433 EVM. The MSP-EXP430FR2433 EVM layout is provided in Figure 1.3 and the pinout diagram in Figure 1.4. The FR2433 is programmed via a host PC or laptop via a USB cable. The upper portion of the EVM is equipped with the eZ-FET Debug Probe. This provides for EVM programming and also communication back to the host computer. Also, the EVM is equipped with EnergyTrace Technology that allows power consumption readings. The link from the upper to the lower board is provided by the Jumper Isolation Block. The lower board is equipped with two switches, two light-emitting diodes (LEDs), and breakout pins for a 20-pin BoosterPack. BoosterPacks allow the features of the MSP430 to be extended [SLAU739, 2017].

MSP-EXP430FR5994 EVM. The MSP-EXP430FR5994 EVM layout is provided in Figure 1.5 and the pinout diagram Figure 1.4. The FR5994 is programmed via a host PC or laptop via a USB cable. The upper portion of the EVM is equipped with the eZ-FET Debug Probe. This provides for EVM programming and also communication back to the host computer. Also, the EVM is equipped with EnergyTrace Technology that allows power consumption readings. The link from the upper to the lower board is provided by the Jumper Isolation Block. The lower board is equipped with two switches, two LEDs, and breakout pins for a 40-pin BoosterPack. BoosterPacks allow the features of the MSP430 to be extended. The FR5994 is also equipped with a Micro SD card [SLAU678A, 2016].

1.7 DEVELOPMENT SOFTWARE

There are multiple software tools available to support the MSP430 line of microcontrollers from Texas Instruments and third-party producers. Throughout the book, we use Texas Instruments Energia and code composer studio (CCS) integrated development environment (IDE).

Energia is an open-source IDE modeled after the Arduino Sketchbook concept. It allows for rapid prototyping of a wide range of Texas Instruments microcontroller products. We use it to rapidly prototype programs and embedded systems using the FR2433 and FR5994 EVMs. In space lore, the Energia was a Soviet heavy lift rocket. Similarly, the Energia IDE performs heavy lifting when learning software programming for the first time (`www.energia.nu`).

The CCS is used to develop code for all of TI's digital processors including digital signal processors (DSPs), microcontrollers, and application processors. The Platinum version provides full product line support. The Microcontroller version, a subset of the Platinum version, provides support for the MSP430 family of microcontrollers and other related product lines. In addition to code development, CCS may be used for debugging and simulation (`www.TI.com`).

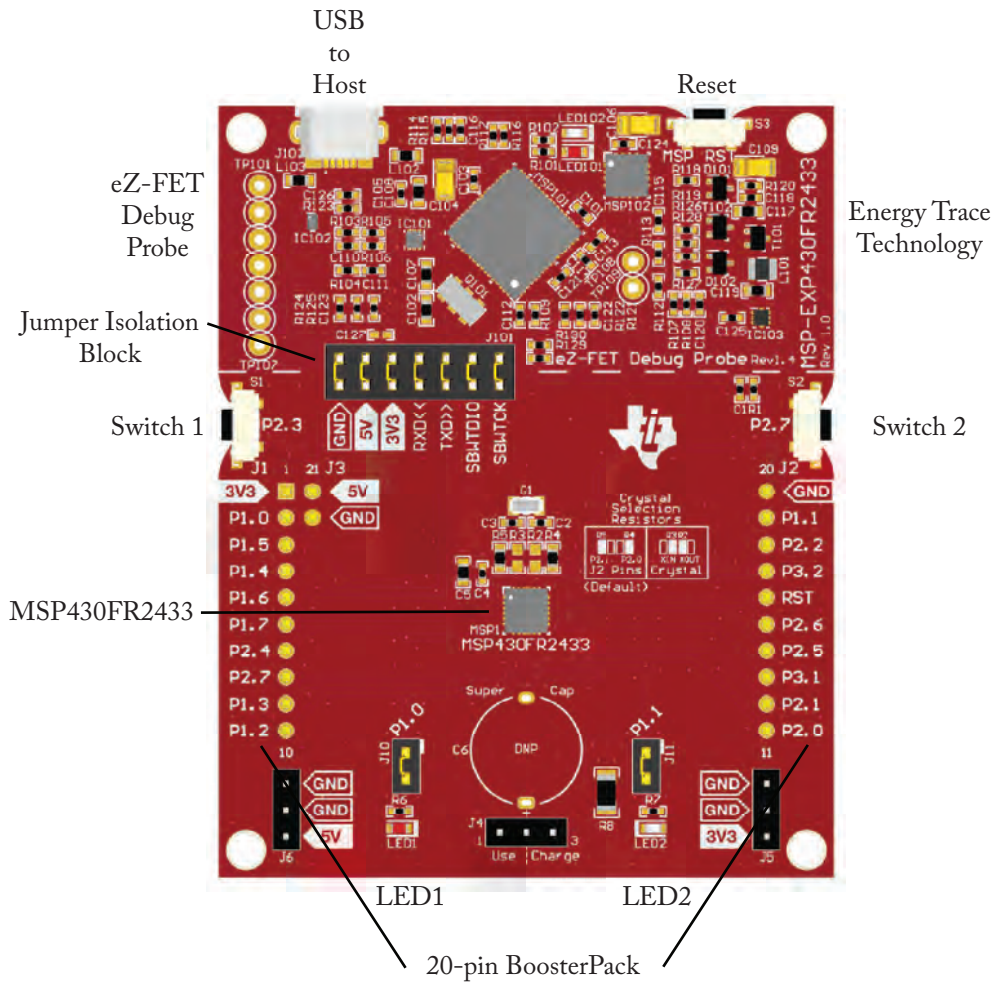


Figure 1.3: FR2433 EVM [SLAU739, 2017]. (Illustration used with permission of Texas Instruments (www.ti.com).)

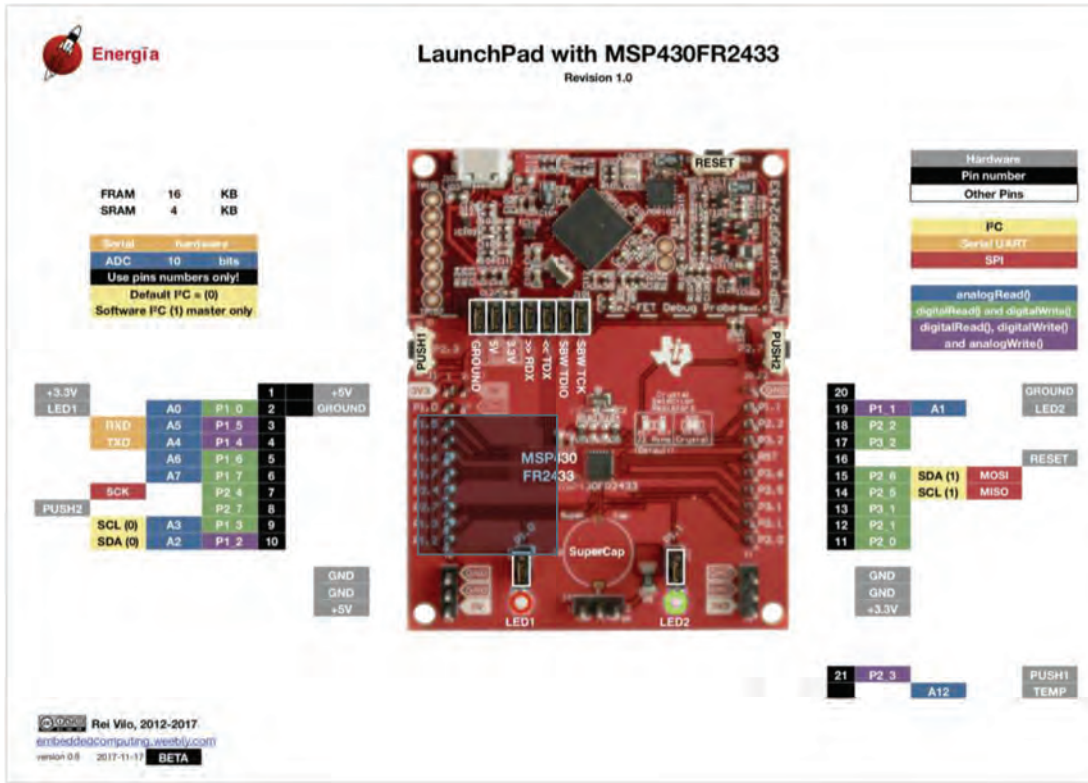


Figure 1.4: FR2433 EVM pinmap [SLAU739, 2017]. (Illustration used with permission of Texas Instruments (www.ti.com).)

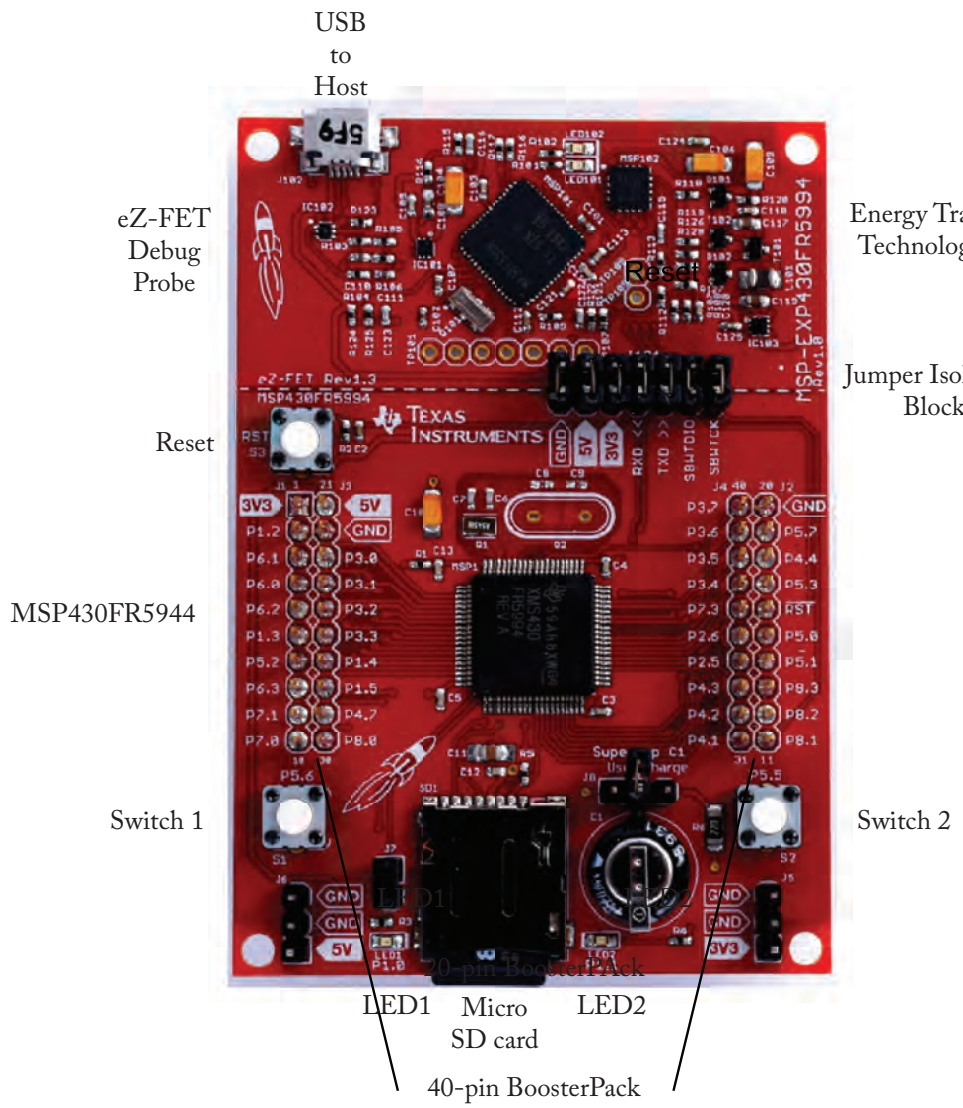


Figure 1.5: FR5994 EVM [SLAU678A, 2016]. (Illustration used with permission of Texas Instruments (www.ti.com).)

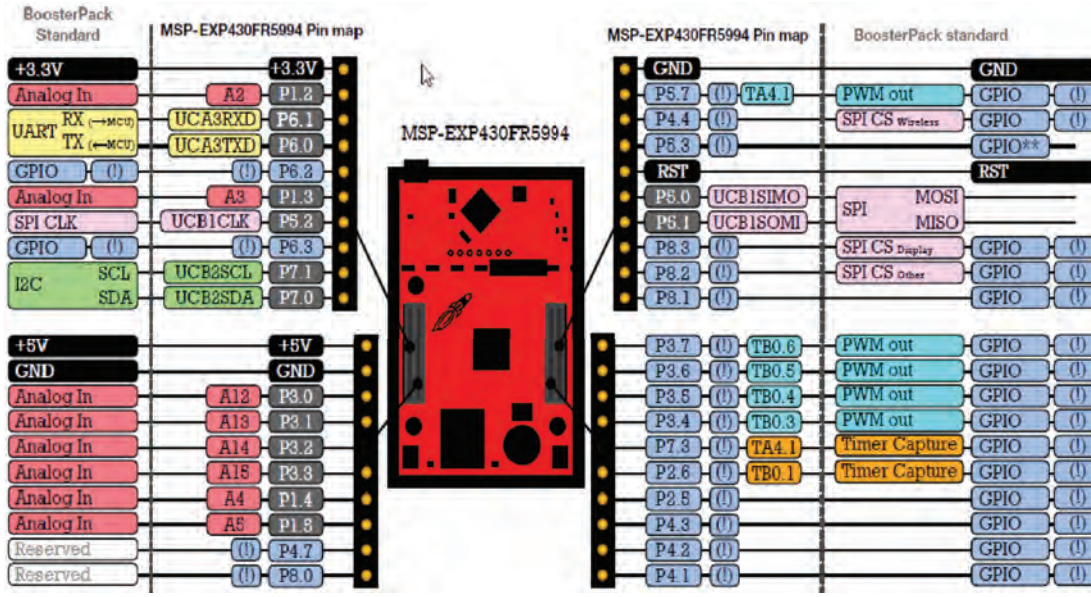


Figure 1.6: FR5994 EVM pinmap [SLAU678A, 2016]. (Illustration used with permission of Texas Instruments (www.ti.com).)

1.8 LAB 1: GETTING ACQUAINTED WITH HARDWARE AND SOFTWARE DEVELOPMENT TOOLS

Introduction. Through this laboratory exercise, you will become familiar with the Texas Instruments MSP430-EXP430FR2433 or the MSP430-EXP430FR5994 EVMs and the Energia IDE.

Procedure: To start working with Energia, follow these steps (energia.nu).

- Download and install the latest version of Energia from the energia.nu website to the host computer. It is available for different operating systems including: Windows, Mac OS X, and Linux.
- Launch Energia on the host computer by going to the Energia folder and clicking on the Energia icon. The icon is a red ball with a rocket silhouette.
- To install the latest Board Manager, go to Tools->Boards Manager->Online Help.
- Connect the LaunchPad to the host computer via the USB cable provided with the EVM.
- With Energia launched, go to Tools->Board-> and select the LaunchPad.

- Check the comm port setting using Tools->Serial Port.
- To load the first example use File->Examples->Basics->Blink.
- To compile, upload, and run the program, use the Upload icon (right facing arrow).
- The red LED on the LaunchPad will blink!

Investigate some of the other sample programs provided in Energia.

1.9 SUMMARY

In this chapter, we introduced microcontrollers and an overview of related technologies. We began with a brief review of computer development leading up to the release of microcontrollers, reviewed microcontroller related terminology and provided an overview of systems associated with the MSP-EXP430FR2433 and the MSP-EXP430FR5994 evaluation modules. The remainder of the chapter was spent getting better acquainted with the EVMs and the Energia IDE.

1.10 REFERENCES AND FURTHER READING

Bartee, T. *Digital Computer Fundamentals*, 3rd ed., New York, McGraw-Hill, Inc., 1972. 2

Boone, G. *Computing System CPU*, United States Patent 3,757,306 filed August 31, 1971, and issued September 4, 1973. 3

Boone, G. *Variable Function Programmable Calculator*, United States Patent 4,074,351 filed February 24, 1977 and issued February 15, 1978. 3

MCS 85 User's Manual, Intel Corporation, 1977. 3

MSP Low-Power Microcontrollers, (SLAB034AD), Texas Instruments, 2017. 5, 6, 7

MSP430FR2433 LaunchPad Development Kit (MSP-EXP430FR2433), (SLAU739), Texas Instruments, 2017. 12, 13, 14

MSP430FR2433 Mixed-Signal Microcontroller, (SLASE59D), Texas Instruments, 2018. 5, 6, 7, 8

MSP430FR58xx, MSP430FR59xx, and MSP430FR6xx Family User's Guide (SLAU367O), Texas Instruments, 2017. 11

MSP430FR5994 LaunchPad Development Kit (MSP-EXP430FR5994), (SLAU678A), Texas Instruments, 2016. 12, 15, 16

MSP430FR599x, MSP430FR596x Mixed-Signal Microcontrollers, (SLASE54C), Texas Instruments, 2018. 5, 6, 7, 8

Nobelprize.org. *The Official Web Site of the Nobel Prize*, `www.nobelprize.org` 2

Osborne, A. *An Introduction to Microcomputers Volume 1 Basic Concepts*, 2nd ed., Berkeley, Osborne/McGraw-Hill, 1980. 2, 3

Patterson, D. and Hennessy, J. *Computer Organization and Design the Hardware/Software Interface*, San Francisco, 1994. 3

Texas Instruments MSP430 FRAM Quality and Reusability, (SLAA526A), Texas Instruments, 2014. 4

Texas Instruments MSP430 FRAM Technology-How to and Best Practices, (SLAA628), Texas Instruments, 2014. 4

1.11 CHAPTER PROBLEMS

Fundamental

1. Define the terms microprocessor, microcontroller, and microcomputer. Provide an example of each.
2. What were the catalysts that led to the multiple generations of computer processors?
3. What are the five main components of a computer architecture? Briefly define each.
4. Distinguish between RAM, ROM, EEPROM, and FRAM memory. Provide an example application of how each are employed within a microcontroller.
5. What is RISC architecture? What is its fundamental premise?

Advanced

1. List the key features of the MSP430 families of microcontrollers.
2. List the key features of the MSP430FR2433 mixed signal processor.
3. List the key features of the MSP430FR5994 mixed signal processor.
4. Describe the tradeoff between processor speed and power consumption. How does the MSP430 meet these competing demands?

Challenging

1. Write a single page paper on a specific generation of computers.

2. Research the difference between CISC and RISC computer architectures. Provide the main features of each approach. Which approach is better suited for microcontroller applications?
3. Research IrDA infrared communication standards. Write a single page paper on the topic.
4. Research the CRC-CCITT standard used to calculate a checksum. Write a single page paper on the topic.

CHAPTER 2

A Brief Introduction to Programming

Objectives: After reading this chapter, the reader should be able to:

- use the Energia Integrated Development Environment to interface with the MSP-EXP430FR2433 and the MSP-EXP430FR5994 LaunchPads;
- describe key components of a C program;
- specify the size of different variables within the C programming language;
- define the purpose of the main program;
- explain the importance of using functions within a program;
- write functions that pass parameters and return variables;
- describe the function of a header file;
- discuss different programming constructs used for program control and decision processing; and
- write programs in C for execution on the MSP-EXP430FR2433 and the MSP-EXP430FR5994 LaunchPads.

2.1 OVERVIEW

The goal of this chapter is to provide a tutorial on how to begin programming on the MSP430 microcontroller.[1] We begin with an introduction to programming using the Energia integrated development environment (IDE), followed by an introduction to programming in C. Throughout the chapter, we provide examples and pointers to several excellent references.

[1]This chapter was adapted with permission from Barret (2013).

2.2 ENERGIA

Energia is an open-source IDE modeled after the Arduino Sketchbook concept. It allows for rapid prototyping of a wide range of Texas Instruments microcontroller products. We use it to prototype programs and embedded systems using the MSP-EXP430FR2433 and the MSP-EXP430FR5994 LaunchPads. In space lore, the Energia was a Soviet heavy lift rocket. Similarly, the Energia IDE performs heavy lifting when learning software programming for the first time.

2.3 ENERGIA QUICKSTART

To quickly get up and operating with Energia, complete the following steps (`energia.nu`).

- Download and install the latest version of Energia from the `energia.nu` website to the host computer. It is available for different operating systems including: Windows, Mac OS X, and Linux.
- Launch Energia on the host computer by going to the Energia folder and clicking on the Energia icon. The icon is a red ball with a rocket silhouette.
- To install the latest Board Manager, go to Tools->Boards Manager->Online Help.
- Connect the LaunchPad to the host computer via the USB cable provided with the EVM.
- With Energia launched, go to Tools->Board-> and select the LaunchPad.
- Check the comm port setting using Tools->Serial Port.
- To load the first example use File->Examples->Basics->Blink.
- To compile, upload, and run the program, use the Upload icon (right facing arrow).
- The red LED on the LaunchPad will blink!

With the first program launched, let's take a closer look at the Energia IDE.

2.4 ENERGIA DEVELOPMENT ENVIRONMENT

In this section, we provide an overview of the Energia IDE. We begin with some background information about the IDE and then review its user-friendly features. We then introduce the sketchbook concept and provide a brief overview of the built-in software features within the IDE. Our goal is to provide readers with a brief introduction to Energia features. All Energia related features are well documented on the Energia homepage (`energia.nu`). We will not duplicate this excellent source of material; but merely provide a brief introduction with pointers to advanced features.

2.4.1 ENERGIA IDE OVERVIEW

At its most fundamental level, the Energia IDE is a user-friendly interface to allow one to quickly write, load, and execute code on a microcontroller. A barebones program needs only a setup() function and a loop() function. The Energia IDE adds the other required pieces such as header files and the main program constructs (`energia.nu`).

The Energia IDE is illustrated in Figure 2.1. The IDE contains a text editor, a message area for displaying status, a text console, a tool bar of common functions, and an extensive menu system. The IDE also provides a user-friendly interface to the LaunchPad which allows for the quick compiling and uploading of code.

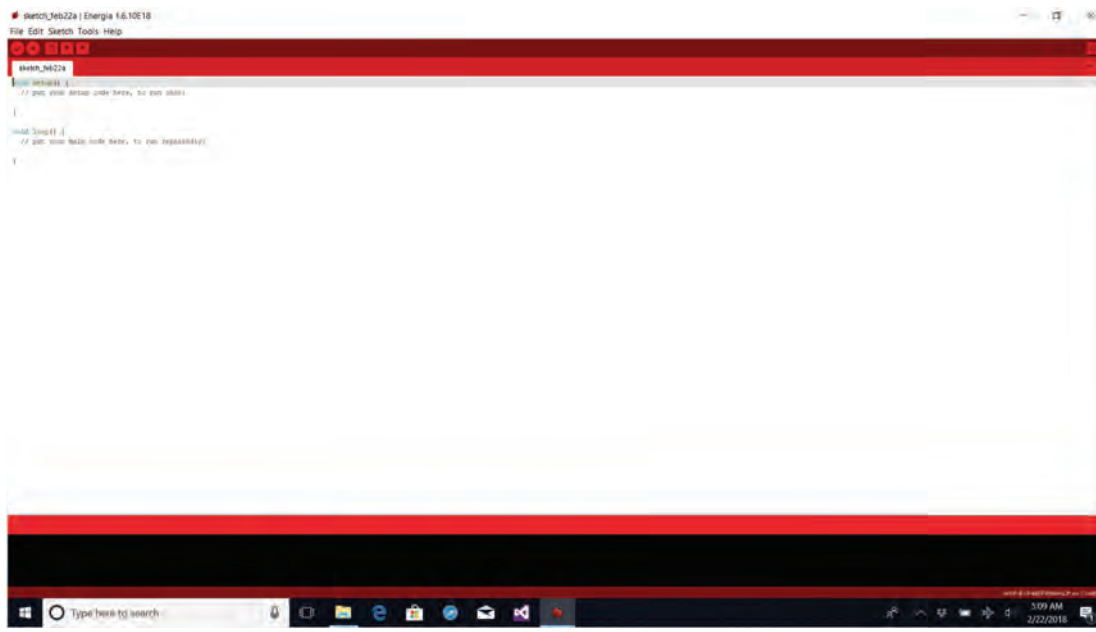


Figure 2.1: Energia IDE (`energia.nu`).

A close-up of the Energia toolbar is provided in Figure 2.2. The toolbar provides single button access to the more commonly used menu features. Most of the features are self-explanatory. The "Upload" button compiles the program and uploads it to the LaunchPad. The "Serial Monitor" button opens a serial monitor to allow text data to be sent to and received from the LaunchPad.

2.4.2 SKETCHBOOK CONCEPT

In keeping with a hardware and software platform for students of the arts, the Energia environment employs the concept of a sketchbook. Artists maintain their works in progress in a

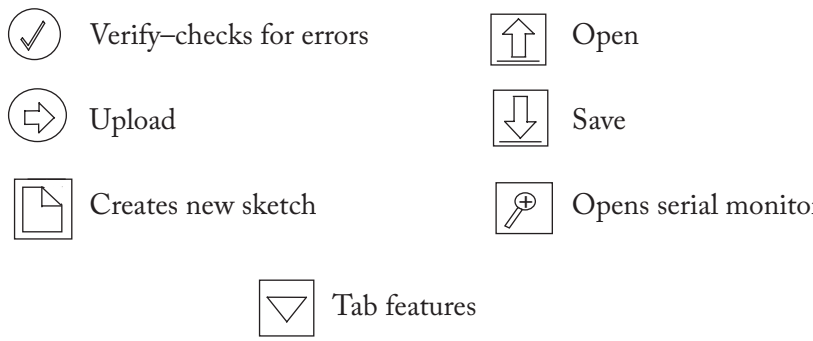


Figure 2.2: Energia IDE buttons.

sketchbook. Similarly, we maintain our programs within a sketchbook in the Energia environment. Furthermore, we refer to individual programs as sketches. An individual sketch within the sketchbook may be accessed via the Sketchbook entry under the file tab.

2.4.3 ENERGIA SOFTWARE, LIBRARIES, AND LANGUAGE REFERENCES

The Energia IDE has a number of built-in features. Some of the features may be directly accessed via the Energia IDE drop-down toolbar illustrated in Figure 2.1. Provided in Figure 2.3 is a handy reference to show the available features. The toolbar provides a wide variety of features to compose, compile, load, and execute a sketch.

Aside from the toolbar accessible features, the Energia IDE contains built-in functions that allow the user to quickly construct a sketch. These built-in functions are summarized in Figure 2.4. Complete documentation for these built-in function is available at the Energia homepage (`energia.nu`). This documentation is easily accessible via the Help tab on the Energia IDE toolbar. We refer to these features at appropriate places throughout the remainder of the book and provide additional background information as needed.

2.5 ENERGIA PIN ASSIGNMENTS

Hardware features onboard the LaunchPad (LEDs, switches, etc.) are accessed via Energia using pin numbers. Pin numbers range from 1–20 for the FR2433 EVM and 1–40 for the FR5994 EVM. Pin diagrams for both evaluation modules are provided in Chapter 1 (Figures 1.3 and 1.5).

2.6 WRITING AN ENERGIA SKETCH

The basic format of the Energia sketch consists of a "setup" function and a "loop" function. The setup function is executed once at the beginning of the program. It is used to configure pins, declare variables and constants, etc. The loop function will execute function sequentially step-

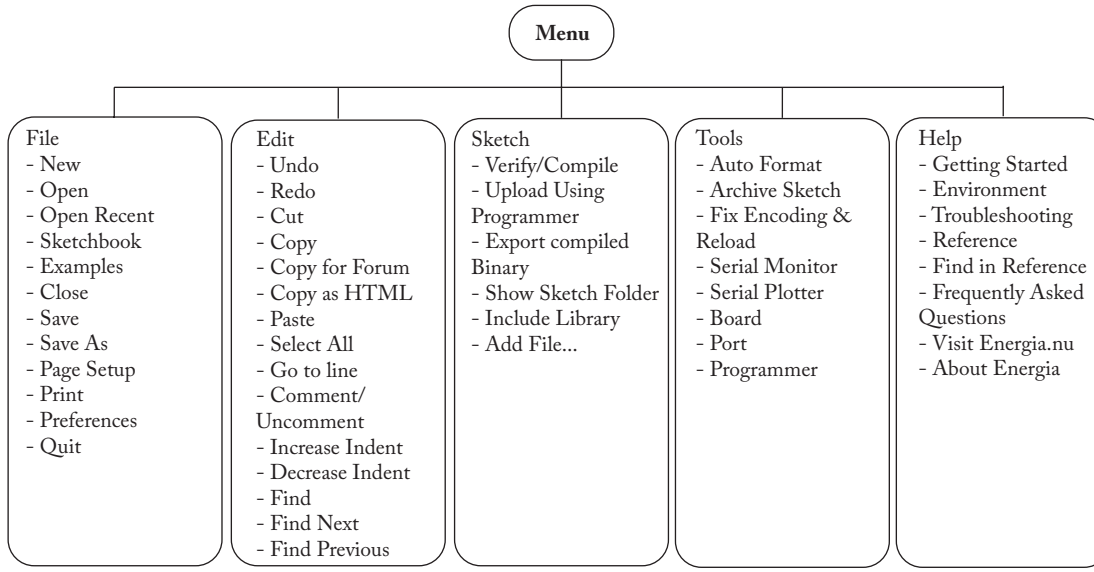


Figure 2.3: Energia IDE menu (`energia.nu`).

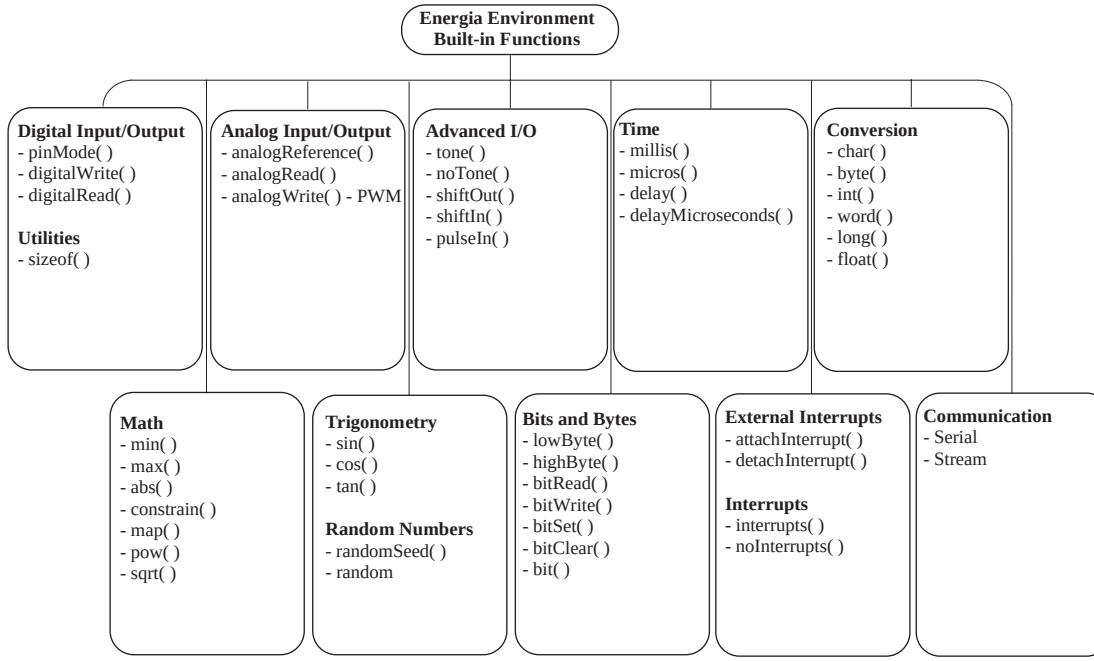


Figure 2.4: Energia IDE built-in features (`energia.nu`).

-by-step. When the end of the loop function is reached, it will automatically return to the first step of the loop function and execute the function again. This goes on continuously until the program is stopped.

```
//*************************************************************

void setup()
  {
  //place setup code here
  }

void loop()
  {
  //main code steps are provided here
  :
  :

  }

//*************************************************************
```

Example: Blink. Let's examine the sketch used to blink the LED (`energia.nu`).

```
//*************************************************************
//Blink---The basic Energia example.
//Turns on an LED on for one second, then off for one second,
//repeatedly. Change the LED define to blink other LEDs.
//Hardware Required: LaunchPad with an LED
//This example code is in the public domain.
//*************************************************************

//most launchpads have a red LED
#define LED RED_LED

//#define LED GREEN_LED

//the setup routine runs once when you press reset:
void setup()
{
//initialize the digital pin as an output.
pinMode(LED, OUTPUT);
```

```
}

//the loop routine runs over and over again forever:
void loop()
{
digitalWrite(LED, HIGH);   //turn the LED on
                           //(HIGH is the voltage level)
delay(1000);               //wait for a second
digitalWrite(LED, LOW);    //turn the LED off by making
                           //the voltage LOW
delay(1000);               // wait for a second
}

//*******************************************************
```

In the first line, the #define statement links the designator "LED" to the pin connected to the red LED on the LaunchPad. In the setup function, LED is designated as an output pin. Recall the setup function is only executed once. The program then enters the loop function that is executed sequentially step-by-step and continuously repeated. In this example, the LED is first set to logic high to illuminate the LED onboard the LaunchPad. Another 1000 ms delay then occurs. The LED is then set low. A 1000 ms delay then occurs. The sequence then repeats. As a second example, comment out the #define statement and remove the comment symbol from in front of the second #define statement. When the modified code is verified and uploaded to the EVM, the onboard green LED will blink.

Aside from the Blink example, there are also many program examples available to allow a user to quickly construct a sketch. They are useful to understand the interaction between the Energia IDE and the LaunchPad. They may also be used as a starting point to write new applications. The program examples are available via the File->Examples tab within Energia. The examples fall within these categories:

1. Basics
2. Digital
3. Analog
4. Communication
5. Control
6. Strings
7. Sensors

8. Display

9. Educational BP Mk II—a multifunction educational development kit containing multiple sensors, an LCD display, and output drivers

10. MultiTasking—allows multiple tasks to be executed simultaneously.

We now examine several more Energia based examples. We use the MSP--EXP430FR2433 LaunchPad in the examples. The MSP-EXP430FR5994 LaunchPad may also be used; however, pin numbers within the examples must be changed from FR2433 pins to FR5994 pins.

Example: External LED. In this example we connect an external LED to the FR2433 LaunchPad pin 18. The onboard green LED will blink alternately with the external LED. The external LED is connected to the LaunchPad as shown in Figure 2.5 using a prototype board. The board is useful for implementing the first prototype of a circuit. Holes within a given column of the board have a common conductor connecting them together. Holes in a row of five are connected via a common conductor. Jumper wires (insulated AWG 22, solid) are used between the MSP430 female pin connectors and the prototype board.

```
//*************************************************************
#define int_LED GREEN_LED
#define ext_LED 18

void setup()
{
pinMode(int_LED, OUTPUT);
pinMode(ext_LED, OUTPUT);
}

void loop()
{
digitalWrite(int_LED, HIGH);
digitalWrite(ext_LED, LOW);
delay(500);              //delay specified in ms
digitalWrite(int_LED, LOW);
digitalWrite(ext_LED, HIGH);
delay(500);
}

//*************************************************************
```

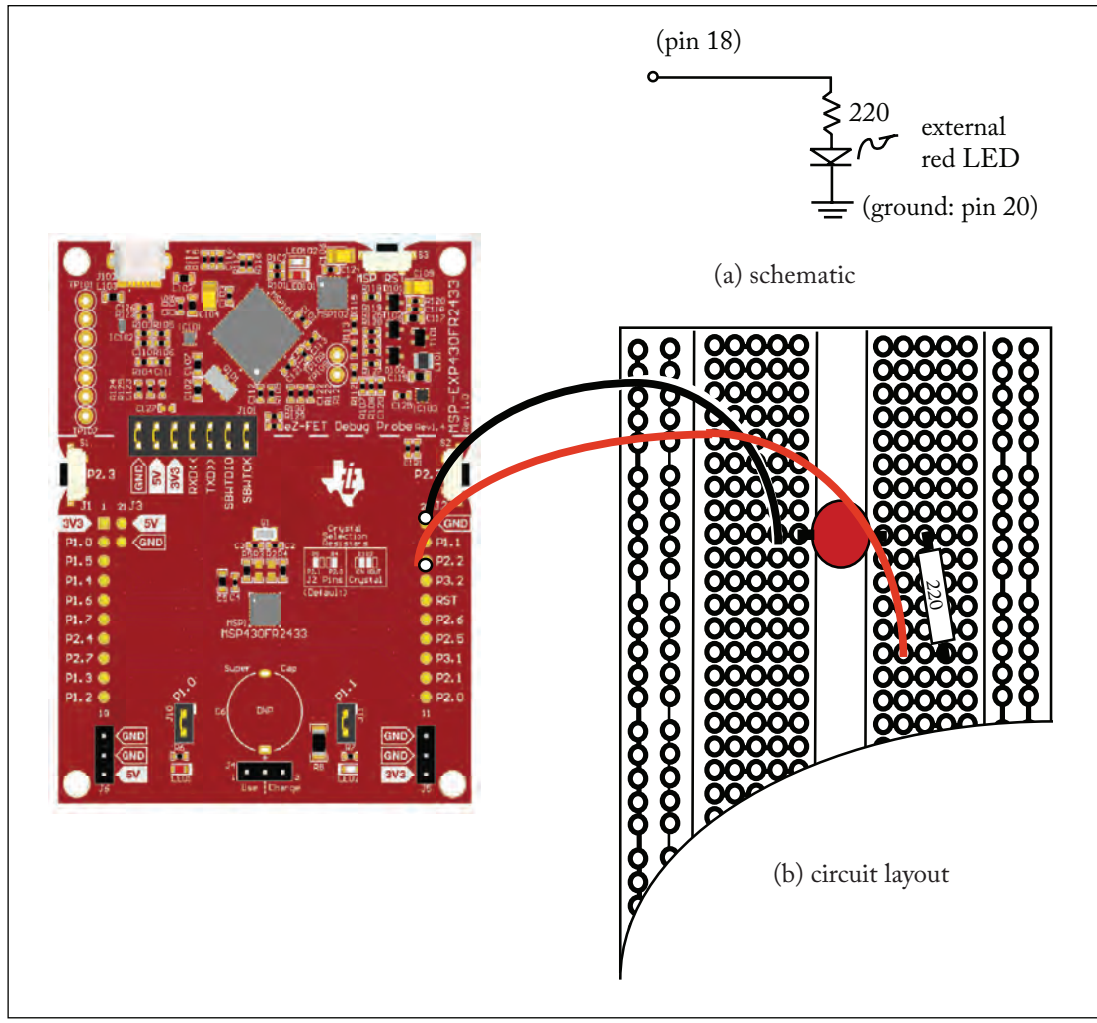


Figure 2.5: LaunchPad with an external LED. (Illustration used with permission of Texas Instruments (www.ti.com).)

Example: External LED and switch. In this example we connect an external LED to LaunchPad pin 18 and an external switch attached to pin 17. The onboard green LED will blink alternately with the external LED when the switch is depressed. The external LED and switch are connected to the LaunchPad, as shown in Figure 2.6.

```
//*********************************************************

#define int_LED GREEN_LED
#define ext_LED 18
#define ext_sw  17

int switch_value;

void setup()
{
pinMode(int_LED, OUTPUT);
pinMode(ext_LED, OUTPUT);
pinMode(ext_sw,  INPUT);
}

void loop()
{
switch_value = digitalRead(ext_sw);
  if(switch_value == LOW)
    {
    digitalWrite(int_LED, HIGH);
    digitalWrite(ext_LED, LOW);
    delay(250);
    digitalWrite(int_LED, LOW);
    digitalWrite(ext_LED, HIGH);
    delay(250);
    }
  else
    {
    digitalWrite(int_LED, LOW);
    digitalWrite(ext_LED, LOW);
    }
}
//*********************************************************
```

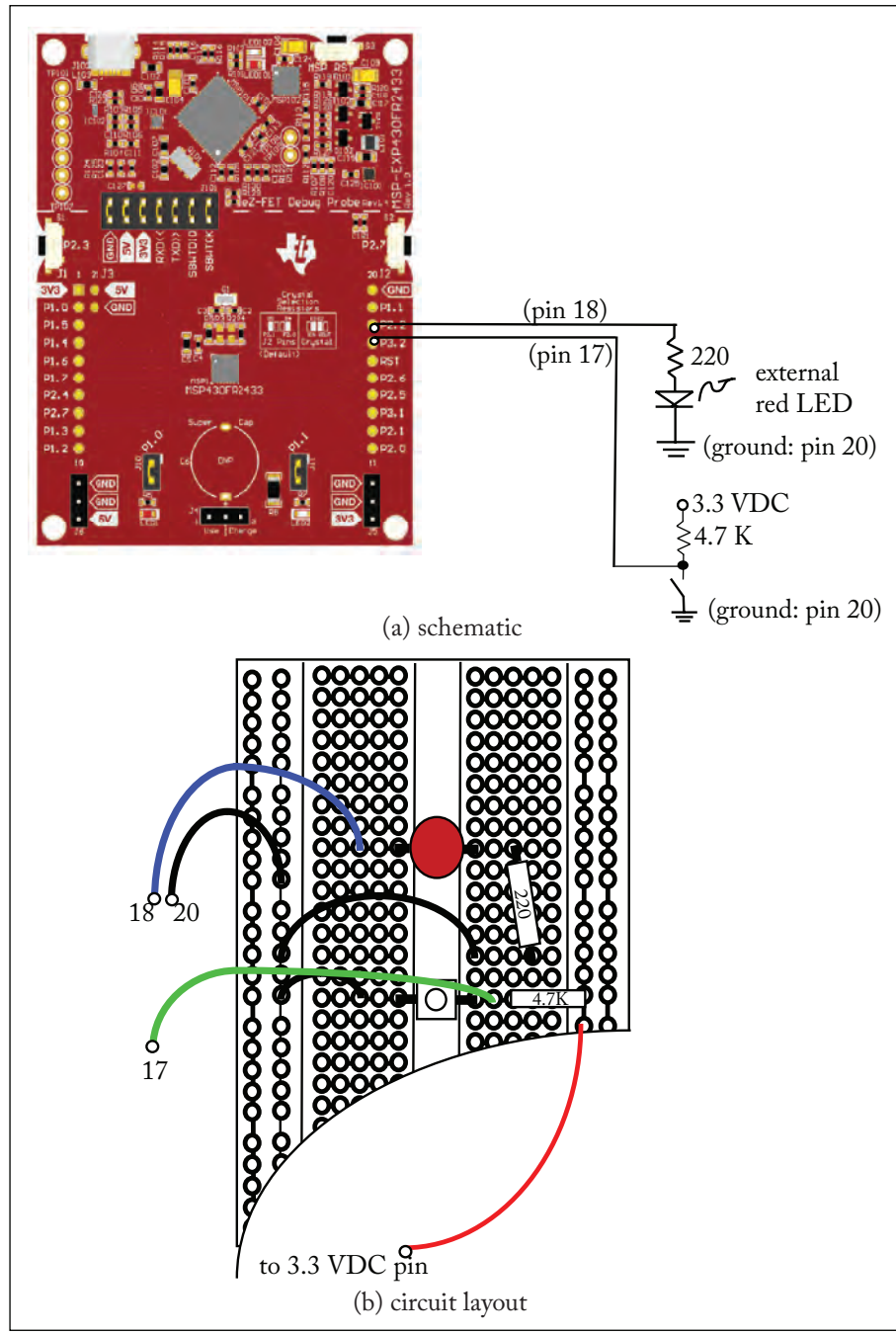


Figure 2.6: LaunchPad with an external LED and switch. (Illustration used with permission of Texas Instruments (www.ti.com).)

Example: LED strip. LED strips may be used for motivational (fun) optical displays, games, or for instrumentation-based applications. In this example we control an LPD8806-based LED strip using Energia. We use a one meter, 32 RGB LED strip available from Adafruit (#306) for approximately $30 USD (www.adafruit.com).

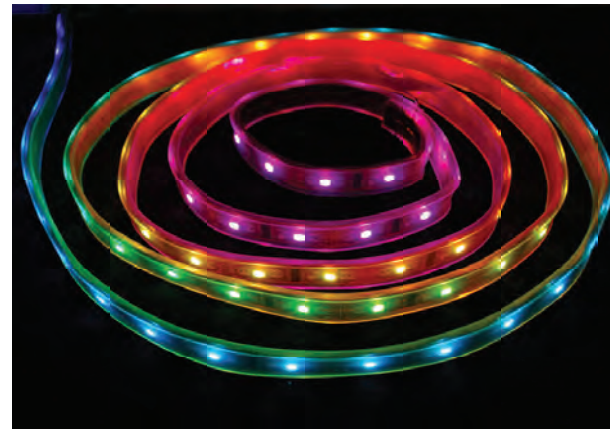
The red, blue, and green component of each RGB LED is independently set using an eight-bit code. The most significant bit (MSB) is logic one followed by seven bits to set the LED intensity (0–127). The component values are sequentially shifted out of the MSP-EXP430FR2433 LaunchPad using the serial peripheral interface (SPI) features. The first component value shifted out corresponds to the LED nearest the microcontroller. Each shifted component value is latched to the corresponding R, G, and B component of the LED. As a new component value is received, the previous value is latched and held constant. An extra byte is required to latch the final parameter value. A zero byte $(00)_{16}$ is used to complete the data sequence and reset back to the first LED (www.adafruit.com).

Only four connections are required between the MSP-EXP430FR2433 LaunchPad and the LED strip as shown in Figure 2.7. The connections are color coded: red-power, black--ground, yellow-data, and green-clock. It is important to note the LED strip requires a supply of 5 VDC and a current rating of 2 amps per meter of LED strip. In this example we use the Adafruit #276 5 V 2A (2000 mA) switching power supply (www.adafruit.com).

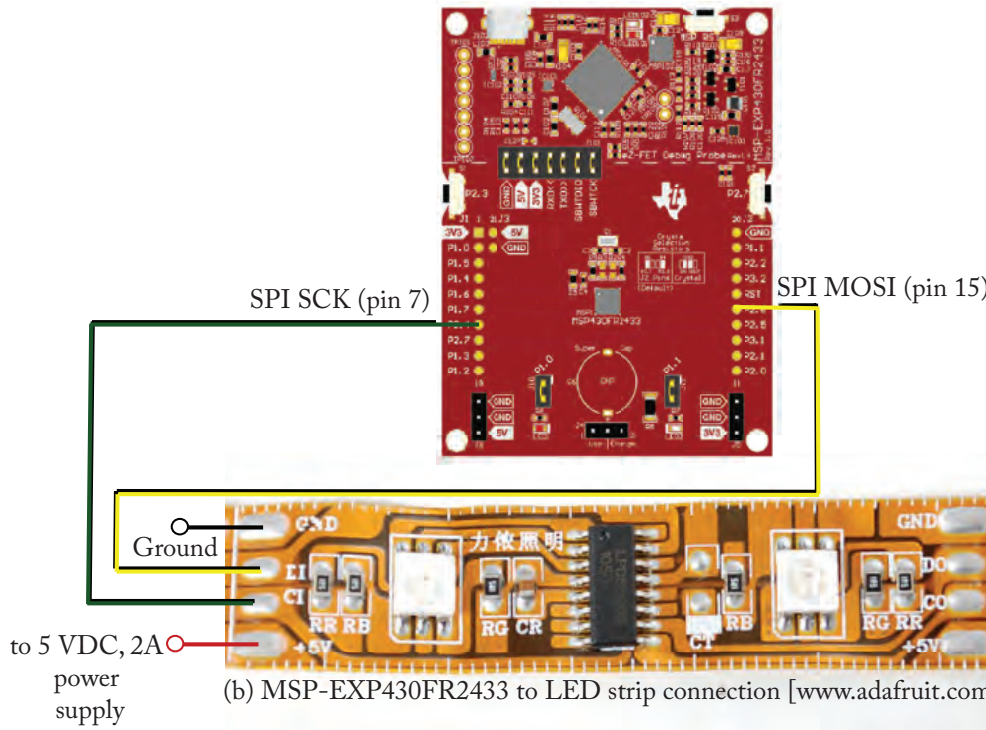
In this example each RGB component is sent separately to the strip. The example illustrates how each variable in the program controls a specific aspect of the LED strip. Here are some important implementation notes.

- SPI must be configured for MSB first.
- LED brightness is 7 bits. MSB must be set to logic one.
- Each LED requires a separate R-G-B intensity component. The order of data is G-R-B.
- After sending data for all LEDs. A byte of (0x00) must be sent to return strip to first LED.
- Data stream for each LED is: 1-G6-G5-G4-G3-G2-G1-G0-1-R6-R5-R4-R3-R2-R1-R0-1-B6-B5-B4-B3-B2-B1-B0

```
//*********************************************************************
//RGB_led_strip_tutorial: illustrates different variables within
//RGB LED strip
//
//LED strip LDP8806 - available from www.adafruit.com (#306)
//
//Connections:
// - External 5 VDC supply - Adafruit 5 VDC, 2A (#276) - red
// - Ground - black
```



(a) LED strip by the meter [www.adafruit.com].



(b) MSP-EXP430FR2433 to LED strip connection [www.adafruit.com].

Figure 2.7: LaunchPad controlling LED strip (www.adafruit.com). (Illustration used with permission of Adafruit (www.adafruit.com).)

```
// - Serial Data In  - LaunchPad pin 15 (MOSI pin) P2.6 - yellow
// - CLK - LaunchPad pin 7 (SCK pin) P2.4 - green
//
//Variables:
// - LED_brightness - set intensity from 0 to 127
// - segment_delay  - delay between LED RGB segments
// - strip_delay    - delay between LED strip update
//
//Notes:
// - SPI must be configured for Most significant bit (MSB) first
// - LED brightness is seven bits.  Most significant bit (MSB)
//   must be set to logic one
// - Each LED requires a seperate R-G-B intensity component.  The order
//   of data is G-R-B.
// - After sending data for all strip LEDs.  A byte of (0x00) must
//   be sent to reutrn strip to first LED.
// - Data stream for each LED is:
//1-G6-G5-G4-G3-G2-G1-G0-1-R6-R5-R4-R3-R2-R1-R0-1-B6-B5-B4-B3-B2-B1-B0
//
//This example code is in the public domain.
//************************************************************************

#include <SPI.h>

#define LED_strip_latch 0x00

const byte strip_length = 32;         //number of RGB LEDs in strip
const byte segment_delay = 100;       //delay in milliseconds
const byte strip_delay = 500;         //delay in milliseconds
unsigned char   LED_brightness;       //0 to 127
unsigned char   position;             //LED position in strip
unsigned char   troubleshooting = 0;  //allows printouts to serial
                                      //monitor

void setup()
  {
  SPI.begin();                         //SPI support functions
  SPI.setBitOrder(MSBFIRST);           //SPI bit order
  SPI.setDataMode(SPI_MODE3);          //SPI mode
```

```
  SPI.setClockDivider(SPI_CLOCK_DIV32);//SPI data clock rate
  Serial.begin(9600);                  //serial comm at 9600 bps
  }

void loop()
  {
  SPI.transfer(LED_strip_latch);       //reset to first segment
  clear_strip();                       //all strip LEDs to black
  delay(500);

  //increment the green intensity of the strip LEDs
  for(LED_brightness = 0; LED_brightness <= 60;
      LED_brightness = LED_brightness + 10)
    {
    for(position = 0; position<strip_length; position = position+1)
      {
      SPI.transfer(0x80 | LED_brightness);  //Green - MSB 1
      SPI.transfer(0x80 | 0x00);            //Red   - none
      SPI.transfer(0x80 | 0x00);            //Blue  - none

      if(troubleshooting)
        {
        Serial.println(LED_brightness, DEC);
        Serial.println(position, DEC);
        }
      delay(segment_delay);
      }
    SPI.transfer(LED_strip_latch);     //reset to first segment
    delay(strip_delay);
    if(troubleshooting)
        {
        Serial.println(" ");
        }
    }

  clear_strip();                       //all strip LEDs to black
  delay(500);

  //increment the red intensity of the strip LEDs
```

```
for(LED_brightness = 0; LED_brightness <= 60;
    LED_brightness = LED_brightness + 10)
  {
  for(position = 0; position<strip_length; position = position+1)
    {
    SPI.transfer(0x80 | 0x00);            //Green - none
    SPI.transfer(0x80 | LED_brightness);  //Red   - MSB1
    SPI.transfer(0x80 | 0x00);            //Blue  - none

    if(troubleshooting)
      {
      Serial.println(LED_brightness, DEC);
      Serial.println(position, DEC);
      }
    delay(segment_delay);
    }
  SPI.transfer(LED_strip_latch);     //reset to first segment
  delay(strip_delay);
  if(troubleshooting)
    {
    Serial.println(" ");
    }
  }

clear_strip();                       //all strip LEDs to black
delay(500);

//increment the blue intensity of the strip LEDs
for(LED_brightness = 0; LED_brightness <= 60;
    LED_brightness = LED_brightness + 10)
  {
  for(position = 0; position<strip_length; position = position+1)
    {
    SPI.transfer(0x80 | 0x00);            //Green - none
    SPI.transfer(0x80 | 0x00);            //Red   - none
    SPI.transfer(0x80 | LED_brightness);  //Blue  - MSB1

    if(troubleshooting)
      {
```

```
        Serial.println(LED_brightness, DEC);
        Serial.println(position, DEC);
        }
      delay(segment_delay);
      }
    SPI.transfer(LED_strip_latch);     //reset to first segment
    delay(strip_delay);
    if(troubleshooting)
      {
      Serial.println(" ");
      }
   }

  clear_strip();                      //all strip LEDs to black
  delay(500);
 }

//*****************************************************************

void clear_strip(void)
{
  //clear strip
  for(position = 0; position<strip_length; position = position+1)
      {
      SPI.transfer(0x80 | 0x00);            //Green - none
      SPI.transfer(0x80 | 0x00);            //Red - none
      SPI.transfer(0x80 | 0x00);            //Blue  - none

      if(troubleshooting)
        {
        Serial.println(LED_brightness, DEC);
        Serial.println(position, DEC);
        }
      }
  SPI.transfer(LED_strip_latch);            //Latch with zero
  if(troubleshooting)
    {
    Serial.println(" ");
    }
```

```
  delay(2000);                                  //clear delay
}

//****************************************************************
```

Example: Analog In-Analog Out-Serial Out. This example is modified from the example Analog In-Analog Out-Serial Out provided with Energia. It illustrates several Energia built-in functions.

- **Serial.begin(baud_rate):** Sets baud rate in bits per second to communicate with the host computer.
- **Serial.print(text):** Prints text to Energia serial monitor.
- **AnalogRead(analog_channel):** Reads the analog value at the designated analog channel and returns a value from 0 (0 VDC) to 1023 (3.3 VDC).
- **map(test_value, input_low, input_high, output_low, output_high):** Remaps test_value from a value between input_low and input_high to a corresponding value between output_low and output_high.
- **analogWrite(analogOutPin, outputValue):** Sends an output value from 0–255 to designated analogOutPin.

```
//****************************************************************
//Analog input, analog output, serial output - Reads an analog input pin,
//maps the result to a range from 0 to 255 and uses the result to set the
//pulsewidth modulation (PWM) of an output pin.  The PWM value is sent to
//the red LED pin to modulate its intensity.  Also prints the results to
//the serial monitor.  Open a serial monitor using the serial monitor
//button in Energia to view the results.
//
//
//The circuit:
//  - Potentiometer connected to analog pin 0 (2).  The center wiper
//    pin of the potentiometer goes to the analog pin.  The side pins of
//    the potentiometer go to +3.3 VDC and ground.  Place a small value
//    resistor is series with the potentiometer
//  - The analog output is designated as the onboard red LED.
//
//Created:  Dec 29, 2008
//Modified: Aug 30, 2011
//Author:   Tom Igoe
```

```
//This example code is in the public domain.
//****************************************************************

const int analogInPin  = 2;   //Energia analog input pin A0
const int analogOutPin = RED_LED;  //Energia onboard red LED pin

int sensorValue = 0;          //value read from potentiometer
int outputValue = 0;          //value output to the PWM (red LED)

void setup()
  {
  // initialize serial communications at 9600 bps:
  Serial.begin(9600);
  }

void loop()
  {
  //read the analog in value:
  sensorValue = analogRead(analogInPin);

  // map it to the range of the analog out:
  outputValue = map(sensorValue, 0, 1023, 0, 255);

  // change the analog out value:
  analogWrite(analogOutPin, outputValue);

  // print the results to the serial monitor:
  Serial.print("sensor = " );
  Serial.print(sensorValue);
  Serial.print("\t output = ");
  Serial.println(outputValue);

  // wait 10 milliseconds before the next loop
  // for the analog-to-digital converter to settle
  // after the last reading:
  delay(10);
}

//****************************************************************
```

Example: Mini round autonomous maze navigating robot. In this example, an autonomous, maze navigating robot is equipped with infrared (IR) sensors to detect the presence of maze walls and navigate about the maze. The robot has no prior knowledge about the maze configuration. It uses the IR sensors and an onboard algorithm to determine the robot's next move. The overall goal is to navigate from the starting point of the maze to the end point as quickly as possible without bumping into maze walls as shown in Figure 2.8. Maze walls are usually painted white to provide a good, light reflective surface, whereas the maze floor is painted matte black to minimize light reflections. Alternatively, a low-cost, table-top maze may be assembled from white foam board.

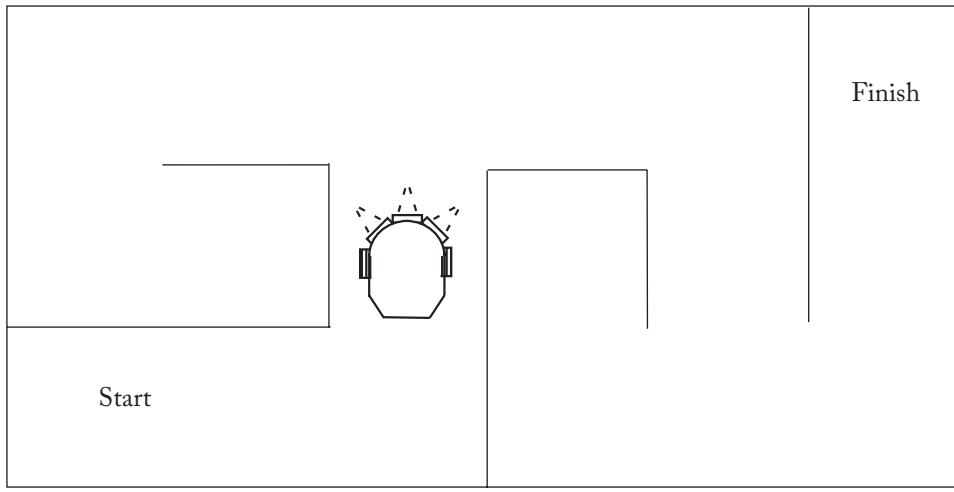
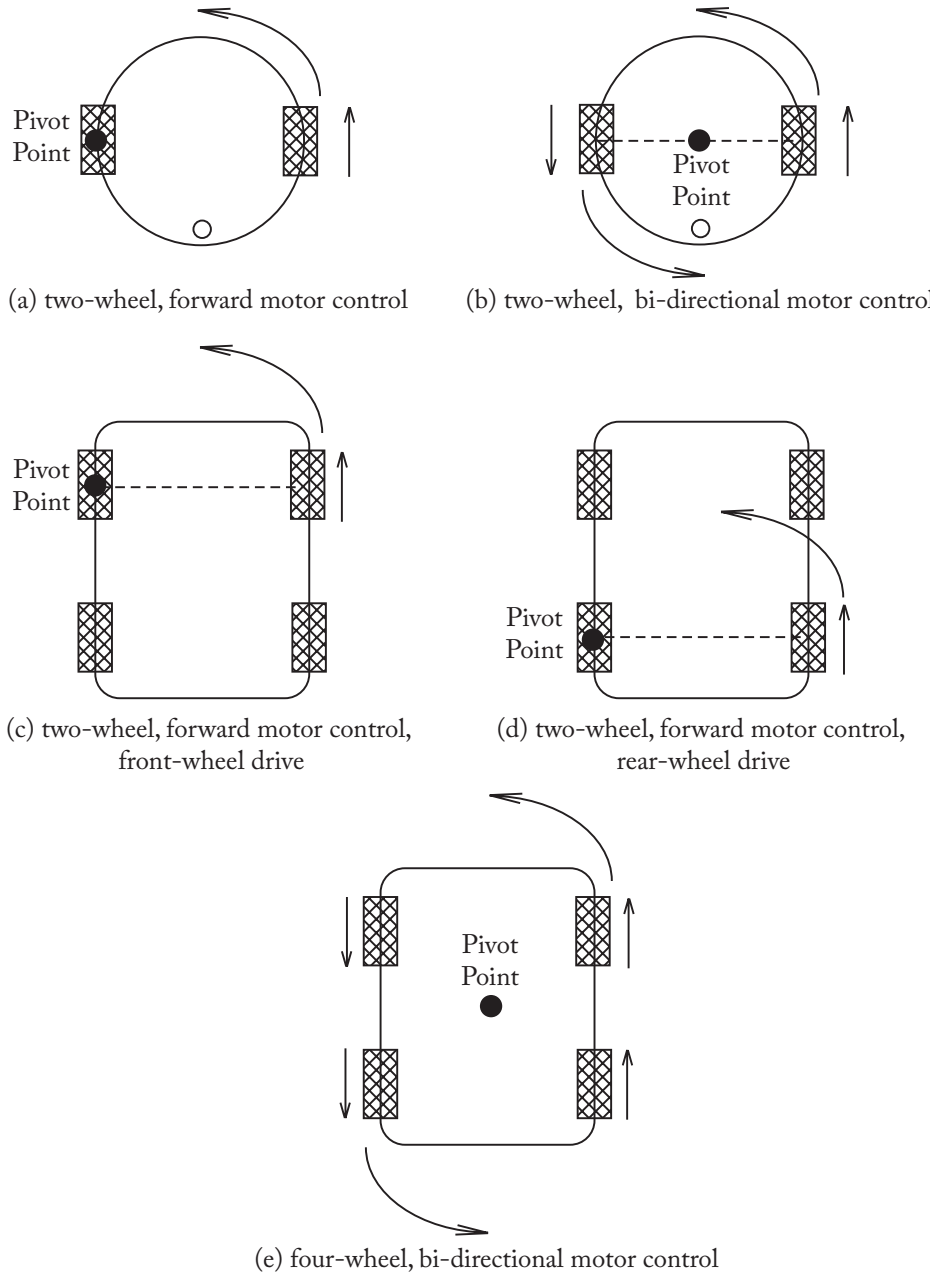


Figure 2.8: Autonomous robot within maze.

Before delving into the robot design, it would be helpful to review the fundamentals of robot steering and motor control. Figure 2.9 illustrates the fundamental concepts. Robot steering is dependent upon the number of powered wheels and whether the wheels are equipped with unidirectional or bidirectional control. Additional robot steering configurations are possible. An H-bridge is typically required for bidirectional control of a DC motor. We discuss the H-bridge in greater detail in an upcoming chapter.

In this application project, we equip the Adafruit mini round robot (#3216) for control by the LaunchPad as a maze navigating robot. Reference Figure 2.11. The robot is controlled by two 6.0 VDC motors which independently drive a left and right wheel. A third non-powered drag ball provides tripod stability for the robot.

We equip the mini round robot platform with three Sharp GP2Y0A21YKOF IR sensors as shown in Figure 2.12. The sensors are available from SparkFun Electronics (`www.sparkfun.com`). We mount the sensors on a bracket constructed from thin aluminum. Dimensions for the bracket are provided in the figure. Alternatively, the IR sensors may be mounted to the



(a) two-wheel, forward motor control

(b) two-wheel, bi-directional motor control

(c) two-wheel, forward motor control, front-wheel drive

(d) two-wheel, forward motor control, rear-wheel drive

(e) four-wheel, bi-directional motor control

Figure 2.9: Robot control configurations.

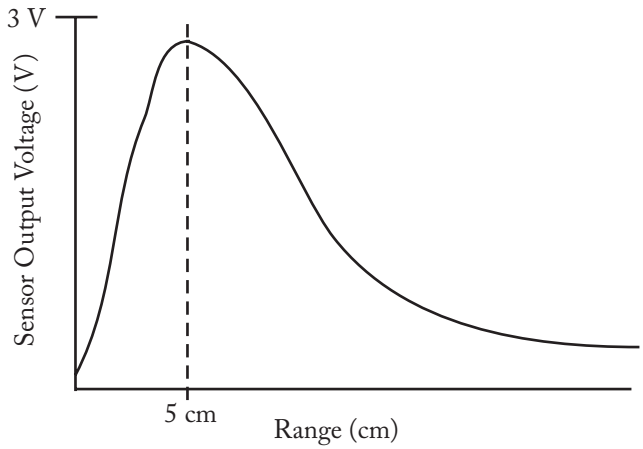


Figure 2.10: Sharp GP2Y0A21YKOF IR sensor profile.

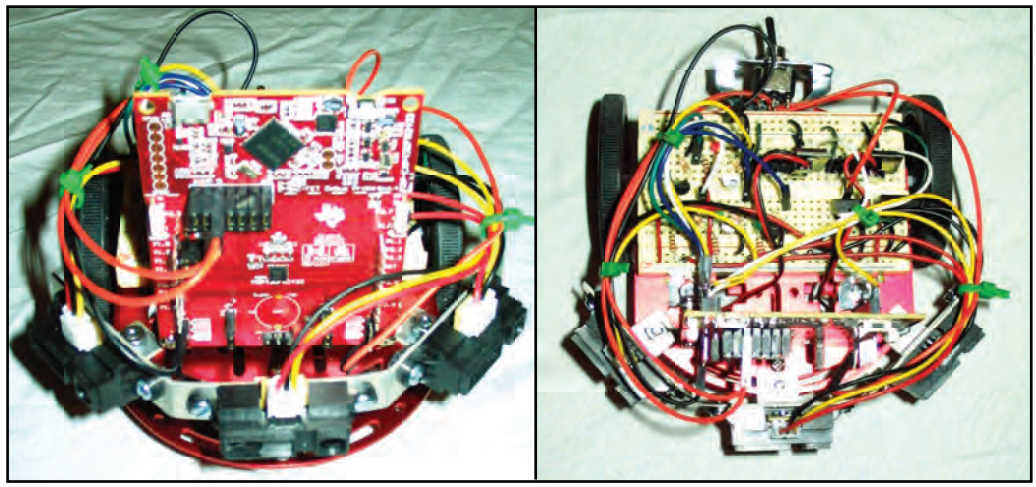


Figure 2.11: Mini round robot.

robot platform using "L" brackets available from a local hardware store. The characteristics of the sensor are provided in Figure 2.10.

The circuit diagram for the robot is provided in Figure 2.13. The three IR sensors (left, middle, and right) are mounted on the leading edge of the robot to detect maze walls. The output from the sensors is fed to three ADC channels (analog in 0-2). The robot motors will be driven by PWM channels (PWM: digital I/O 3 and PWM: digital I/O 4).

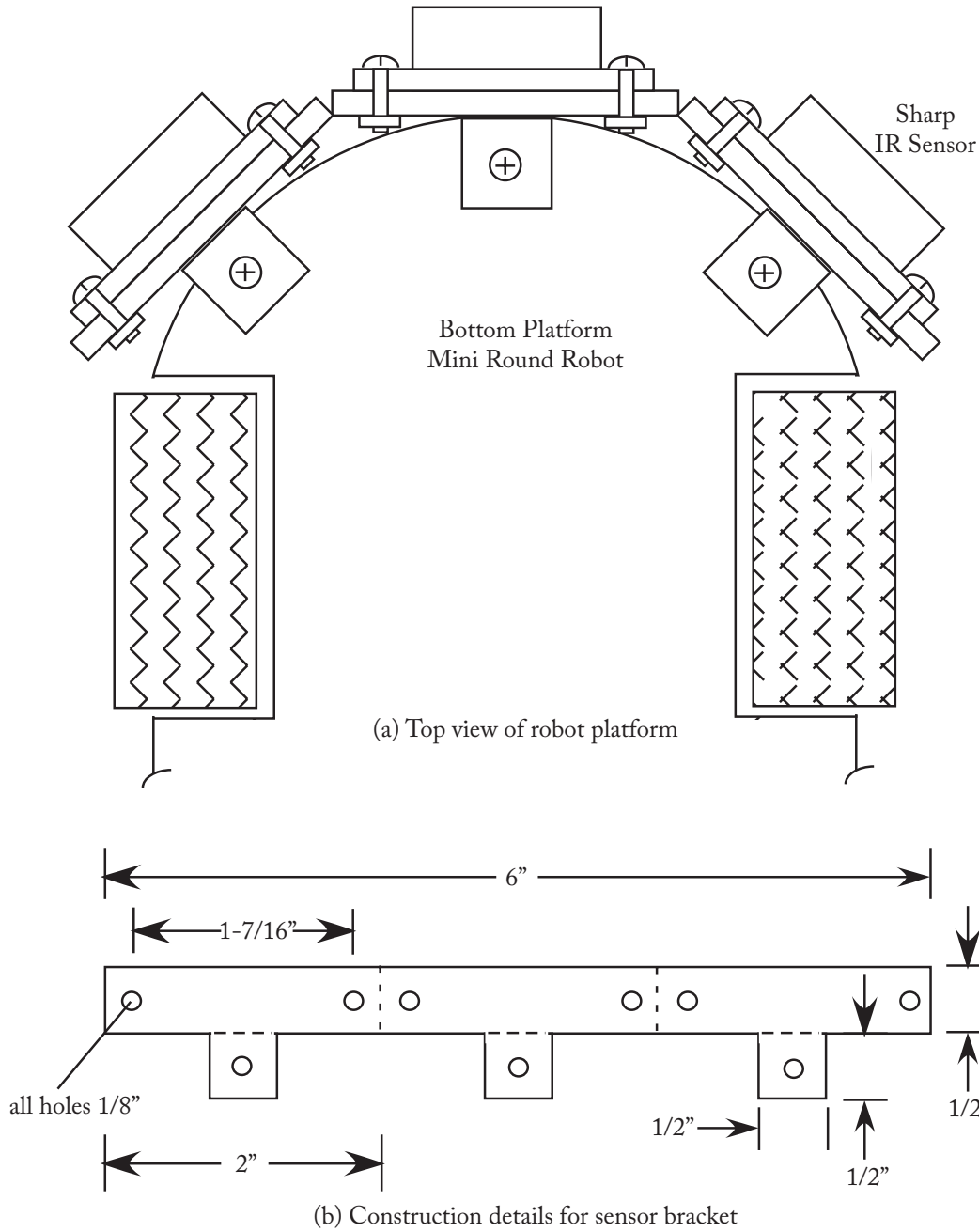


Figure 2.12: Mini round robot platform modified with three IR sensors.

To save on battery expense, a 9 VDC, 2A rated inexpensive, wall-mount power supply is used to provide power to the robot. A power umbilical of flexible, braided wire may be used to link the power supply to the robot while navigating about the maze. The robot motors are rated at 6.0 VDC. Therefore, four 1N4001 diodes are placed in series with the motor to reduce the supply voltage to be approximately 6.2 VDC. The LaunchPad is interfaced to the motors via a Darlington NPN transistor (TIP120) with enough drive capability to handle the maximum current requirements of the motor. A 3.3 VDC voltage regulator is used to supply power to the LaunchPad.

Warning: It is important **not** to have the LaunchPad connected to the host computer via the USB cable and an external 3.3 VDC supply at the same time. It is recommended to download the program to the LaunchPad, disconnect the USB cable, remove the 3.3 VDC header jumper on the Jumper Isolation Block, and then connect the 3.3 VDC external supply to the J6 connector. Alternatively, a double-throw double-pole (DPDT) switch may be used, as shown in Figure 2.13.

2.6.1 CONTROL ALGORITHM FOR THE MINI ROUND ROBOT

In this section, we provide the basic framework for the robot control algorithm. The control algorithm will read the IR sensors attached to the LaunchPad analog in (pins 0–2). In response to the wall placement detected, it will render signals to turn the robot to avoid the maze walls. Provided in Figure 2.14 is a truth table that shows all possibilities of maze placement that the robot might encounter. A detected wall is represented with a logic one. An asserted motor action is also represented with a logic one.

The robot motors may only be moved in the forward direction. We review techniques to provide bi-directional motor control in an upcoming chapter. To render a left turn, the left motor is stopped and the right motor is asserted until the robot completes the turn. To render a right turn, the opposite action is required.

The task of writing the control algorithm is to take the Unified Modeling Language (UML) activity diagram provided in Figure 3.14 and the actions specified in the robot action truth table (Figure 2.14) and transform both into an Energia sketch. This may seem formidable, but we take it a step at a time.

The control algorithm begins with Energia pin definitions. Variables are then declared for the readings from the three IR sensors. The two required Energia functions follow: setup() and loop(). In the setup() function, Energia pins are declared as output. The loop() begins by reading the current value of the three IR sensors.

The analogRead function reports a value between 0 and 1023. The 0 corresponds to 0 VDC while the value 1023 corresponds to 3.3 VDC. A specific value corresponds to a particular IR sensor range. The threshold detection value may be adjusted to change the range at which the maze wall is detected.

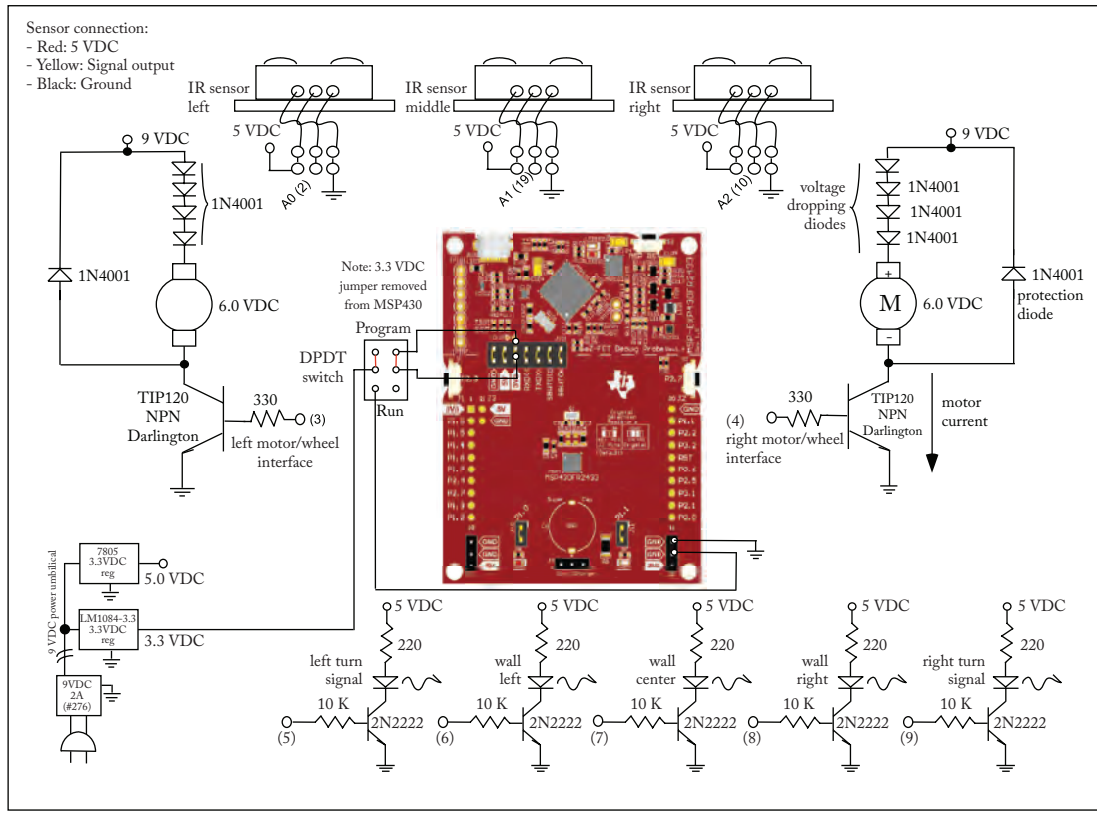


Figure 2.13: Robot circuit diagram. (Illustration used with permission of Texas Instruments (www.ti.com).)

	Left Sensor	Middle Sensor	Right Sensor	Wall Left	Wall Middle	Wall Right	Left Motor	Right Motor	Left Signal	Right Signal	Comments
0	0	0	0	0	0	0	1	1	0	0	Forward
1	0	0	1	0	0	1	1	1	0	0	Forward
2	0	1	0	0	1	0	1	0	0	1	Right
3	0	1	1	0	1	1	0	1	1	0	Left
4	1	0	0	1	0	0	1	1	0	0	Forward
5	1	0	1	1	0	1	1	1	0	0	Forward
6	1	1	0	1	1	0	1	0	0	1	Right
7	1	1	1	1	1	1	1	0	0	1	Right

Figure 2.14: Truth table for robot action.

The read of the IR sensors is followed by an eight-part if-else if statement. The statement contains a part for each row of the truth table provided in Figure 2.14. For a given configuration of sensed walls, the appropriate wall detection LEDs are illuminated followed by commands to activate the motors (analogWrite) and illuminate the appropriate turn signals.

The analogWrite command issues a signal from 0–3.3 VDC by sending a constant from 0–255 using PWM techniques. PWM techniques will be discussed in an upcoming chapter. The turn signal commands provide to actions: the appropriate turns signals are flashed and a 1.5 s total delay is provided. This provides the robot 1.5 s to render a turn. This delay may need to be adjusted during the testing phase.

```
//*********************************************************************
//robot
//
////This example code is in the public domain.
//*********************************************************************
                                    //analog input pins
#define left_IR_sensor    2         //analog pin - left IR sensor
#define center_IR_sensor  19        //analog pin - center IR sensor
#define right_IR_sensor   10        //analog pin - right IR sensor

                                    //digital output pins
                                    //LED indicators - wall detectors
#define wall_left         6         //digital pin - wall_left
#define wall_center       7         //digital pin - wall_center
#define wall_right        8         //digital pin - wall_right

                                    //LED indicators - turn signals
#define left_turn_signal  5         //digital pin - left_turn_signal
#define right_turn_signal 9         //digital pin - right_turn_signal

                                    //motor outputs
#define left_motor        3         //digital pin - left_motor
#define right_motor       4         //digital pin - right_motor


int left_IR_sensor_value;           //variable for left IR sensor
int center_IR_sensor_value;         //variable for center IR sensor
int right_IR_sensor_value;          //variable for right IR sensor
```

```
void setup()
  {
                                    //LED indicators - wall detectors
  pinMode(wall_left,   OUTPUT);     //configure pin for digital output
  pinMode(wall_center, OUTPUT);     //configure pin for digital output
  pinMode(wall_right,  OUTPUT);     //configure pin for digital output

                                    //LED indicators - turn signals
  pinMode(left_turn_signal,OUTPUT); //configure pin for digital output
  pinMode(right_turn_signal,OUTPUT); //configure pin for digital output

                                    //motor outputs - PWM
  pinMode(left_motor,  OUTPUT);     //configure pin for digital output
  pinMode(right_motor, OUTPUT);     //configure pin for digital output
  }

void loop()
  {
                                    //read analog output from IR sensors
   left_IR_sensor_value   = analogRead(left_IR_sensor);
   center_IR_sensor_value = analogRead(center_IR_sensor);
   right_IR_sensor_value  = analogRead(right_IR_sensor);

   //robot action table row 0 - robot forward
   if((left_IR_sensor_value < 512)&&(center_IR_sensor_value < 512)&&
      (right_IR_sensor_value < 512))
     {
                                          //wall detection LEDs
     digitalWrite(wall_left,   LOW);      //turn LED off
     digitalWrite(wall_center, LOW);      //turn LED off
     digitalWrite(wall_right,  LOW);      //turn LED off
                                          //motor control
     analogWrite(left_motor,  128);       //0(off)-255(full speed)
     analogWrite(right_motor, 128);       //0(off)-255(full speed)
                                          //turn signals
     digitalWrite(left_turn_signal,  LOW); //turn LED off
     digitalWrite(right_turn_signal, LOW); //turn LED off
     delay(500);                          //delay 500 ms
     digitalWrite(left_turn_signal,  LOW); //turn LED off
```

```
  digitalWrite(right_turn_signal, LOW);   //turn LED off
  delay(500);                             //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, LOW);   //turn LED off
  delay(500);                             //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, LOW);   //turn LED off
  analogWrite(left_motor, 0);             //turn motor off
  analogWrite(right_motor,0);             //turn motor off
  }

//robot action table row 1 - robot forward
else if((left_IR_sensor_value < 512)&&(center_IR_sensor_value < 512)&&
        (right_IR_sensor_value > 512))
  {
                                          //wall detection LEDs
  digitalWrite(wall_left,   LOW);         //turn LED off
  digitalWrite(wall_center, LOW);         //turn LED off
  digitalWrite(wall_right,  HIGH);        //turn LED on
                                          //motor control
  analogWrite(left_motor,  128);          //0(off)-255(full speed)
  analogWrite(right_motor, 128);          //0(off)-255(full speed)
                                          //turn signals
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, LOW);   //turn LED off
  delay(500);                             //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, LOW);   //turn LED off
  delay(500);                             //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, LOW);   //turn LED off
  delay(500);                             //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, LOW);   //turn LED off
  analogWrite(left_motor, 0);             //turn motor off
  analogWrite(right_motor,0);             //turn motor off
  }

//robot action table row 2 - robot right
```

```
else if((left_IR_sensor_value < 512)&&(center_IR_sensor_value > 512)&&
        (right_IR_sensor_value < 512))
  {
                                          //wall detection LEDs
  digitalWrite(wall_left,   LOW);         //turn LED off
  digitalWrite(wall_center, HIGH);        //turn LED on
  digitalWrite(wall_right,  LOW);         //turn LED off
                                          //motor control
  analogWrite(left_motor,  128);          //0(off)-255(full speed)
  analogWrite(right_motor, 0);            //0(off)-255 (full speed)
                                          //turn signals
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, HIGH);  //turn LED on
  delay(500);                             //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, LOW);   //turn LED off
  delay(500);                             //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, HIGH);  //turn LED on
  delay(500);                             //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, LOW);   //turn LED off
  analogWrite(left_motor, 0);             //turn motor off
  analogWrite(right_motor,0);             //turn motor off
  }

//robot action table row 3 - robot left
else if((left_IR_sensor_value < 512)&&(center_IR_sensor_value > 512)&&
        (right_IR_sensor_value > 512))
  {
                                          //wall detection LEDs
  digitalWrite(wall_left,   LOW);         //turn LED off
  digitalWrite(wall_center, HIGH);        //turn LED on
  digitalWrite(wall_right,  HIGH);        //turn LED on
                                          //motor control
  analogWrite(left_motor,  0);            //0(off)-255 (full speed)
  analogWrite(right_motor, 128);          //0(off)-255 (full speed)
                                          //turn signals
  digitalWrite(left_turn_signal,  HIGH);  //turn LED on
```

```
  digitalWrite(right_turn_signal, LOW);   //turn LED off
  delay(500);                             //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, LOW);   //turn LED off
  delay(500);                             //delay 500 ms
  digitalWrite(left_turn_signal,  HIGH);  //turn LED on
  digitalWrite(right_turn_signal, LOW);   //turn LED off
  delay(500);                             //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, LOW);   //turn LED off
  analogWrite(left_motor, 0);             //turn motor off
  analogWrite(right_motor,0);             //turn motor off
  }

//robot action table row 4 - robot forward
else if((left_IR_sensor_value > 512)&&(center_IR_sensor_value < 512)&&
        (right_IR_sensor_value < 512))
  {
                                          //wall detection LEDs
  digitalWrite(wall_left,   HIGH);        //turn LED on
  digitalWrite(wall_center, LOW);         //turn LED off
  digitalWrite(wall_right,  LOW);         //turn LED off
                                          //motor control
  analogWrite(left_motor,  128);          //0(off)-255 (full speed)
  analogWrite(right_motor, 128);          //0(off)-255 (full speed)
                                          //turn signals
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, LOW);   //turn LED off
  delay(500);                             //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, LOW);   //turn LED off
  delay(500);                             //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, LOW);   //turn LED off
  delay(500);                             //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, LOW);   //turn LED off
  analogWrite(left_motor, 0);             //turn motor off
  analogWrite(right_motor,0);             //turn motor off
```

```
  }

//robot action table row 5 - robot forward
else if((left_IR_sensor_value > 512)&&(center_IR_sensor_value < 512)&&
        (right_IR_sensor_value > 512))
  {
                                          //wall detection LEDs
  digitalWrite(wall_left,   HIGH);        //turn LED on
  digitalWrite(wall_center, LOW);         //turn LED off
  digitalWrite(wall_right,  HIGH);        //turn LED on
                                          //motor control
  analogWrite(left_motor,  128);          //0(off)-255 (full speed)
  analogWrite(right_motor, 128);          //0(off)-255 (full speed)
                                          //turn signals
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, LOW);   //turn LED off
  delay(500);                             //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, LOW);   //turn LED off
  delay(500);                             //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, LOW);   //turn LED off
  delay(500);                             //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, LOW);   //turn LED off
  analogWrite(left_motor, 0);             //turn motor off
  analogWrite(right_motor,0);             //turn motor off
  }

//robot action table row 6 - robot right
else if((left_IR_sensor_value > 512)&&(center_IR_sensor_value > 512)&&
        (right_IR_sensor_value < 512))
  {
                                          //wall detection LEDs
  digitalWrite(wall_left,   HIGH);        //turn LED on
  digitalWrite(wall_center, HIGH);        //turn LED on
  digitalWrite(wall_right,  LOW);         //turn LED off
                                          //motor control
  analogWrite(left_motor,  128);          //0(off)-255 (full speed)
```

```
  analogWrite(right_motor, 0);            //0(off)-255 (full speed)
                                          //turn signals
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, HIGH);  //turn LED on
  delay(500);                             //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, LOW);   //turn LED off
  delay(500);                             //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, HIGH);  //turn LED off
  delay(500);                             //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);   //turn LED OFF
  digitalWrite(right_turn_signal, LOW);   //turn LED OFF
  analogWrite(left_motor, 0);             //turn motor off
  analogWrite(right_motor,0);             //turn motor off
  }

//robot action table row 7 - robot right
else if((left_IR_sensor_value > 512)&&(center_IR_sensor_value > 512)&&
        (right_IR_sensor_value > 512))
  {
                                          //wall detection LEDs
  digitalWrite(wall_left,   HIGH);        //turn LED on
  digitalWrite(wall_center, HIGH);        //turn LED on
  digitalWrite(wall_right,  HIGH);        //turn LED on
                                          //motor control
  analogWrite(left_motor,  128);          //0(off)-255 (full speed)
  analogWrite(right_motor, 0);            //0(off)-255 (full speed)
                                          //turn signals
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, HIGH);  //turn LED on
  delay(500);                             //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, LOW);   //turn LED off
  delay(500);                             //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, HIGH);  //turn LED on
  delay(500);                             //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
```

```
      digitalWrite(right_turn_signal, LOW);   //turn LED off
      analogWrite(left_motor, 0);             //turn motor off
      analogWrite(right_motor,0);             //turn motor off
      }
}
//*********************************************************************
```

Testing the control algorithm: It is recommended that the algorithm be first tested without the entire robot platform. This may be accomplished by connecting the three IR sensors and LEDS to the appropriate pins on the LaunchPad as specified in Figure 2.13. In place of the two motors and their interface circuits, two LEDs with the required interface circuitry may be used. The LEDs illuminate to indicate motor assertion. Once the algorithm is fully tested in this fashion, the LaunchPad may be mounted to the robot platform and connected to the motors. Full up testing in the maze may commence. Enjoy!

2.7 SOME ADDITIONAL COMMENTS ON ENERGIA

Energia is based on the open source concept. Users throughout the world are constantly adding new built-in features. As new features are added, they will be released in future Energia IDE versions. As an Energia user, you too may add to this collection of useful tools. In the next section we investigate programming in C.

2.8 PROGRAMMING IN C

Most microcontrollers are programmed with some variant of the C programming language. The C programming language provides a nice balance between the programmer's control of the microcontroller hardware and time efficiency in programming writing.

As you can see in Figure 2.15, the compiler software is hosted on a computer separate from the LaunchPad. The job of the compiler is to transform the program provided by the program writer (filename.c and filename.h) into machine code suitable for loading into the processor.

Once the source files (filename.c and filename.h) are provided to the compiler, the compiler executes two steps to render the machine code. The first step is the compilation process. Here the program source files are transformed into assembly code (filename.asm). If the program source files contain syntax errors, the compiler reports these to the user. Syntax errors are reported for incorrect use of the C programming language. An assembly language program is not generated until the syntax errors have been corrected. The assembly language source file is then passed to the assembler. The assembler transforms the assembly language source file to machine code suitable for loading to the LaunchPad.

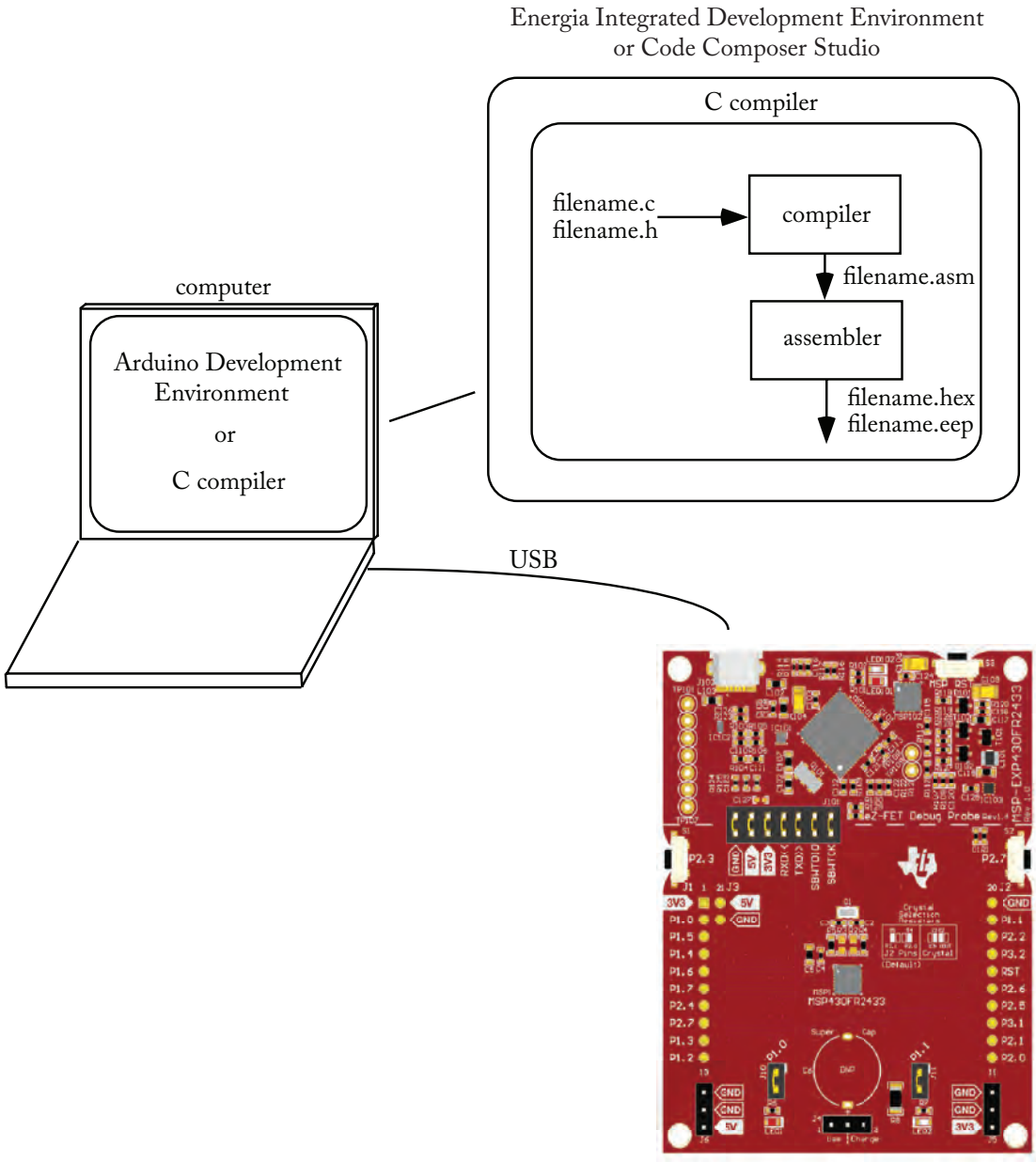


Figure 2.15: Programming the LaunchPad. (Illustration used with permission of Texas Instruments (www.ti.com).)

During the compilation process, warnings may also be generated. Warnings do not prevent the creation of an assembly language version of the C program. However, they should be resolved since flagged incorrect usage of the C language may result in unexpected program run time errors.

As seen earlier in the chapter, the Energia Integrated Development Environment provides a user-friendly interface to aid in program development, transformation to machine code, and loading into the LaunchPad. As described in Chapter 1, the LaunchPad may also be programmed using Code Composer Studio, Keil and IAR Systems software. We use Code Composer Studio throughout the book.

For the remaining portion of the chapter we present a brief introduction to C. Many examples are provided. We encourage the reader to modify, load, and run the examples on the LaunchPad.

In the next section, we will discuss the components of a C program.

2.9 ANATOMY OF A PROGRAM

Programs written for a microcontroller have a repeatable format. Slight variations exist but many follow the format provided.

```
//*************************************************************
//Comments containing program information
// - file name:
// - author:
// - revision history:
// - compiler setting information:
// - hardware connection description to microcontroller pins
// - program description
//*************************************************************

//include file(s)
#include<file_name.h>

//function prototypes
A list of functions and their format used within the program

//program constants
#define    TRUE   1
#define    FALSE  0
#define    ON     1
#define    OFF    0
```

```
//interrupt handler definitions
Used to link the software to hardware interrupt features

//global variables
Listing of variables used throughout the program

//main program

void main(void)
{

body of the main program

}

//************************************************************
//function definitions: A detailed function body and definition
//for each function used within the program.  For larger
//programs, function definitions may be placed in accompanying
//header files.
//************************************************************
```

Let's take a closer look at each part of the program.

2.9.1 COMMENTS

Comments are used throughout the program to document what and how things were accomplished within a program. The comments help you and others to reconstruct your work at a later time. Imagine that you wrote a program a year ago for a project. You now want to modify that program for a new project. The comments will help you remember the key details of the program.

Comments are not compiled into machine code for loading into the microcontroller. Therefore, the comments will not fill up the memory of your microcontroller. Comments are indicated using double slashes (//). Anything from the double slashes to the end of a line is then considered a comment. A multi-line comment is constructed using a /* at the beginning of the comment and a */ at the end of the comment. These are handy to block out portions of code during troubleshooting or providing multi-line comments.

At the beginning of the program, comments may be extensive. Comments may include some of the following information:

- file name,

- program author and dates of creation,
- revision history or a listing of the key changes made to the program,
- compiler setting information,
- hardware connection description to microcontroller pins, and
- program description.

2.9.2 INCLUDE FILES

Often you need to add extra files to your project besides the main program. For example, most compilers require a "personality file" on the specific microcontroller that you are using. This file is provided with the compiler and provides the name of each register used within the microcontroller. It also provides the link between a specific register's name within software and the actual register location within hardware. These files are typically called header files and their name ends with a ".h". In MSP430 applications, typically the "msp430.h" header files or the device specific header file (e.g., "msp430fr5994.h") will be used. Within the C compiler there will also be other header files to include in your program such as the "math.h" file when programming with advanced math functions.

To include header files within a program, the following syntax is used:

```
//C programming: include files
#include<file_name1.h>   //searches for file in a standard list
#include<file_name2.h>
#include "file_name3.h"  //searches for file in current directory
```

2.9.3 FUNCTIONS

Later in the book we discuss in detail the top-down design, bottom-up implementation approach to designing microcontroller-based systems. In this approach, a project including both hardware and software is partitioned into systems, subsystems, etc. The idea is to take a complex project and break it into smaller, doable pieces with a defined action.

We use the same approach when writing computer programs. At the highest level is the main program which calls functions that have a defined action. When a function is called, program control is released from the main program to the function. Once the function is complete, program control returns to the main program.

Functions may in turn call other functions as shown in Figure 2.16. This approach results in a collection of functions that may be reused in various projects. Most importantly, the program is now subdivided into doable pieces, each with a defined action. This makes writing the program easier but also makes it convenient to modify the program since every action is in a known location.

There are three different pieces of code required to properly configure and call a function:

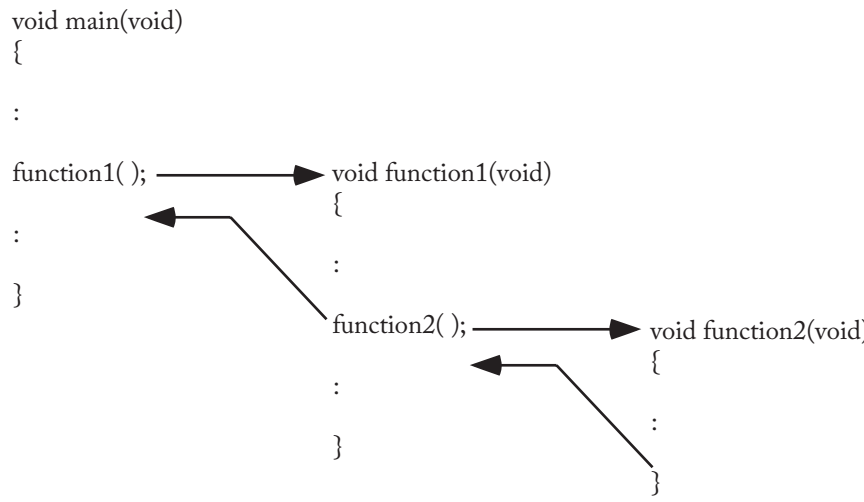


Figure 2.16: Function calling.

- function prototype,
- function call, and
- function body.

Function prototypes are provided early in the program as previously shown in the program template. The function prototype provides the name of the function and any variables required and returned by the function.

The function prototype follows this format:

```
return_variable  function_name(required_variable1, required_variable2);
```

If the function does not require variables or sends back a variable the word "void" is placed in the variable's position.

The **function call** is the code statement used within a program to execute the function. The function call consists of the function name and the actual arguments required by the function. If the function does not require arguments to be delivered to it for processing, the parenthesis containing the variable list is left empty.

The function call follows this format:

```
function_name(required_variable1, required_variable2);
```

A function that requires no variables is called by:

```
function_name( );
```

When the function call is executed by the program, program control is transferred to the function, the function is executed, and program control is then returned to the portion of the program that called it.

The **function body** is a self-contained "mini-program." The first line of the function body contains the same information as the function prototype: the name of the function, any variables required by the function, and any variable returned by the function. The last line of the function contains a "return" statement. Here a variable may be sent back to the portion of the program that called the function. The processing action of the function is contained within the open ({) and close brackets (}). If the function requires any variables within the confines of the function, they are declared next. These variables are referred to as local variables. A local variable is known only within the confines of a specific function. The actions required by the function follow.

The function prototype follows this format:

```
return_variable  function_name(required_variable1, required_variable2)
{
//local variables required by the function
unsigned int  variable1;
unsigned char variable2;

//program statements required by the function


//return variable
return return_variable;
}
```

2.9.4 PORT CONFIGURATION

The MSP430 FR2433 is equipped with a single 16-bit digital I/O port designated PA. This port may also be subdivided into two 8-bit ports designated P1 and P2. The FR2433 also has a 3-bit port designated PB. The FR5994 is equipped with a four 16-bit digital I/O ports designated PA through PD. These port may also be subdivided into two 8-bit ports. For example, port PA may be designated P1 and P2. The FR5994 also has an 8-bit port designated PJ.

Configuration and access to digital I/O pins are provided by a complement of registers. These registers include the following.

- **Input Registers (PxIN):** Allows input logic value of pin to be read (1: High, 0: Low).
- **Output Registers (PxOUT):** Value of output register is provided to corresponding output pin (1: High, 0: Low).
- **Direction Registers (PxDIR):** Bit in PxDIR selects corresponding digital I/O pin as output (1) or input(0).

- **Pull-up or Pull-down Resistor Enable Registers (PxREN):** Each bit determines if an internal pulled up (or pulled down) resistor is enabled at the corresponding pin. The value of the corresponding PxOUT register determines if pulled up (1) or pulled down (0) is selected. In summary, use the following PxDIR, PxREN, and PxOUT settings:
  - 00x: input
  - 010: input with pull-down resistor
  - 011: input with pull-up resistor
  - 1xx: output
- **Output Drive Strength Selection Registers (PxDS):** The value of the register determines the drive strength for specific pins (1: high drive strength, 0: regular drive strength).
- **Function Select Registers (PxSEL0, PxSEL1):** Allows specific function of multi-function pins to have access to I/O pin.

Throughout the book we use two approaches to configure MSP430 subsystems via their complement of control registers. The registers may be configured directly using C programming techniques (the "bare metal" approach) or via higher level application program interface (APIs). The MSP430 has an extensive complement of APIs for all MSP430 subsystems within the MSPWare library. Details on how to use the APIs are available through the Code Composer Studio Resource Manager in the following documents:

- FR2433: *MSP430 DriverLib for MSP430FR2xx 4xx Devices User's Guide*
- FR5994: *MSP430 DriverLib for MSP430FR5xx 6xx Devices User's Guide*

In the following examples, we revisit the blink LED example using the "bare metal" approach. Techniques are similar for the MSP-EXP430FR5994. **Note:** In this first example we provide the full Texas Instruments Incorporated copyright notification. In examples that follow throughout the book an abbreviated version will be provided.

```
//******************************************************************
// --COPYRIGHT--,BSD_EX
// Copyright (c) 2014, Texas Instruments Incorporated,
// All rights reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions
// are met:
// *  Redistributions of source code must retain the above copyright
//    notice, this list of conditions and the following disclaimer.
```

```
// *  Redistributions in binary form must reproduce the above copyright
//    notice, this list of conditions and the following disclaimer in the
//    documentation and/or other materials provided with the
//    distribution.
// *  Neither the name of Texas Instruments Incorporated nor the names of
//    its contributors may be used to endorse or promote products derived
//    from this software without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
// "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
// LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
// A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
// OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
// SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
// LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
// DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
// THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
// (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
// OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
//
//*****************************************************************************
//                          MSP430 CODE EXAMPLE DISCLAIMER
// MSP430 code examples are self-contained low-level programs that
// typically
// demonstrate a single peripheral function or device feature in a highly
// concise manner. For this the code may rely on the device's power-on
// default register values and settings such as the clock configuration
// and care must be taken when combining code from several examples to
// avoid potential side effects. Also see www.ti.com/grace for a GUI- and
// www.ti.com/msp430ware
// for an API functional library-approach to peripheral configuration.
//
// --/COPYRIGHT/--
//*****************************************************************************
//MSP430FR243x Demo - Toggle P1.0 using software
//
//Description: Toggle P1.0 every 0.1s using software.
//By default, FR24xx select XT1 as FLL reference.
//If XT1 is present, the PxSEL(XIN & XOUT) needs to configure.
```

```
//If XT1 is absent, switch to select REFO as FLL reference automatically.
//XT1 is considered to be absent in this example.
//ACLK = default REFO ~32768Hz, MCLK = SMCLK = default DCODIV ~1MHz.
//
//           MSP430FR2433
//         ---------------
//     /|\|               |
//      | |               |
//      --|RST            |
//        |           P1.0|-->LED
//
//Cen Fang
//Texas Instruments Inc.
//Feb 2015
//Built with IAR Embedded Workbench v6.20 & Code Composer Studio v6.0.1
//******************************************************************************

#include <msp430.h>

int main(void)
{
WDTCTL = WDTPW | WDTHOLD;               //Stop watchdog timer

P1OUT &= ~BIT0;                         //Clear P1.0 output latch for a
                                        //defined power-on state
P1DIR |= BIT0;                          //Set P1.0 to output direction

PM5CTL0 &= ~LOCKLPM5;                   //Disable the GPIO power-on
                                        //default high-impedance mode
                                        //to activate previously
                                        //configured port settings
while(1)
  {
  P1OUT ^= BIT0;                        //Toggle P1.0 using exclusive-OR
  __delay_cycles(100000);               //Delay for 100000*(1/MCLK)=0.1s
  }
}

//******************************************************************************
```

The code example begins with the inclusion of the msp430.h header file. The Watchdog timer that is normally on is then turned off. This is typically done during code development. The next three statements are a shorthand method of accomplishing the following:

```
P1OUT = P1OUT & ~BIT0;                  //Clear P1.0 output latch for a
                                        //defined power-on state
P1DIR = P1DIR |  BIT0;                  //Set P1.0 to output direction

PM5CTL0 = PM5CTL0 & ~LOCKLPM5;          //Disable the GPIO power-on
                                        //default high-impedance mode
                                        //to activate previously
                                        //configured port settings
```

BIT0 is defined in the MSP430.h header files as 0x0001. The P1OUT register is used to set PORT P1, pin 0 initially to logic 0. The P1DIR register is used to configure PORT P1, pin 0 to output. The Power Mode 5 Control Register 0 is then used to disable the general-purpose I/O power-on default configuration. LOCKLPM5 is defined as 0x0001 in the MSP430.h header file. The program then enters an infinite loop where PORT 1, pin 0 is toggled every 0.1 s.

2.9.5 PROGRAM CONSTANTS

The `#define` statement is used to associate a constant name with a numerical value in a program. It can be used to define common constants such as pi. It may also be used to give terms used within a program a numerical value. This makes the code easier to read. For example, the following constants may be defined within a program:

```
//program constants
#define    TRUE   1
#define    FALSE  0
#define    ON     1
#define    OFF    0
```

2.9.6 INTERRUPT HANDLER DEFINITIONS

Interrupts are functions that are written by the programmer but usually called by a specific hardware event during system operation. We discuss interrupts and how to properly configure them in an upcoming chapter.

2.9.7 VARIABLES

There are two types of variables used within a program: global variables and local variables. A global variable is available and accessible to all portions of the program, whereas a local variable is only known and accessible within the function where it is declared.

When declaring a variable in C, the number of bits used to store the variable is also specified. Variable specifications may vary by compiler. For code portability among different platforms fixed formats may be used.

Type	Size	Range
unsigned char	1	0..255
signed char	1	-128..127
unsigned int	2	0..65535
signed int	2	-32768..32767
float	4	+/-1.175e-38.. +/-3.40e+38
double	4 - 8	compiler dependent

Figure 2.17: C variable sizes.

Fixed format variables are defined within the "stdint.h" header file [stdint.h]. Provided below is a small extract from this header file.

```
//*****************************************************************

 typedef   signed char           int8_t;
 typedef unsigned char          uint8_t;
 typedef          int           int16_t;
 typedef unsigned int          uint16_t;
 typedef          long          int32_t;
 typedef unsigned long         uint32_t;
 typedef          long long     int64_t;
 typedef unsigned long long    uint64_t;

//*****************************************************************
```

When programming microcontrollers, it is important to know the number of bits and the memory location used to store the variable. For example, assigning the contents of an unsigned char variable, which is stored in 8-bits, to an 8-bit output port will have a predictable result.

However, assigning an unsigned int variable, which is stored in 16-bits, to an 8-bit output port does not provide predictable results. It is wise to ensure your assignment statements are balanced for accurate and predictable results. The modifier "unsigned" indicates all bits will be used to specify the magnitude of the argument. Signed variables will use the left most bit to indicate the polarity (±) of the argument.

Variables may be read (scanned) into a program using the "scanf" statement. The general format of the scanf statement is provided below. The format of the variable and the variable name are specified. Similarly, the variables may be printed using the "printf" statement. The backslash n specifies start a new line.

```
//*********************************************************************

#include<stdio.h>

int main( )
{
int input_variable;

scanf("

printf("

}

//*********************************************************************
```

A global variable is declared using the following format provided below. The type of the variable is specified, followed by its name, and an initial value if desired.

```
//*********************************************************************

//global variables
unsigned int  loop_iterations = 6;

//*********************************************************************
```

2.9.8 MAIN PROGRAM

The main program is the hub of activity for the entire program. The main program typically consists of program steps and function calls to initialize the processor followed by program steps to collect data from the environment external to the microcontroller, process the data and

make decisions, and provide external control signals back to the environment based on the data collected.

2.10 FUNDAMENTAL PROGRAMMING CONCEPTS

In the previous section, we covered many fundamental concepts. In this section we discuss operators, programming constructs, and decision processing constructs to complete our fundamental overview of programming concepts.

2.10.1 OPERATORS

There are a wide variety of operators provided in the C language. An abbreviated list of common operators is provided in Figures 2.18 and 2.19. The operators have been grouped by general category. The symbol, precedence, and brief description of each operator are provided. The precedence column indicates the priority of the operator in a program statement containing multiple operators. Only the fundamental operators are provided.

General Operations

Within the general operations category are brackets, parenthesis, and the assignment operator. We have seen in an earlier example how bracket pairs are used to indicate the beginning and end of the main program or a function. They are also used to group statements in programming constructs and decision processing constructs. This is discussed in the next several sections.

The parenthesis is used to boost the priority of an operator. For example, in the mathematical expression $7 \times 3 + 10$, the multiplication operation is performed before the addition since it has a higher precedence. Parenthesis may be used to boost the precedence of the addition operation. If we contain the addition operation within parenthesis $7 \times (3 + 10)$, the addition will be performed before the multiplication operation and yield a different result from the earlier expression.

The assignment operator (=) is used to assign the argument(s) on the right-hand side of an equation to the left-hand side variable. It is important to insure that the left- and the right--hand side of the equation have the same type of arguments. If not, unpredictable results may occur.

Arithmetic Operations

The arithmetic operations provide for basic math operations using the various variables described in the previous section. As described in the previous section, the assignment operator (=) is used to assign the argument(s) on the right-hand side of an equation to the left-hand side variable.

In this example, a function returns the sum of two unsigned int variables passed to the function.

General		
Symbol	Precedence	Description
{ }	1	Brackets, used to group program statements
( )	1	Parenthesis, used to establish precedence
=	12	Assignment

Arithmetic Operations		
Symbol	Precedence	Description
*	3	Multiplication
/	3	Division
+	4	Addition
-	4	Subtraction

Logical Operations		
Symbol	Precedence	Description
<	6	Less than
<=	6	Less than or equal to
>	6	Greater
>=	6	Greater than or equal to
= =	7	Equal to
!=	7	Not equal to
&&	9	Logical AND
||	10	Logical OR

Figure 2.18: C operators (adapted from Barrett and Pack [2005]).

Bit Manipulation Operations		
Symbol	Precedence	Description
<<	5	Shift left
>>	5	Shift right
&	8	Bitwise AND
^	8	Bitwise exclusive OR
|	8	Bitwise OR

Unary Operations		
Symbol	Precedence	Description
!	2	Unary negative
~	2	One's complement (bit-by-bit inversion)
++	2	Increment
__	2	Decrement
Type(argument)	2	Casting operator (data type conversion)

Figure 2.19: C operators (continued) (adapted from Barrett and Pack [2005]).

```
//*********************************************************************

unsigned int  sum_two(unsigned int variable1, unsigned int variable2)
{
unsigned int  sum;

sum = variable1 + variable2;

return sum;
}

//*********************************************************************
```

Logical Operations

The logical operators provide Boolean logic operations. They can be viewed as comparison operators. One argument is compared against another using the logical operator provided. The result is returned as a logic value of one (1, true, high) or zero (0 false, low). The logical operators are

used extensively in program constructs and decision processing operations to be discussed in the next several sections.

Bit Manipulation Operations

There are two general types of operations in the bit manipulation category: shifting operations and bitwise operations. Let's examine several examples.

Given the following code segment, what will the value of variable2 be after execution?

```
//*****************************************************************

unsigned char   variable1 = 0x73;
unsigned char   variable2;

variable2 = variable1 << 2;

//*****************************************************************
```

Answer: Variable "variable1" is declared as an 8-bit unsigned char and assigned the hexadecimal value of $(73)_{16}$. In binary, this is $(0111\_0011)_2$. The << 2 operator provides a left shift of the argument by two places. After two left shifts of $(73)_{16}$, the result is $(cc)_{16}$ and will be assigned to the variable "variable2." Note that the left and right shift operation is equivalent to multiplying and dividing the variable by a power of two.

The bitwise operators perform the desired operation on a bit-by-bit basis. That is, the least significant bit (LSB) of the first argument is bit-wise operated with the LSB of the second argument and so on.

Given the following code segment, what will the value of variable3 be after execution?

```
//*****************************************************************

unsigned char   variable1 = 0x73;
unsigned char   variable2 = 0xfa;
unsigned char   variable3;

variable3 = variable1 & variable2;

//*****************************************************************
```

Answer: Variable "variable1" is declared as an eight bit unsigned char and assigned the hexadecimal value of $(73)_{16}$. In binary, this is $(0111\_0011)_2$. Variable "variable2" is declared as an 8-bit unsigned char and assigned the hexadecimal value of $(fa)_{16}$. In binary, this is $(1111\_1010)_2$. The bitwise AND operator is specified. After execution variable "variable3," declared as an 8-bit unsigned char, contains the hexadecimal value of $(72)_{16}$.

Unary Operations

The unary operators, as their name implies, require only a single argument. For example, in the following code segment, the value of the variable "i" is incremented. This is a shorthand method of executing the operation "$i = i + 1$;"

```
//*********************************************************************
unsigned int    i;

i++;

//*********************************************************************
```

Bit Twiddling

It is not uncommon in embedded system design projects to have every pin on a microcontroller employed. Furthermore, it is not uncommon to have multiple inputs and outputs assigned to the same port but on different port I/O pins. Some compilers support specific pin reference. Another technique that is not compiler specific is **bit twiddling**. Figure 2.20 provides bit twiddling examples on how individual bits may be manipulated without affecting other bits using bitwise and unary operators. The information provided here was extracted from the ImageCraft ICC AVR compiler documentation [ImageCraft].

Syntax	Description	Example
a |b	bitwise or	P2OUT|= 0x80; // turn on bit 7 (msb)
a &b	bitwise and	if ((P2IN &0x81) == 0) // check bit 7 and bit 0
a ^ b	bitwise exclusive or	P2OUT^= 0x80; // flip bit 7
~a	bitwise complement	P2OUT&= ~0x80; // turn off bit 7

Figure 2.20: Bit twiddling [ImageCraft].

2.10.2 PROGRAMMING CONSTRUCTS

In this section, we discuss several methods of looping through a piece of code. We will examine the "for" and the "while" looping constructs.

The **for** loop provides a mechanism for looping through the same portion of code a fixed number of times. The for loop consists of three main parts:

- loop initiation,
- loop termination testing, and
- the loop increment.

In the following code fragment the for loop is executed ten times.

```
//*********************************************************************

unsigned int  loop_ctr;

for(loop_ctr = 0; loop_ctr < 10; loop_ctr++)
  {
                          //loop body

  }

//*********************************************************************
```

The for loop begins with the variable "loop_ctr" equal to 0. During the first pass through the loop, the variable retains this value. During the next pass through the loop, the variable "loop_ctr" is incremented by one. This action continues until the "loop_ctr" variable reaches the value of ten. Since the argument to continue the loop is no longer true, program execution continues with the next instruction after the close bracket of the for loop.

In the previous example, the for loop counter was incremented by one. The "loop_ctr" variable can be updated by any amount. For example, in the following code fragment the "loop_ctr" variable is increased by three for every pass of the loop.

```
//*********************************************************************

unsigned int  loop_ctr;

for(loop_ctr = 0; loop_ctr < 10; loop_ctr=loop_ctr+3)
  {
                          //loop body

  }

//*********************************************************************
```

The "loop_ctr" variable may also be initialized at a high value and then decremented at the beginning of each pass of the loop as shown below.

```
//*********************************************************************

unsigned int  loop_ctr;

for(loop_ctr = 10; loop_ctr > 0; loop_ctr--)
  {
                          //loop body

  }

//*********************************************************************
```

As before, the "loop_ctr" variable may be decreased by any numerical value as appropriate for the application at hand.

The **while** loop is another programming construct that allows multiple passes through a portion of code. The while loop will continue to execute the statements within the open and close brackets while the condition at the beginning of the loop remains logically true. The code snapshot below will implement a ten iteration loop. Note how the "loop_ctr" variable is initialized outside of the loop and incremented within the body of the loop. As before, the variable may be initialized to a greater value and then decremented within the loop body.

```
//*********************************************************************

unsigned int  loop_ctr;

loop_ctr = 0;
while(loop_ctr < 10)
  {
                          //loop body
  loop_ctr++;
  }

//*********************************************************************
```

Frequently, within a microcontroller application, the program begins with system initialization actions. Once initialization activities are completed, the processor enters a continuous loop. This may be accomplished using the following code fragment.

```
//*******************************************************************

while(1)
  {


  }

//*******************************************************************
```

2.10.3 DECISION PROCESSING

There are a variety of constructs that allow decision making. These include the following:

- the **if** statement,
- the **if-else** construct,
- the **if-else if-else** construct, and
- the **switch** statement.

The **if** statement will execute the code between an open and close bracket set, should the condition within the if statement be logically true. The **if-else** statement will execute the code between an open and close bracket set, should the condition within the if statement be logically true. If the statement is not true, the code after the else is executed.

Example: In this example switch S1 is connected to P1.3 (S1 on the MSP-EXP430FR2433 LaunchPad). If the switch is at logic one (switch not pressed), the red LED at pin P1.0 is illuminated. If the switch is pressed taking P1.3 to logic low, the LED goes out. In the example, pay close attention to how P1.0 is configured for output and P1.3 is configured for input with the corresponding pull-up resistor activated using bit-twiddling techniques.

```
//*******************************************************************
//Copyright (c) 2014, Texas Instruments Incorporated,
//All rights reserved.
//Reference full copyright statement at first coding example and
// MSP430 CODE EXAMPLE DISCLAIMER
//*******************************************************************
//*******************************************************************
//MSP430FR243x Demo - Software Poll P1.3, Set P1.0 if P1.3 = 1
//
//Description:
//Poll P1.3 in a loop. Set P1.0 if P1.3 = 1, or reset P1.0. By default,
```

```
//FR243x select XT1 as FLL reference. If XT1 is present, the XIN and XOUT
//pin needs to be configured.  If XT1 is absent, REFO is automatically
//switched for FLL reference.  XT1 is considered to be absent.
//ACLK = default REFO ~32768Hz, MCLK = SMCLK = default DCODIV ~1MHz.
//
//              MSP430FR2433
//            ---------------
//        /|\|               |
//         | |               |
//         --|RST            |
//     /|\   |               |
//      --o--|P1.3       P1.0|-->LED
//     \|/   |               |
//           |               |
//
//Cen Fang
//Texas Instruments Inc.
//June 2013
//Built with IAR Embedded Workbench v6.20 & Code Composer Studio v6.0.1
//******************************************************************************

#include <msp430.h>

void main(void)
{
WDTCTL = WDTPW | WDTHOLD;               //Stop watchdog timer

P1OUT &= ~BIT0;                         //Clear P1.0 output latch for a
                                        //defined power-on state
P1DIR |= BIT0;                          //Set P1.0 to output direction
P1DIR &= ~BIT3;                         //Set P1.3 as input

PM5CTL0 &= ~LOCKLPM5;                   //Disable the GPIO power-on
                                        //default high-impedance mode
                                        //to activate previously
                                        //configured port settings

while(1)                                //Test P1.3
  {
```

```
  if(P1IN & BIT3)
     P1OUT |= BIT0;                    //if P1.3 set, set P1.0
  else
     P1OUT &= ~BIT0;                   //else reset
  }
}

//*******************************************************************
```

The **switch** statement is used when multiple if-else conditions exist. Each possible condition is specified by a case statement. When a match is found between the switch variable and a specific case entry, the statements associated with the case are executed until a **break** statement is encountered. When a case match is not found, the default case is executed. Provided below is a template for the switch statement.

```
//*******************************************************************

switch (switch_variable)
{
case 1: code steps associated with case 1;
        :
        :
        break;


case 2: code steps associated with case 2;
        :
        :
        break;

:
:
:

case n: code steps associated with case n;
        :
        :
        break;

default:code steps associated with default;
```

```
}

//*********************************************************************
```

An example using the switch statement is provided later in the text. That completes our brief overview of Energia and the C programming language.

2.11 LABORATORY EXERCISE: GETTING ACQUAINTED WITH ENERGIA AND C

Introduction. In this laboratory exercise, you will become familiar with Energia and the C programming language through a variety of programming exercises.

If you have not done so already, download Code Composer Studio and also execute the blink program provided in *Code Composer Studio v7.x for MSP430 User's Guide* [SLAU157AP, 2017]. Provided here are steps to compile and execute a program using CCS from [SLAU157AP, 2017].

- Launch Code Composer Studio by clicking Start->All Programs-> Texas Instruments->Code Composer Studio->Code Composer Studio or by double clicking the CCS icon.
- Create a new project by clicking File->New->CCS Project and entering a project name.
- Set the Device Family to MSP430 and select the device variant to use (for example, MSP430FR2433).
- Select "Blink The LED" in the "Project templates and example" section and then click Finish.
- Compile the code and download the application to the target device by clicking Run->Debug (F11).
- To run the application, click Run->Resume (F8) or click the Play button on the toolbar.

Procedure 1: Energia.

1. Create a counter that counts continuously from 1–100 and repeats with a 50 ms delay between counts. The onboard red LED should illuminate for odd numbers and the onboard green LED for even numbers.
2. Modify the code produced for the previous step to illuminate the green LED when the number is evenly divisible by three and the red LED for the other numbers.

Procedure 2: C.

1. Develop a program that prompts the user for two integer numbers. If the first number is less than the second, the program should count-up continuously from the lower to the higher number with a 50 ms delay between counts. The onboard red LED should illuminate for odd numbers and the onboard green LED for even numbers. If the first number is higher than the second, the program should count down continuously from the lower to the higher number with a 50 ms delay between counts. The onboard red LED should illuminate for odd numbers and the onboard green LED for even numbers.

2. Develop a program that prompts the user for an integer number. If the number is evenly divisible by 2 the red LED illuminates, evenly divisible by three the green LED, and for other numbers no LEDs illuminate. Note: More than one LED may illuminate depending on the number provided.

2.12 SUMMARY

The goal of this chapter was to provide a tutorial on how to begin programming. We began with a discussion on the Energia Development Environment and how it may be used to develop a program for the MSP430 LaunchPads. For C, we used a top-down design approach. We began with the "big picture" of the program of interest followed by an overview of the major pieces of the program. We then discussed the basics of the C programming language. Only the most fundamental concepts were covered. Throughout the chapter, we provided examples and several excellent references.

2.13 REFERENCES AND FURTHER READING

Adafruit. `www.adafruit.com`

Arduino homepage. `www.arduino.cc`

Barrett, S. F. *Embedded Systems Design with the Atmel AVR Microcontroller*, Morgan & Claypool Publishers, San Rafael, CA, 2010. DOI: 10.2200/s00225ed1v01y200910dcs025.

Barrett, S. F. and Pack, D. J. *Atmel AVR Microcontroller Primer Programming and Interfacing*, Morgan & Claypool Publishers, San Rafael, CA, 2008. DOI: 10.2200/s00100ed1v01y200712dcs015.

Barrett, S. F. and Pack, D. J. *Embedded Systems Design and Applications with the 68HC12 and HCS12*, Pearson Prentice Hall, 2005. 67, 68

Barrett, S. F. and Pack, D. J. *Microcontrollers Fundamentals for Engineers and Scientists*, Morgan & Claypool Publishers, San Rafael, CA, 2006. DOI: 10.2200/s00025ed1v01y200605dcs001.

Code Composer Studio v7.x for MSP430 User's Guide, (SLAU157AP), Texas Instruments, 2017. 76

Energia. www.energia.nu

ImageCraft Embedded Systems C Development Tools, Palo Alto, CA. www.imagecraft.com 70

2.14 CHAPTER PROBLEMS

Fundamental

1. Describe the steps in writing a sketch and executing it on an MSP430 LaunchPads processing board.
2. Describe the basic components of any C program.
3. Describe two different methods to program an MSP430 LaunchPad.
4. What is an include file?
5. What are the three pieces of code required for a function?
6. Describe how a program constant is defined in C.
7. What is the difference between a for loop and a while loop?
8. When should a switch statement be used vs. the if-then statement construct?
9. What is the serial monitor feature used for in the Energia Development Environment?
10. Describe what variables are required and returned and the basic function of the following built-in Energia functions: Blink, Analog Input.

Advanced

1. Provide the C program statement to set PORT 1 pins 1 and 7 to logic one. Use bit-twiddling techniques.
2. Provide the C program statement to reset PORT 1 pins 1 and 7 to logic zero. Use bit-twiddling techniques.
3. Using a MSP430 LaunchPad, write a program in Energia that takes an integer input from the user. If negative, the red LED is illuminated. If positive, the green LED is illuminated.
4. Repeat the program above using C.

Challenging

1. Create a counter that counts continuously from 1–100 an repeats with a 50 ms delay between counts. The onboard red LED should illuminate for odd numbers and the onboard green LED for even numbers.

2. Develop a program that prompts the user for two integer numbers. If the first number is less than the second, the program should count-up continuously from the lower to the higher number with a 50 ms delay between counts. The onboard red LED should illuminate for odd numbers and the onboard green LED for even numbers. If the first number is greater than the second, the program should count down continuously from the lower to the higher number with a 50 ms delay between counts. The onboard red LED should illuminate for odd numbers and the onboard green LED for even numbers.

CHAPTER 3

Hardware Organization and Software Programming

Objectives: After reading this chapter, the reader should be able to:

- explain the organization of MSP430 hardware functional units;
- use controller software development tools;
- describe available MSP430 operating modes of the MSP430;
- identify and use appropriate assembly instructions;
- explain advantages and disadvantages of using a high-level programming language and the MSP430 assembly programming language;
- program the MSP430 controller for simple applications using both C and assembly language programs; and
- debug programs using joint test action group (JTAG) tools.

In this chapter, we present fundamental materials to understand the software and hardware systems of the MSP430 microcontroller. Mastering the contents of this chapter is critical in proceeding with the contents in the rest of this book, as they will serve as foundational MSP430 knowledge. Before getting into the specific hardware and software environments of the MSP430 microcontroller, the overall hardware and software organizations of a typical RISC (Reduced Instruction Set Computer) based microcontroller is presented, followed by the MSP430 specific hardware and software environments. The objectives of this chapter are to provide readers with a solid understanding of basic hardware and software concepts required to design, develop, and evaluate embedded systems using the MSP430 microcontroller.

3.1 MOTIVATION

The MSP430 microcontroller was designed for applications requiring ULP consumption, typically found in portable, battery-operated embedded systems applications. The MSP430 is currently used in a wide range of products, including medical diagnostic equipment, smart power

systems, security and fire monitoring sensors, alarm systems, power tools, and a host of consumer products. It is expected that the MSP430 will continue to grow its market share in smart homes and personal health monitoring applications.

3.2 MSP430 HARDWARE ORGANIZATION/ARCHITECTURE

In a RISC based machine, designers attempt to optimize the performance of a computer by simplifying hardware and providing a limited number of basic building block instructions. By reducing the time to execute each RISC instruction, even if you may require more instructions to execute a specific function, the goal of an RISC-based controller attempts to reduce the overall execution time of a software program. MSP430 is a good example of an RISC-based machine.

In this section, we present the hardware components and their organization within the MSP430 microcontroller. Once the MSP430 microcontroller hardware is presented, we delve into additional details of the MSP430FR2433 and MSP430FR5994 controllers.

3.2.1 CHIP ORGANIZATION

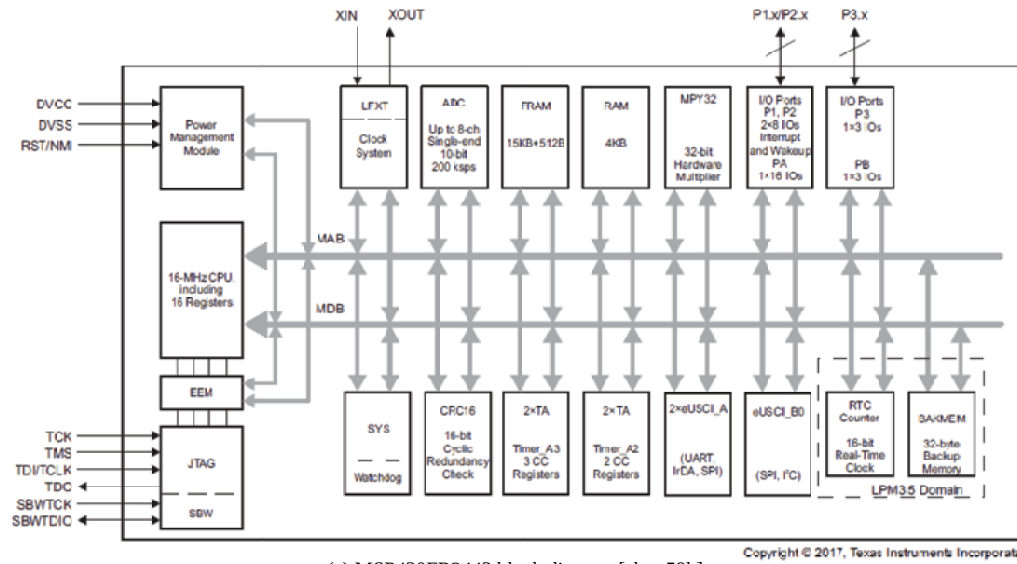
Figure 3.1 shows the block diagrams of the MSP430FR2433 and the MSP430FR5994 microcontrollers. In Chapter 1 we provided a brief overview of each of the subsystems aboard these processors. We briefly recap some of the features here.

Beginning in the upper-left corner of the MSP430FR2433 figure, the power management module is used by the controller to manage power when operating in one of the power saving modes. The CS governs all operations of the controller. The CS may be connected to an external crystal. For the MSP430FR2433 microcontroller, the clock speed can be programmed to be up to 16 MHz and as low as 10 KHz. The speed of the clock is important since at each clock cycle (periodic clock signal completing a cycle to go high and low) the MSP430 is executing an instruction. In general, the faster the microcontroller clock speed, the more tasks that can be accomplished in a given amount of time. Why not use the fastest available clock, then? The faster the clock speed, the more power a microcontroller consumes, causing a designer to balance the speed of the machine with the power consumption. This is especially critical in portable, battery-operated applications, for which MSP430 controllers are designed.

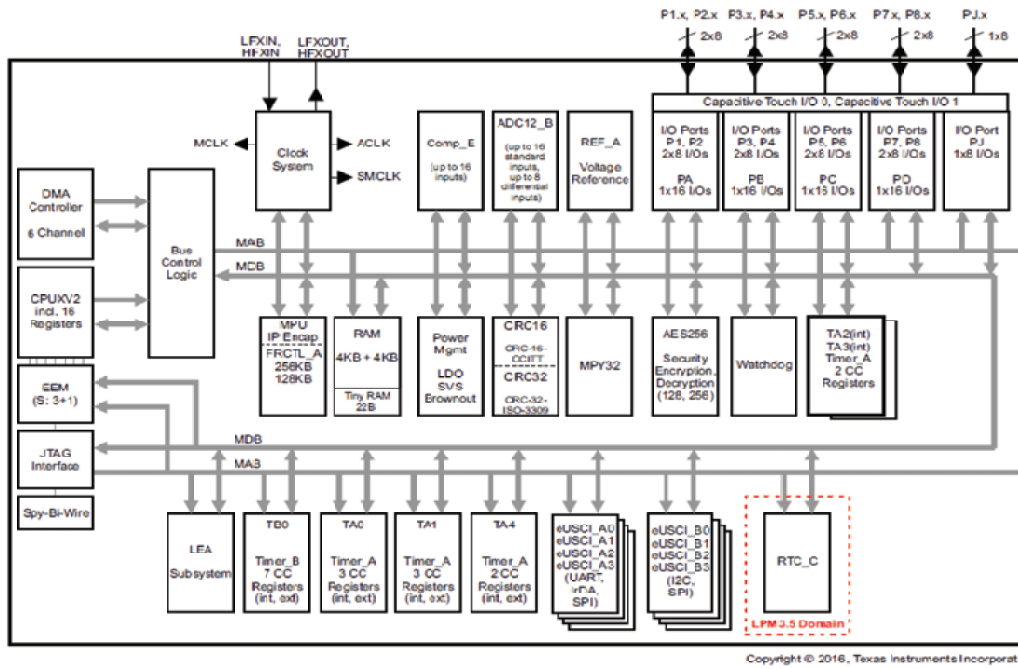
Next is the 10-bit ADC converter block. It is an input port system that takes in analog signals and converts captured signal values into their equivalent 10-bit digital representations.

Continuing from left to right, we find two different types of memory blocks (FRAM and RAM). Typically, the FRAM memory is used to store non-volatile data and programs while RAM is used for temporary data and a storage space during program execution.

The MPY32 module is a hardware-based multiplier subsystem added to the MSP430 microcontrollers to optimize execution time of multiplication operations used frequently in microcontroller-based operations. It typically requires multiple clock cycles to execute the



(a) MSP430FR2443 block diagram [slase59b].



(b) MSP430FR5994 block diagram [slase54b].

Figure 3.1: MSP430FR2433 and the MSP430FR5994 block diagrams [SLASE59D, 2018, SLASE54C, 2018]. Illustrations used with permission of Texas Instruments (www.ti.com).

most basic multiplication operations, requiring excessive execution time. The MSP430 provides the MPY32 hardware-based multiplier to reduce the multiplication operation time. Typically, hardware-based solutions are much faster than their software-based counterparts.

Moving to the right in the figure, the two remaining modules on the first row, show general-purpose I/O port subsystems of the controller. The ports provide the controller access and interface to a wide variety of peripheral components.

On the second row, the purpose of embedded emulator module (EEM) is to assist system developers in providing access to program execution during a debugging process. This is especially helpful during program development and troubleshooting. The built-in emulator allows us to view memory contents and insert break points. The JTAG module is used to interface the controller with a desktop or laptop computer for loading programs and debugging programs using the EEM during program execution. Finally, the CPU in the center of the figure contains the arithmetic and logical unit (ALU), the CPU registers, and a Control subunit responsible for performing program branching and data transfer.

The system Watchdog module (WDT) is responsible for keeping the microcontroller operating properly and taking appropriate actions when abnormal activities are detected. The WDT is a timer that, if expired, results in a processor reset. It is used to reset the processor when a software malfunction has occurred. During normal program processing the WDT is reset by specific program steps. Should a software malfunction occur and the WDT timer is not reset, the WDT will timeout and result in a processor reset. In response to the processor reset, the software malfunction may clear.

The CRC16 subsystem is responsible for the cyclical redundancy check (using 16 bits). Cyclical redundancy checks are used to validate error-free communications among digital systems. Each time a transmitter sends a set of data to a receiver, it sends an additional value derived from the original data called a "key." The receiver, having received the data being sent, uses the original data to perform the same computation to generate the key value. By comparing the generated value with the one received, the receiver can quickly verify whether or not the original data it received are valid.

The timer blocks represent timer-related I/O port systems. The timers are I/O systems where external signals can be captured to measure time-related parameters such as signal frequency, period, and duty cycle, and output signals intended for an external world with specific time-related parameters are generated. In other microcontroller manufacturers, these ports are referred to as input capture and output compare timer ports. The timer port module may also contain channels to generate a variety of PWM output signals. A PWM signal may be used to control the speed of a motor, the light intensity of an LED, or control various peripheral devices.

The next several modules contain the serial communication subsystem that allows the MSP430 microcontroller to interface with external devices using the serial peripheral interface (SPI), universal asynchronous receiver-transmitter (UART), and the inter-integrated circuit (I2C) protocols.

As shown in Figure 3.1b, the MSP430FR5994 has similar features and layout to the MSP430FR2433. In addition, the MSP430FR5994 hosts the additional subsystems.

- The DMA controller shown on the left edge of the figure is responsible for accessing memory by all subsystems of the controller. DMA provides the capability to move data from memory location to memory location without involving the central processing unit. This is especially useful for low power operation when moving data from peripherals to specific memory locations [SLAU367O, 2017].

- The reference module (REF) is responsible for generating voltages required by all peripheral subsystems of the MSP430, including the analog-to-digital converter.

- Advanced encryption standard (AES) accelerator. The accelerator speeds up the encryption and decryption of data by one to two orders of magnitude over a software-based implementation. It can be used for 128- or 256-bit encryption.

3.2.2 HARDWARE PIN ASSIGNMENTS

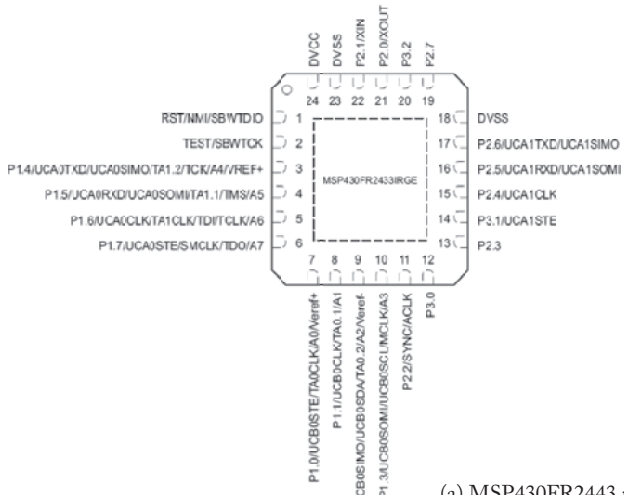
In this section, we consider the pin assignments of the MSP430FR2433 and MSP430FR5994 controllers shown in Figure 3.2. All pins can be divided into I/O pins and pins used to power the controller. For the inputs and outputs pins, we can further categorize them into general-purpose I/O pins, clock signal pins, input pins for the analog-to-digital converter, serial and parallel communication signal pins, and reset and interrupt pins. These pins are also multiplexed, meaning that many of them are used for more than one purpose. The particular use of a pin is software programmed. As a project designers, we typically use a development board with its own I/O pins mapped to the ones for the controller. The pinouts for the EVMs were provided earlier in the book.

3.3 HARDWARE SUBSYSTEMS

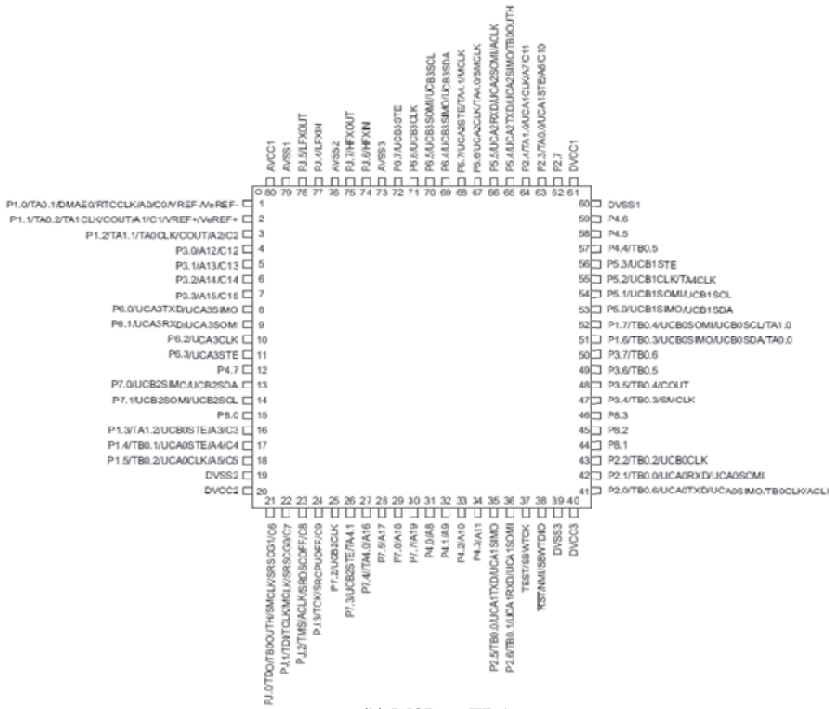
In this section we present an overview of selected hardware subsystems of the controllers: register block, port system, timer system, memory system, resets and interrupts, and communication systems, analog-to-digital converter, and hardware multiplier.

3.3.1 REGISTER BLOCK

The MSP430FR2433 controller has a memory address range from 0000h to FFFFh. These memory locations are partitioned for a variety of uses, including the register block. For the register block, memory locations from 0100h to 074Fh are used. As the name implies, the register block contains registers used to program functional units of the microcontroller. For example, the analog-to-digital converter control registers reside from locations 0700h to 074Fh, whereas, the timer control registers occupy memory locations 0380h to 047FH. Understanding



(a) MSP430FR2443 pinout



(b) MSP430FR5994 pinout

Figure 3.2: MSP430FR2433 and MSP430FR5994 pin assignments [SLASE59D, 2018, SLASE54C, 2018]. Illustrations used with permission of Texas Instruments (www.ti.com).

that the functional unit control registers are part of the allocated memory map is important. The locations should not be used for actual data or instruction storage.

3.3.2 PORT SYSTEM

One primary feature of a microcontroller compared to a general-purpose computer processor is its ability to interface with external devices. The MSP430 microcontroller has a number of flexible I/O capabilities through its I/O ports. The MSP430FR2433 has Port 1, Port 2, and a small Port 3 that can be configured to be used as general-purpose input and output pins. For the MSP430FR5994, there are nine different I/O ports: Port 1 through Port 8 and Port J. These ports can be programmed as general I/O ports or shared with other subsystems. The MSP430FR5994 is equipped with the TI incorporated capacitive touch capabilities for the ports.

3.3.3 TIMER SYSTEM

The MSP430 controller has a number of clock sources it can use to manage its computational speed and power consumption. The timer system is based on a clock that is used by the CPU of the microcontroller to fetch instructions and data from memory or I/O ports, encode and execute instructions, and store results to memory or send them to external devices. The timer system can use its built-in function such as the pulse width modulated signals for controlling devices, the Watchdog timer to monitor proper execution of instructions, the real-time counter to measure timer-related parameters of an incoming signals. The important capabilities of the timer system are its ability to capture signal characteristics and generate desired signal with time-related parameters such as the frequency of a periodic signal. These subsystems of the timer system are called input capture and output compare systems.

3.3.4 MEMORY SYSTEM

The memory system for the MSP430 controller is made of two parts: actual memory locations and the memory control system. The memory locations are governed by the number of address lines used to identify each memory location. Sixteen address lines (can be expanded using additional address lines) are used to access 64 K locations which are made of a register block, ROM, RAM, and FRAM for the MSP430FR2433 controller. For the MSP430FR5994, 18 address lines are used to access 256 K locations of the register block, ROM, RAM, and FRAM locations. The memory map for the MSP430FR2433 microcontroller is shown in Figure 3.3 and the one for the MSP430FR5994 microcontroller in Figure 3.4. Memory maps are necessary to inform a programmer to use proper memory locations and types to store data and instructions for his or her application.

The memory control system is made of three memory controllers, the FRAM memory controller, the RAM controller, and the DMA controller. The FRAM memory controller is used to program the fast-write nonvolatile memory locations, access capability, wait state, power

	Memory Contents	Start Address	Stop Address	Span
$(FFFF)_{16}$	Interrupt Vectors (FRAM)	$(FF80)_{16}$	$(FFFF)_{16}$	$(80)_{16}$ $(128)_{10}$ ~ 128 bytes
$(FF80)_{16}$	Code Memory (FRAM)	$(C400)_{16}$	$(FFFF)_{16}$	$(3C00)_{16}$ $(15360)_{10}$ ~ 15K bytes
$(C400)_{16}$				
$(3000)_{16}$	Random Access Memory (RAM)	$(2000)_{16}$	$(2FFF)_{16}$	$(01000)_{16}$ $(4096)_{10}$ ~ 4K bytes
$(2000)_{16}$	Information Memory (FRAM)	$(1800)_{16}$	$(19FF)_{16}$	$(00200)_{16}$ $(512)_{10}$ bytes
$(1800)_{16}$	Bootstrap Loader Segment 1	$(1000)_{16}$	$(17FF)_{16}$	$(0800)_{16}$ $(2048)_{10}$ ~ 2K bytes
$(1000)_{16}$	Peripherals	$(0000)_{16}$	$(0FFF)_{16}$	$(1000)_{16}$ $(4096)_{10}$ ~ 4K bytes
$(0000)_{16}$				

Figure 3.3: The MSP430FR2433 memory map [SLAU445G, 2016].

saving, and error correction coding. The RAM controller is used to power down in a power saving mode, and the DMA controller is used to transfer contents of memory locations to other memory locations without the intervention of a controller.

3.3.5 RESETS AND INTERRUPTS

Resets and interrupts are used to "interrupt" the regular flow of a program sequence to attend to a set of special instructions for designated purposes. For example, regardless of the current instruction being executed, a reset will bring the status of the controller to a power on state. MSP430 controllers have a variety of resets and interrupts. There are internal and external interrupts and resets. MSP controllers also have timer system interrupts, I/O-related interrupts, and analog-to-digital controller interrupts. The use of interrupts is presented throughout the book.

	Memory Contents	Start Address	Stop Address	Span
$(FFFF)_{16}$	Interrupt Vectors (FRAM)	$(0FF80)_{16}$	$(0FFFF)_{16}$	$(80)_{16}$ $(128)_{10}$ ~ 128 bytes
$(FF80)_{16}$	Code Memory (FRAM)	$(04000)_{16}$	$(043FFF)_{16}$	$(40000)_{16}$ $(262,144)_{10}$ ~ 256K bytes
$(03C00)_{16}$	Random Access Memory (RAM)	$(01C00)_{16}$	$(03BFF)_{16}$	$(2000)_{16}$ $(8192)_{10}$ ~ 8K bytes
$(01C00)_{16}$				
	Device Descriptor (TLV) (FRAM)	$(01A00)_{16}$	$(01AFF)_{16}$	$(00100)_{16}$ $(256)_{10}$ bytes
$(01A00)_{16}$	Information Memory A-D (FRAM)	$(01800)_{16}$	$(019FF)_{16}$	$(00200)_{16}$ $(512)_{10}$ bytes
$(01800)_{16}$	Bootstrap Loader Segment 0-3	$(01000)_{16}$	$(017FF)_{16}$	$(800)_{16}$ $(2048)_{10}$ ~ 2K bytes
$(01000)_{16}$	Peripherals	$(00020)_{16}$	$(00FFF)_{16}$	$(\sim 1000)_{16}$ $(4096)_{10}$ ~ 4K bytes
$(00000)_{16}$				

Figure 3.4: The MSP430FR5994 memory map [SLAU367O, 2017].

3.3.6 COMMUNICATION SYSTEMS

MSP430 controllers support different communication modes: serial communication interface (SCI), synchronous peripheral interface (SPI), and inter-integrated circuit (I^2C) communication. The first one is the universal asynchronous receiver and transmitter (UART) mode. It uses either seven or eight data bits, programmable baud rates, and interrupt capabilities. The controller supports both synchronous and asynchronous communication patterns, and the automatic error detection. In the SPI mode, multiple devices can be connected to transmit and receive data using a same clock signal with a master unit providing the clock source. In the I^2C mode, multiple devices are connected together with multiple master units working on the same network. The communication units of the controller is discussed in Chapter 10.

3.3.7 ANALOG-TO-DIGITAL CONVERTER

For microcontrollers to work with analog signals, we must first convert the analog signal into a digital one. The 10-bit analog-to-digital converter of the MSP430FR2433 (12-bit

MSP430FR5994) controller performs that task by implementing sampling, quantization, and encoding processes. The sampling process is taking time-based samples of an analog signal, the quantization process is mapping a sampled analog value to one of the available encoded level (1024 levels generated by the 10 bits), and the encoding step is representing the quantized level as a digital number.

Once a signal has been converted into a digital form, it can now be used in any application that requires digital representation of an analog signal. The controller has multiple analog-to-digital converters that can be programmed with interrupt capabilities. The converter is presented in Chapter 9.

3.3.8 HARDWARE MULTIPLIER (MPY32)

The multiplication operation typically requires multiple clock cycles. A hardware-based multiplier significantly speeds up the multiplication operation. The MSP430 is equipped with the MPY32 Hardware Multiplier. This system provides the flexibility to multiply any signed or unsigned 8-bit, 16-bit, and 32-bit numbers. There is also the option to multiply a 24-bit number with an 8-bit number as well as performing a multiplication and accumulation (add to the previous operation result). This combination of operations is common in digital signal processing (DSP) operations. Signed numbers are used when both negative and positive numbers need to be represented. These numbers use the 2's complement representation. Unsigned numbers assume all numbers, including results of operations, are positive numbers.

The multiplication operation is performed automatically, without explicit multiplication instructions, when both operand registers are loaded with numbers. It takes some practice to perform the multiplication operations since a number of specific registers that correspond to the right type of multiplication desired to be performed must be used for different types of multiplications. The registers for the operands are listed in Table 3.2 along with their base offset addresses. Pay special attention to the register column containing information on which type of operand registers are created.

3.4 CPU PROGRAMMING MODEL/REGISTER DESCRIPTIONS

The MSP430 CPU block diagram is provided in Figure 3.5. The MSP430 CPU consists of a series or registers designated R0 through R15. The registers are served by two buses: the memory data bus (MDB) and the memory address bus (MAB). Register contents are provided to the ALU. The MCLK serves as the clock source for the ALU. ALU operations are specified by the program under execution. In response to ALU operations different flags (Zero, Carry, Overflow, and Negative) are set and reset during program execution.

The programming model of a computer provides a simplified model of the CPU and the operational status of the computer at a given time. A programming model is typically comprised

Table 3.1: Arithmetic operations

Register	Type	Access	Address Offset
16-bit operand one (MPY)	R/W	Word	00h
16-bit operand one low byte (MPY_L)	R/W	Byte	00h
16-bit operand one high byte (MPY_H)	R/W	Byte	01h
8-bit operand one (MPY_B)	R/W	Byte	00h
16-bit operand one signed (MPYS)	R/W	Word	02h
16-bit operand one signed low byte (MPYS_L)	R/W	Byte	02h
16-bit operand one signed high byte (MPYS_H)	R/W	Byte	03h
8-bit operand one signed (MPYS_B)	R/W	Byte	02h
16-bit operand one accumulate (MAC)	R/W	Word	04h
16-bit operand one accumulate low byte (MAC_L)	R/W	Byte	04h
16-bit operand one accumulate high byte (MAC_H)	R/W	Byte	05h
8-bit operand one accumulate (MAC_B)	R/W	Byte	04h
16-bit operand one signed accumulate (MACS)	R/W	Word	06h
16-bit operand one signed accumulate low byte (MACS_L)	R/W	Byte	06h
16-bit operand one signed accumulate high byte (MACS_H)	R/W	Byte	07h
8-bit operand one signed accumulate (MACS_B)	R/W	Byte	06h
16-bit operand two (OP2)	R/W	Word	08h
16-bit operand two low byte (OP2_L)	R/W	Byte	08h
16-bit operand two high byte (OP2_H)	R/W	Byte	09h
8-bit operand two (OP2_B)	R/W	Byte	08h
16x16 result low word (RESLO)	R/W	Word	0Ah
16x16 result low word low byte (RESLO_L)	R/W	Byte	0Ah
16x16 result high word (RESHI_H)	R/W	Word	0Ch
16x16 sum extension (SUMEXT)	R	Word	0Eh
32-bit operand one low word (MPY32L)	R/W	Word	10h

Table 3.2: Arithmetic operations (*Continues.*)

Register	Type	Access	Address Offset
32-bit operand one low word low byte (MPY32L_L)	R/W	Byte	10h
32-bit operand one low word high byte (MPY32L_H)	R/W	Byte	11h
32-bit operand one high word (MPY32H)	R/W	Word	12h
32-bit operand one high word low byte (MPY32H_L)	R/W	Byte	12h
32-bit operand one high word high byte (MPY32H_H)	R/W	Byte	13h
24-bit operand one high byte (MPY32H_B)	R/W	Byte	12h
32-bit operand one signed low word (MPYS32L)	R/W	Word	14h
32-bit operand one signed low word low byte (MPYS32L_L)	R/W	Byte	14h
32-bit operand one signed low word high byte (MPYS32L_H)	R/W	Byte	15h
32-bit operand one signed high word (MPYS32H)	R/W	Word	16h
32-bit operand one signed high word low byte (MPYS32H_L)	R/W	Byte	16h
32-bit operand one signed high word high byte (MPYS32H_H)	R/W	Byte	17h
24-bit operand one signed high byte (MPYS32H_B)	R/W	Byte	16h
32-bit operand one accumulate low word (MAC32L)	R/W	Word	18h
32-bit operand one accumulate low word low byte (MAC32L_L)	R/W	Byte	18h
32-bit operand one accumulate low word high byte (MAC32L_H)	R/W	Byte	19h
32-bit operand one accumulate high word (MAC32H)	R/W	Word	1Ah
32-bit operand one accumulate high word low byte (MAC32H_L)	R/W	Byte	1Ah
32-bit operand one accumulate high word high byte (MAC32H_H)	R/W	Byte	1Bh
24-bit operand one accumulate high byte (MAC32H_B)	R/W	Byte	1Ah

Table 3.2: (*Continued.*) Arithmetic operations

Register	Type	Access	Address Offset
32-bit operand one signed accumulate low word (MACS32L)	R/W	Word	1Ch
32-bit operand one signed accumulate low word low byte (MACS32L_L)	R/W	Byte	1Ch
32-bit operand one signed accumulate low word high byte (MACS32L_H)	R/W	Byte	1Dh
32-bit operand one signed accumulate high word (MACS32H)	R/W	Word	1Eh
32-bit operand one signed accumulate high word low byte (MACS32H_L)	R/W	Byte	1Eh
32-bit operand one signed accumulate high word high byte (MACS32H_H)	R/W	Byte	1Fh
24-bit operand one signed accumulate high byte (MACS32H_B)	R/W	Byte	1Eh
32-bit operand two low word (OP2L)	R/W	Word	20h
32-bit operand two low word low byte (OP2L_L)	R/W	Byte	20h
32-bit operand two low word high byte (OP2L_H)	R/W	Byte	21h
32-bit operand two high word (OP2H)	R/W	Word	22h
32-bit operand two high word low byte (OP2H_L)	R/W	Byte	22h
32-bit operand two high word high byte (OP2H_H)	R/W	Byte	23h
24-bit operand two high byte (OP2H_B)	R/W	Byte	22h
32x32 result 0 (RES0)	R/W	Word	24h
32x32 result 0 low byte (RES0_L)	R/W	Byte	24h
32x32 result 1 (RES1)	R/W	Word	26h
32x32 result 2 (RES2)	R/W	Word	28h
32x32 result 3 (RES3)	R/W	Word	2Ah

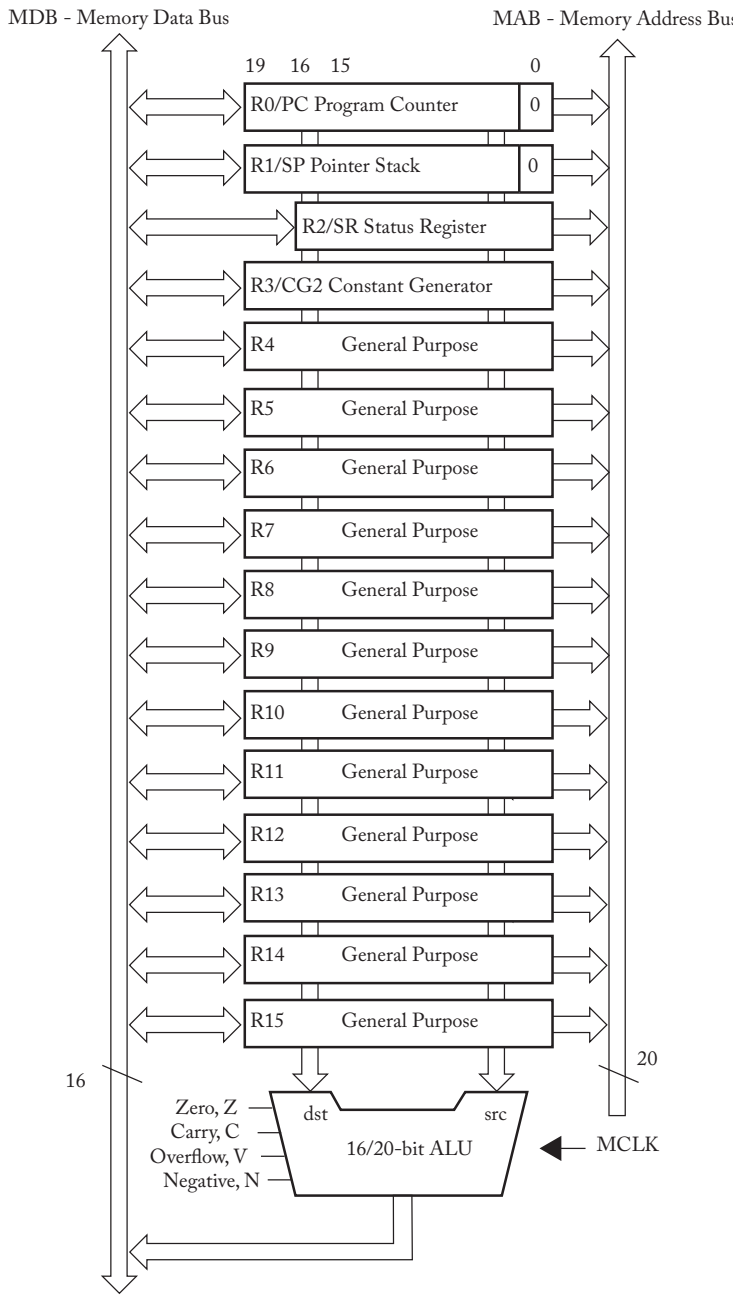


Figure 3.5: MSP430 CPU block diagram [SLAU208Q, 2018]. Illustration used with permission of Texas Instruments (www.ti.com).

of CPU registers which collectively contain pertinent information about the operational status of the computer. For example, a programming model contains a status register that shows whether the most recently executed instruction resulted in a negative number, a positive number, or a zero. It shows which instruction is on queue to be executed next via the program counter (PC) and the next available location on the stack via the stack pointer (SP).

A stack is a designated portion of RAM memory which is used to store data temporarily during program execution. It is a first-in-last-out data structure, similar to the spring-loaded plate holder one finds in a buffet restaurant. The first plate placed in the plate holder is the last one used by customers. To keep track of the size of a stack, the top of the stack (either the next available memory location to be used by the stack or the last memory location used by the stack depending on the computer architecture) is always identified by a special register called the SP. A programming model also includes the PC, which contains the address of the instruction to be executed next.

The programming model for MSP430 is shown in Figure 3.6. As shown, the MSP430 microcontroller has four special and 12 general-purpose CPU registers that make up its programming model. Registers are 16 or 20 bits wide. The first special register, R0, is the PC register. This register contains the address of the next instruction to be executed by CPU. Since all instructions are aligned and start at even addresses, the LSB of R0 is hardwired to zero. The value of PC is incremented by two each time an instruction is executed to update the address of the instruction to be executed next.

The second special register, R1, is the SP register, which contains the address of the top of the stack. Since data is "pushed" onto the stack and "pulled" off the stack using only one end of the stack, the top of the stack must be monitored always. The top of the stack contains the address of the most recently added word to the stack. Again, the MSP430 always puts (pushes) and pulls data onto and from the stack one word (16 bits) at a time, which allows the designer of the MSP430 to hardwire the LSB of SP to zero.

The R2 register contains the current status of the computer operation. Figure 3.7 shows the contents of the Status Register R2. The MSP430 status is shown as a series of status bits. Most of the flag definitions are quite straightforward. The definition of the V bit is a bit more involved.

The V bit represents overflow from the execution of the previous instruction. An overflow occurs when the result of a **signed** number operation is incorrect due to the available, limited space of a register to correctly hold the result. When this bit is set, an overflow occurred, and when it is cleared, no overflow occurred. There are several different situations that result in an overflow condition (adapted from Hamann [2017], Miller [1995], Mano [2002], and CPU12 [1997]).

- In an **unsigned addition** operation, an overflow occurs when there is a carry out of the MSB after an addition operation.
- In an **unsigned subtraction** operation, an overflow occurs if the MSB requires a borrow.

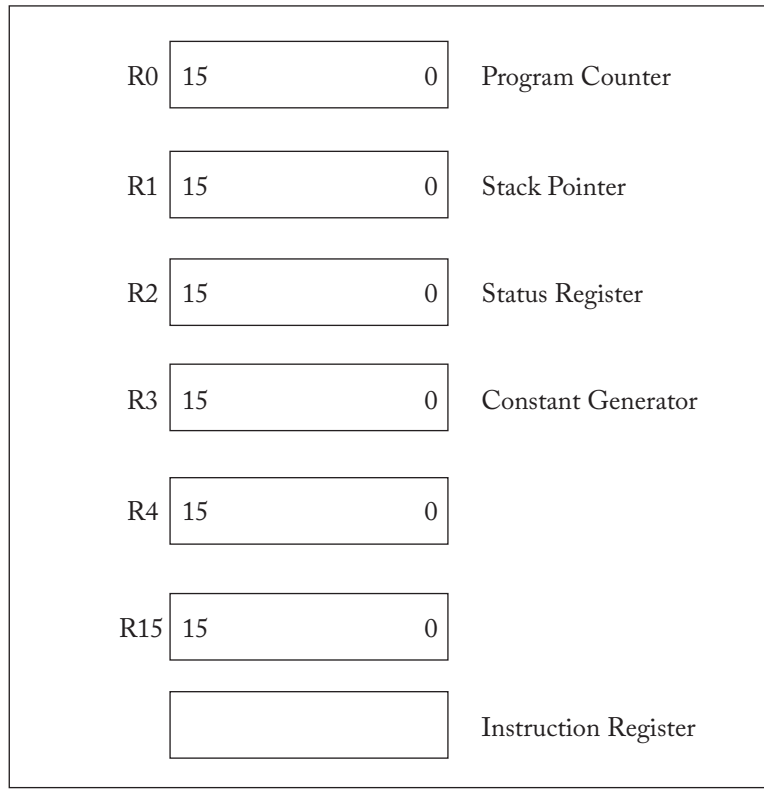


Figure 3.6: Programming model of MSP430.

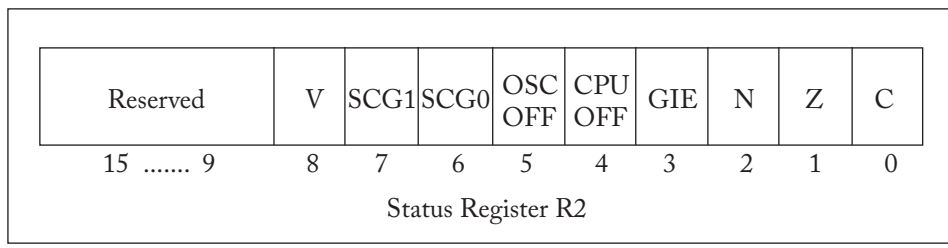


Figure 3.7: The contents of status register R2 [SLAU056L, 2013].

- In a **signed addition** operation, an overflow occurs if the addition of two positive numbers results in a negative number or if the addition of two negative numbers results in a positive number. Recall that, in the two's complement system, a positive number is represented by having the MSB of a variable set to logic 0, whereas a negative number has a logic 1 in the MSB position.
- In a **signed subtraction** operation, an overflow occurs if the subtraction of a negative number from a positive number yields a negative number or if the subtraction of a positive number from a negative number yields a positive number.

The N bit is set when the result of the previous operation is a negative number in the 2's complement sense. It is cleared if this bit is zero. The Z bit is set when the previous instruction's result was zero. It is cleared if the result of the previous instruction was not zero. The C bit represents a carry bit. This bit is set when a carry to or borrow from of the MSB occurs. It is cleared if there is no carry or borrow was generated by the previously executed instruction.

The GIE bit (General Interrupt Enable) is used to enable or disable maskable interrupts. Interrupts are requests made by hardware or software to the CPU to temporarily suspend the current flow of instruction execution and handle a special request by performing a set of instructions specially written to take care of the request. Interrupt-related activities are of higher priority than routine processing tasks.

The remaining status bits SCG1 (System Clock Generator 1), SGC0 (System Clock Generator 0), OSCOFF (Oscillator Off), and CPUOFF (CPU Off) are used to control the operational mode of the CPU, which is the topic of the next section. The numbers under each status bit in Figure 3.7 provide their bit location within the 16-bit status register. The bits are numbered from the LSB (0) on the right to the MSB (15) on the left.

The last special register, R3, is a constant generator, which generates numeric values zero, one, two, four, and eight. The rationale for having this register is based on the frequent use of the constant values. By having these values in the CPU register, the MSP430 microcontroller can compute instructions that use such constants with minimal clock cycles. The rest of CPU registers, R4–R15, are general purpose-registers.

Examples: In each of the examples below, predict the value of the status register bits (V, N, Z, C) after each operation. Assume all arguments and results are contained within 16-bit registers.

1. 0007h + 0003h

   Answer: The results is 000Ah, no status flags are set.

2. CAFEh + 0D00h

   Answer: The result is D7FEh, the N flag will set.

3. 7B7Ah - 7B7Ah

   Answer: The results is 0000h, the Z flag will set.

4. DECAh + CA70h

   Answer: The result is A93A, the C, V, and N flags will set.

3.5 OPERATING MODES

The MSP430 microcontroller can run in an active mode and in a number of power-saving operating modes. The basic premise behind the low-power operating modes is to turn off system clocks that are not currently in use. Since a CMOS circuit consumes power when switching, turning off clocks not in use conserves power. The operating mode of the controller is determined by the settings of four bits within the status register (R2): CPUOFF, OSCOFF, SCG0, and SCG1. By configuring these four bits, a programmer can select the operating mode of the controller based on the needs of a system application.

One of the advantages of designating the low-power operating mode using the bits in the status register is that when an interrupt occurs, the operating mode configured in Status Register R2 is automatically saved onto the stack, and the same operating mode is retrieved when the interrupt is serviced and the program returns to routine execution. We discuss options to set or clear low-power operational modes upon returning from servicing an interrupt in Chapter 8.

Here is a brief summary of MSP430 operating modes.

- **Active mode.** During normal operation in the active mode, all four bits should be cleared (CPUOFF = 0, OSCOff = 0, SCG0 = 0, and SCG1 = 0), which allows all CPU clocks to become active.

- **LPM0.** When only the CPUOFF bit is set, called low-power mode 0 (LPM0), the CPU and the master clock (MCLK) are turned off and only the subsystem clock (SMCLK) and auxiliary clock (ACLK) operate. The controller can be awakened by enabled interrupts, special requests generated internally or externally (to be studied in Chapter 8).

- **LPM1.** Low-power mode 1 (LPM1) is similar to LPM0 with CPUOFF and SCG0 bits set while the other two bits are cleared. In this mode, the CPU, MCLK, SMCLK, and ACLK are configured the same as in LPM0. The digitally controlled oscillator (DCO) is turned on/off based on the use of the DCO by the SMCLK.

- **LPM 2.** The MSP430 operates in the low-power mode 2 (LPM2) when both SCG0 and OSCOFF bits are cleared and the SCG1 and CPUOFF bits are set. In this mode, the CPU, MCLK, and SMCLK are turned off as well as the DCO. The dc-generator of the DCO remains active along with the ACLK.

- **LPM 3.** When the CPUOFF, SCG0, and SCG1 bits are set and the OSCOFF bit is cleared, the MSP430 is operating in the LPM3 operating mode; all clocks and the DCO and CPU are turned off. Only the ACLK is enabled.

- **LPM 3.5.** When all of bits are set (CPUOFF, OSCOff, SCG0, and SCG1) and the PMMREGOFF is set (Regulator off bit in the Power Management Module Control Register 0), the MSP430 is operating in LPM3.5. In this LPM the voltage regulator in the Power Management Module is disabled. The RAM and register contents are lost but RTC operation is possible.

- **LPM 4.** When all of bits are set (CPUOFF, OSCOff, SCG0, and SCG1), the MSP430 is operating in LPM4; the CPU, all clocks, the DCO, and the crystal oscillator are turned off. The controller is wakened up by an enabled interrupt.

- **LPM 4.5.** In low-power mode 4.5 (LPM4.5), the MSP430 achieves the lowest power consumption. In this mode, in addition to the CPU, MCLK, SMCLK, and the DCO being turned off, the regulator for the PMM is also turned off. This results in turning off the JTAG and EEM logic devices. These operating modes will be discussed in depth in Chapter 5 when we discuss the CS onboard the MSP430 microcontroller. The different processor operating modes are summarized in Figure 3.8.

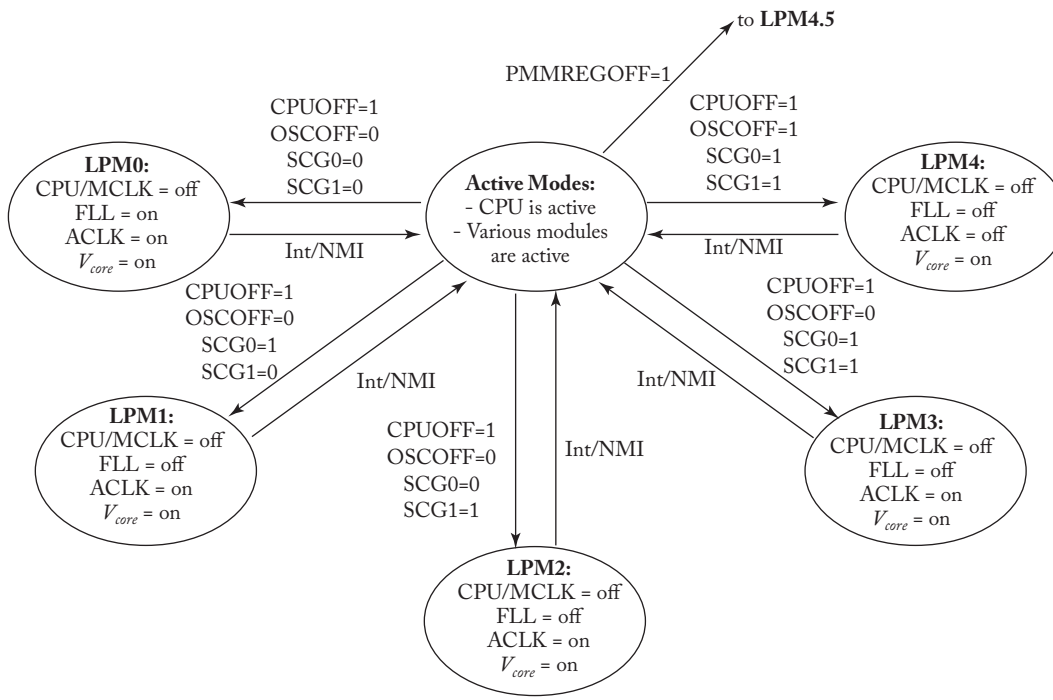


Figure 3.8: MSP430 operating modes (adapted from Texas Instruments SLAU208Q [2018]).

3.6 SOFTWARE PROGRAMMING

In this section, we describe tools and constructs of software programming for MSP430 controllers, including what is known as the instruction set architecture of the microcontroller. An instruction set architecture of a CPU determines the complexity of computer hardware and required compiler technologies. The MSP430 controller architects chose to design a RISC-based microcontroller by limiting the type of instructions that are interpreted by the controller. The designers of the MSP430 sought to increase the CPU performance by making all instructions have a fixed length (1–4 words) and a fixed execution time. Such simplification of the instruction set decreases hardware complexity and allows easier implementation of hardware modules such as pipelining and scoreboarding.[1]

Each instruction consists of an opcode and an optional operand. The opcode is the portion of the instruction that instructs hardware the nature of the instruction to be performed, and the operand specifies either the location of or the data itself necessary for the instruction. As we will see in this section, a method used to identify the necessary data for an instruction is called an addressing mode.

When an editor is used to write assembly instructions for the MSP430, each instruction should comply with the format shown below. If the C language is used to write your program, each C instruction is converted to one or more assembly language instructions with the same format shown below. Each instruction can have up to four separate fields, each field separated by one or more blank spaces.

```
Label   Mnemonic(Opcode)   Operand   ; Comment
```

The Label field should start at column one of your editor and should be made of alphanumeric characters. Labels are used to name particular locations within a program, such as a location where a loop should return to or a location of an instruction set that needs to be executed as a result of a conditional decision (if-then-else). The mnemonic field contains assembly language instructions while the operand field contains an expression, symbols, and/or constants identifying data required for the instruction specified in the mnemonic field. For TI's presentation of the MSP430 instruction set, reference the *MSP430x4xx Family User's Guide* [SLAU056L, 2013]. It is not mandatory that each instruction have all four fields filled. As you will see, some instructions do not require operands. For some instructions, providing comments may not be appropriate. Furthermore, it would be excessive to put a label on every instruction.

[1]The principle of pipelining is to utilize all computer resources at all times to minimize the overall clock cycles to execute a set of instructions. RISC-based instructions support such design better than CISC (Complex Instruction Set Computer)--based instructions since each instruction goes through the same process for execution, making it easy for a CPU to issue one instruction and to complete an instruction at each clock cycle. The scoreboarding module's goal is the same as the one for pipelining: minimize clock cycles required to execute instructions. By keeping track of the progress of instructions and allowing instructions that are ready to be executed to proceed to completion without waiting for other instructions to complete their tasks, the module supports the goal of executing instructions within a minimal time.

3.6.1 MSP430 ASSEMBLY LANGUAGE

The MSP430 assembly language is designed to support the hardware functional units of the controller. It is made of an instruction set, discussed in the next section. There is also a set of instructions, directives used by an assembler, which we discuss next.

3.6.2 DIRECTIVES

Simply put, directives are instructions to assemblers. Directives are part of an assembly program, but they do not get converted to machine code. Instead, directives are interpreted and executed by assemblers. Since directives are executed by assemblers, available directives vary from an assembler to an assembler. In this section, we describe the directives available for most of MSP430 assemblers: the IAR assembler and Code Composer.

There are two types of directives: macro directives and assembler directives. We explain assembler directives first. Assembler directives are used by programmers to specify locations for different segments of your program, initialize defined variables, reserve memory locations for variables used in your program, and define constants, to name some of the common uses of assembler directives. Some assembler directives are used to control the assembly process, examine symbols, and specify libraries for macros. All MSP430 microcontroller directives start with a period (.). For a complete presentation of directives, see the *MSP430 Assembly Language Tools User's Guide* [SLAU131R, 2018].

One can group these assembler directives into one of the following categories.

1. Define sections.
2. Define constants, variables, and reserve memory locations.
3. Control output lists.
4. Specify reference files.
5. Control assembly process flow.
6. Others

Directives that Define Sections

These directives are used by a programmer to define the starting locations of data and segments of instructions. Each program section should start with directive *.text* as shown below:

```
    .text
Assembly instructions
```

The assembler will take all instructions that follow the *.text* directive and assemble them into the *.text* section. When the *.sect* directive is used, all instructions that follow the directive will be assembled into a pre-named section that follows the directive: *.sect* "section name." The

.data directive works similarly. All declarations that follow this directive are assembled into the *.data* section. The data section is used to initialize data used in the program.

Directives to Define Constants, Variables, and Reserve Memory Locations

The *.bss* directive is used to reserve bytes in a section whose data are not initialized. The format to use the directive is as follows:

.bss symbol name,# bytes [,alignment]

The above directive reserves the designated number of bytes in the .bss section. The *.byte* directive initializes one or more successive bytes: .byte value1, value 2,.. If characters need to be initialized, one can use the *.char* directive, which has the same format as the *.byte* directive. For initializing double (32-bit IEEE double precision) or floating point (32-bit, IEEE single precision) values, use .double and .float directives, respectively. One can also define different size bit values using the *.field* directive whose format is shown below:

.field value[, size]

A programmer can specify the size and specify the value. Integers are also defined with *.half, .int, .long, .short*, and *.word* directives. The *.half, .int, .short*, and *.word* directives initialize one or more successive 16-bit integers while the *.long* directive initializes 32-bit integers. To initialize strings, one should use the *.string* directive with the following format:

.string {expr$_1$|"string$_1$"}[,...,{expr$_n$|"string$_n$"}]

There are a number of directives that reserve memory locations. The *.symbol* directive reserves a number of bytes in a named section as follows.

.symbol .usect "section name",# bytes [,alignment]

The *.align* directive is used to align the program counter (PC) to a location specified by the number of bytes within a boundary, which must be power of two. Both the *.bes* and *.space* directives reserve the #bytes specified by the parameter which follows the directives in the current section of the code. The difference between the two directives is that the label associated with the *.bes* directive contains the memory address of the end of the reserved space while the label associated with the *.space* directive contains the address of the beginning directive of the reserved space. The *.struct* and *.endstruct* directives begins and ends a structure definition. The *.tag* directive is used to define a structure and assign to a label. The *.asg* assigns a character string to a symbol: *.asg* character string, symbol. The *.equ* directive is used to equate a constant to a symbol, while the *.label* directive is used to define a re-locatable label in a section. The *.eval* directive (.eval expression, symbol) performs arithmetic expression and assigns to a symbol, while the *.var* directive adds a local symbol definition to a macro's permanent list.

Control Output Lists

The directives that we present in this section govern what will be available for programmers to see as the output of the assembler. The *.drlist* directive is used to enable all directive lines to show in the *.lst* file, which is the default setting. If one does not wish to see the directive lines, he/she can use the *.drnolist* directive to suppress the output. The *.fclist* directive, the default setting, is used to list the false conditional code block, while the *.fcnolist* disables the output. The *.length* [page length] directive designates the page length of the source listing. The *.list* directive instructs an assembler to restart the source listing.

The *.mlist* directive enables macro listings and loop blocks to be listed (default), while the *.mnolist* directive disables the feature. The *.option* directive allows a programmer to select output listings and the *.page* directive removes a page in the source listing. The *.sslist* directive is used to include expanded symbols in the listing while the *.ssnolist* directive disables this feature. The *.tab* directive is used to specify the tab size. The *.title* directive instructs an assembler to print the title of the list while the *.width* directive sets the page width of the list file.

Directives to Reference Other Files

In this section, we consider assembler directives that allow a programmer to refer to variables and programs written previously. The *.copy* directive is used to include declarations made in another file: *.copy* filename. The *.def* directive is used to define one or more symbols which can be used in the current program or other programs. The *.global* directive is used to define external or global symbols while the *.ref* directive is used to identify symbols that are defined in other program modules.

The *.include* directive is used to include source statements from other files just as the *.macro* directive is used to define macro libraries.

Control Assembly Process Flow

The following directives are used to enable conditional assembly. The *.break* directive ends the execution of a program if the expression that follows the directive is true. The *.if* directive dictates an assembler to assemble a block of code if the expression that follows the directive is true. The *.else* directive is used toassemble a block of code if the expression for the *.if* directive is false. The *.elseif* directive is used to assemble code if the *.if* directive expression is false and the *.elseif* expression is true. The *.endif* directive ends the block of code that starts with the *.if* directive. The *.loop* directive is used to designate a block of code that will repeat as long as the expression that follows the directive is true. The *.endloop* directive ends the *.loop* directive code.

Others

There are other directives that do not fall under the categories we defined. We explain the remaining assembler directives here. The *.asmfunc* directive is used to locate the beginning of a code section that contains a designated function. The *.cdecls* directive allows an assembly pro-

gram and a C program to share a header file. The *.clink* directive is used to conditionally link current code segment. The *.emsg* directive can be used to send pre-defined error messages to an output device. The *.end* directive informs an assembler to ignore instructions that follow this directive, while the *.endasmfunc* directive identifies the end of a block that contains a function. The *.mmsg* directive is used to send user messages to output device while the *.wmsg* directive is used to send warning messages to output device.

Finally, the *.newblock* directive is used to indicate a new section of code which undefines any previous local labels. Table 3.4 contains a summary of all assembler directives presented in this section.

As you just saw, there are many, many directives you can use. We suggest you start with a small set of directives in your arsenal and expand your use of other directives as you acquire the MSP430 controller programming skills. Before leaving this section, we present a type of directives called "macro" directives. Macro directives are also instructions to an assembler. There are two reasons why you may want to learn to use macro directives.

The first reason is macro directives allow a programmer to combine a number of assembly instructions and assembly directives and define them as a single macro instruction. Such capability becomes very useful if you want to repeat executing a set of instructions over and over. For example, suppose your application requires you to read from an input port (16 bits), swap the high byte with the low byte, invert each bit of the resulting value, and write to an output port (16 bits). Instead of writing the required instructions each time you need the task to be performed, you can write the following macro directive and call it when it is required. Suppose, for our discussion, the input port is at address 0000h and the output port is at 0002h.

```
    Invert  .macro  input, output
SWAP    input, R15
BIT.W   R15, output
.endm
```

The general format of a macro directive has the following format.

1. macname .macro [parameter$_1$[,...,parameter$_n$]]
2. assembly instructions or assembler directives
n. [.mexit]
m. .endm

The Name_of_macro on line 1 is used as the label of a macro directive. For each macro directive, you have an option to include parameters. These parameters are called substitution symbols. On line 2 and before line 3, you put in assembly language instructions and assembler directives that perform the task you desire. The optional[2] *.mexit* assembler directive is used to direct the flow of

[2]Brackets [ ] are used to specify optional items.

Table 3.3: Directives (*Continues.*)

Directive	Description	Section
$	Includes a file	Assembler control
#define	Assigns a value to a label	C-style preprocessor
#elif	Introduces a new condition in a #if..#endif block	C-style preprocessor
#else	Assembles instructions if a condition is false	C-style processor
#endif	Ends a #if, #ifdef, or #ifndef block	C-style processor
#error	Generates an error	C-style processor
#if	Assembles instructions if a condition is true	C-style processor
#ifdef	Assembles instructions if a symbol is defined	C-style processor
#ifndef	Assembles instructions if a symbol is undefined	C-style processor
#include	Includes a file	C-style processor
#message	Generates a number of messages on standard output	C-style processor
#undef	Undefines a label	C-style processor
/*comment*/	C-style comment delimiter	Assembler control
//	C++ style comment delimiter	Assembler control
=	Assigns a permanent value local to a module	Value assignment
ALIAS	Assigns a permanent value local to a module	Value assignment
ALIGN	Aligns the location counter by inserting zero-filled bytes	Segment control
ALIGNRAM	Aligns the program location counter	Segment control
ASEG	Begins an absolute segment	Segment control
ASEGN	Begins a named absolute segment	Segment control
ASSIGN	Assigns a temporary value	Value assignment
CASEOFF	Disables case sensitivity	Assembler control
CASEON	Enables case sensitivity	Assembler control
CFI	Specifies call frame information	Call frame information
COL	Sets the number of columns per page	Listing control
COMMON	Begins a common segment	Segment control
DB	Generates 8-bit byte constants, including strings	Data definition or allocation
DC16	Generates 16-word constants, including strings	Data definition or allocation
DC32	Genrates 32-bit long word constants	Data definition or allocation
DC8	Generates 8-bit byte constants, including strings	Data definition or allocation

Table 3.3: (*Continued.*) Directives

Directive	Description	Section
DEFINE	Defines a file-wide value	Value assignment
DF	Generates a 32-bit floating point constant	Data definition or allocation
DL	Generates a 32-bit constant	Data definition or allocation
.double	Generates 32-bit value	Data definition or allocation
DS	Allocates space for 8-bit bytes	Data definition or allocation
DS16	Allocates space for 16-bit words	Data definition or allocation
DS32	Allocates space for 32-bit words	Data definition or allocation
DS8	Allocates space for 8-bit bytes	Data definition or allocation
DW	Generates 16-bit word constants, including strings	Data definition or allocation
ELSE	Assembles instructions if a condition is false	Conditional assembly
ELSEIF	Specifies a new condition in an IF ENDIF block	Conditional assembly
END	Terminates the assembly of the last module in a file	Module control
ENDIF	Ends an IF block	Conditional assembly
ENDM	Ends a macro definition	Macro processing
ENDMOD	Terminates the assembly of the current module	Module control

Table 3.4: Directives (*Continues.*)

Directive	Description	Section
ENDR	Ends a repeat structure	Macro processing
EQU	Assigns a permanent value local to a module	Value assignment
EVEN	Aligns the program counter to an even address	Segment control
EXITM	Exits prematurely from a macro	Macro processing
EXPORT	Exports symbols to other modules	Symbol control
EXTERN	Imports an external symbol	Symbol control
.float	Generates 48-bit values in TI's floating point format	Data definition or allocation
IF	Assembles intructions if a condition is true	Conditional assembly
IMPORT	Imports an external symbol	Symbol control
LIBRARY	Begins a library module	Module control
LIMIT	Checks a value against limits	Value assignment
LOCAL	Creates symbols local to a macro	Macro processing
LSTCND	Controls conditional assembler listing	Listing control
LSTCOD	Controls multi-line code listing	Listing control
LSTEXP	Controls the listing of macro generated lines	Listing control
LSTMAC	Controls the listing of macro definitions	Listing control
LSTOUT	Controls assembler listing output	Listing control
LSTPAG	Controls the formatting of output into pages	Listing control
LSTREP	Controls the listing of lines generated by repeat directives	Listing control
LSTXRF	Generates a cross-reference table	Listing control
MACRO	Defines a macro	Macro processing
MODULE	Begins a library module	Module control
NAME	Begins a program module	Module control
ODD	Aligns the program location counter to an odd address	Segment control
ORG	Sets the location counter	Segment control
PAGE	Generates a new page	Listing control
PAGSIZ	Sets the number of lines per page	Listing control
PROGRAM	Begins a program module	Module control
PUBLIC	Exports symbols to other modules	Symbol control
PUBWEAK	Exports symbols to other modules, multiple definitions allowed	Symbol control

Table 3.4: (*Continued.)* Directives

Directive	Description	Section
RADIX	Sets the default base	Assembler control
REPT	Assembles instructions a specified number of times	Macro processing
REPTC	Repeats and substitutes characters	Macro processing
REPTI	Repeats and substitutes strings	Macro processing
REQUIRE	Forces a symbol to be references	Symbol control
RSEG	Begins a relocatable segment	Segment control
RTMODEL	Declares runtime model attributes	Module control
SET	Assigns a temporary value	Value assignment
SFRB	Creates byte-access SFR labels	Value assignment
SFRTYPE	Specifies SFR attributes	Value assignment
SFRW	Creates word-access SFR labels	Value assignment
STACK	Begins a stack segment	Segment control
VAR	Assigns a termporary value	Value assignment

instructions in the macro directive to execute the last assembler directive shown on line m. The *.mexit* directive is often used to debug your program. By inserting this directive, the execution flow directly goes to line m, ignoring all instructions and assembler directives between *.mexit* and *.endm*.

The second main purpose of macro directives is that they allow you, a programmer, to access a collection of macro definitions written by you and others. Similar to using libraries in C, you can use libraries of macro directives that have been defined. To include those directives in your program, you need to include the library (a file with macro directive definitions) in your program before using those macros using the following statement:

```
.mlib   filename
```

where filename must be xx.asm and xx.asm contains macro definitions, such as the Invert macro we saw earlier.

3.6.3 ASSEMBLY PROCESS

Once an assembly language program is created, either from a high-level program using the C/C++ programming language through a cross-compiler or by directly writing the assembly language program, the resulting program must once more go through a conversion process. The advantage of being able to write and understand assembly language programs is that we can dictate the clock-by-clock execution of instructions by hardware. Such capabilities are desirable when we are dealing with time critical applications. It is only fair to state that the cross-compiler technologies have come a long way from the late 1980s and today's typical cross-compilers usually generate compact and efficient assembly programs, making it less desirable for aspiring engineers to learn assembly language programming skills. Nevertheless, we still believe both electrical and computer engineers must learn assembly language programming skills to not only program real-time microcontroller algorithms for embedded systems but also to fully understand computer architecture issues, such as instruction sets and instruction execution cycles. In this section, we describe how an assembler converts assembly language instructions into machine language instructions of MSP430.

An assembler takes an assembly program as its input and generates the corresponding machine code. In addition to the machine code, the assembler also produces a list file, a symbol table, and an object code, as shown in Figure 3.9. A list file contains the source program (assembly program) line numbers, the memory locations of instructions, actual machine code for each instruction, and the original assembly instructions. The file is a convenient location to find most of possible problems with your program during a debugging period.

A symbol table is generated to be used by the assembler during the assembly process, which contains, as the name of the file indicates, all symbols used in your assembly program. An object file contains actual machine code and their corresponding memory locations which

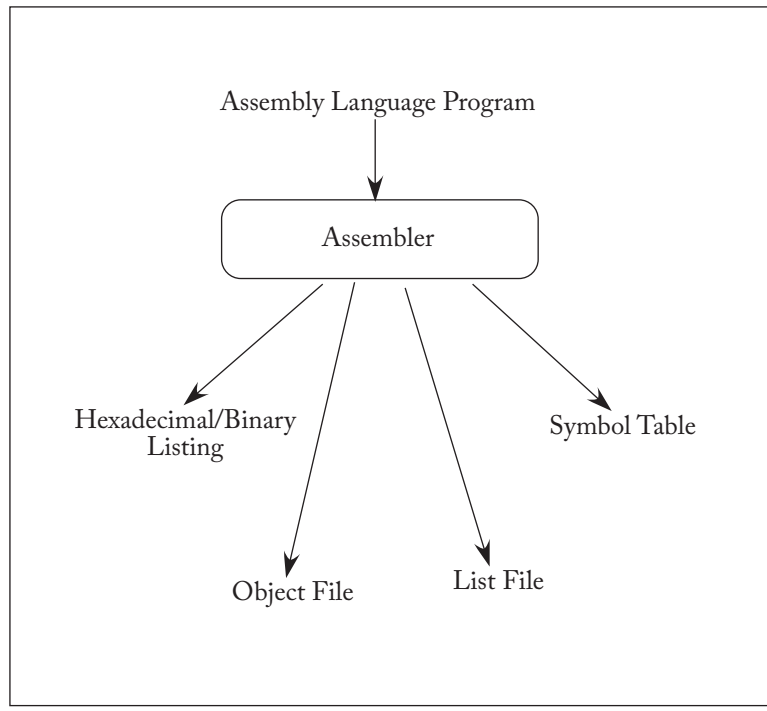


Figure 3.9: Assembly process of MSP430.

is used by a linker program to generate an overall machine code when more than one assembly program is used to generate an overall code. In an object file, there are three separate sections. The first section is executable code designated by the *.text* directive. The second section contains initialized variables which is designated by the *.data* directive, and, finally, the third and last section contains uninitialized variables which are reserved by the *.bss* directive.

The assembly process involves, typically, two phases. During the first phase, an assembler scans through an assembly language program to identify all symbols used, including labels, and computes their values (memory locations, offsets) and stores them to a symbol table in preparation of the second phase where actual machine code is generated using the symbol table obtained during the first phase.

3.6.4 INSTRUCTION SET ARCHITECTURE

The MSP430 microcontroller has 27 different unique instructions and 24 emulated instructions. Emulated instructions are those created by combining unique instructions together to create a new instruction. Each instruction can have one of the following three formats: (1) instructions

requiring two operands, (2) instructions requiring a single operand, and (3) instructions whose operands are relative offsets.

To help you get a handle on the instruction set, we group all MSP430 instructions into one of the following six categories.

1. Data Transfer and Manipulation Instructions – Move and manipulate data.
2. Arithmetic Instructions – Perform arithmetic operations.
3. Logic and Bit instructions – Used to execute logical operations.
4. Data Test Instructions – Test the contents of a memory location or a register.
5. Branch Instructions – Performs IF-THEN-ELSE operations.
6. Function Call Instructions and Others - Initiate or terminate a subroutine or a service routine.

Data Transfer and Manipulation Instructions

Data transfer instructions are used to move data among memory, CPU registers, and the stack. These instructions affect the N and Z lag bits of the R2 (Status Register). Unlike accumulator based microcontrollers, the MSP430's move instructions can directly work with memory contents without loading them into accumulators first. Mnemonics for the MSP430 move instructions are as follows.

- MOV.W/MOV.B – moves a word/byte from a source location to a destination location
- PUSH.W/PUSH.B – moves a word/byte from a source location to the stack
- POP.W/POP.B – moves a word/byte from the stack to a destination location
- CLR.W/CLR.B – moves a zero to a destination location

The manipulation instructions modify the contents of the a register or a memory location. These include arithmetic and logical shift instructions in addition to rotate instructions. A shift instruction brings in a zero and removes an original bit whether the instruction is a logical one or an arithmetic one, except in the case of an arithmetic shift right, which repeats the sign or the MSB. Rotate instructions do not remove any original bits, but they are simply shifted either left or right along with a carry bit, resulting in the original value after eight rotations for a byte number. Figure 3.10 shows both shift and rotate operations of a typical microcontroller.

Arithmetic shift instructions perform multiplications and divisions by the power of 2 for signed numbers. The function of shift operations is a familiar one for us with decimal numbers. Given a decimal number, say 1527, if we shift the digits to the right with respect to the radix point, we get 0152.7. This is the same as dividing the original number by base 10. If we shift to

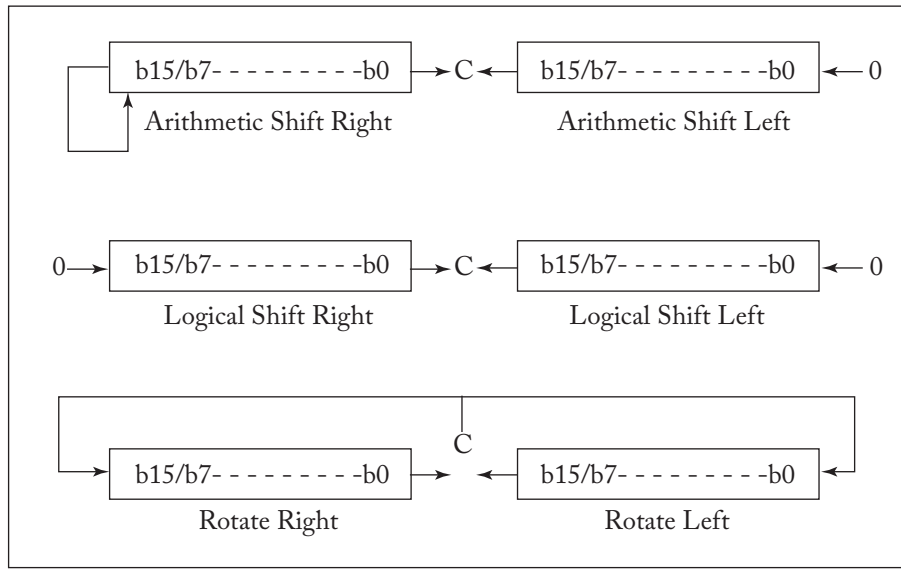


Figure 3.10: Rotate and shift instructions of MSP430.

the left, we get 15,270, or the value obtained by multiplying the original number by base 10. The shift instructions of the MSP430 perform the same task except in the binary number system.

Example: Suppose you start with a signed decimal number 4 represented as an 8-bit number (0000_ 0100 binary). When we perform an arithmetic right shift, the resulting value is decimal number 2 (0000_ 0010 binary). Shift it right again, we have decimal number 1 (0000_ 0001 binary). Thus, each shift operation is equivalent to dividing the original value by 2, and repeating the shift operation n times results in dividing the original number by 2^n.

Example: Suppose now we are working with decimal number -2 (1111_ 1110 binary 2's complement representation). An arithmetic shift left produces decimal number -4 (1111_ 1100). Two more left shifts result in decimal number -16 (1111_0000). Thus, the arithmetic left shift is equivalent to performing multiplication by 2^n where n is the number of left shifts.

As can be seen by the simple examples above, arithmetic shift operations of the MSP430 microcontroller are used to perform multiplications and divisions by a factor of 2^n.

Logical shift instructions are used to test each bit of a byte one at a time. Note from Figure 3.10 that, for logical shift operations, a zero is introduced from the left for a right shift and from the right for a left shift each time a shift operation is performed. We can test each bit of a byte, say by testing the LSB, as we repeat a logical shift right operation eight times. Also, note that once we have tested all eight bits of the byte, we replace them with zeros.

You might ask, where would such operations be useful? It is fairly common to use such operations in multiple applications. One such application is managing appliances in a smart home. As we discussed, one of the main reasons for using a microcontroller over a general-purpose computer is its ability to interface with multiple external systems using built-in I/O interface capabilities via the physical ports. Suppose for our discussion, that one of the MSP430 input ports (eight pins) is connected to eight different appliances in your home, where each input pin is used by the appliances to let the MSP430 know the on/off status of the appliances. Assuming you have the remote access capability of the MSP430, you can login to your controller, read the input port value, perform logical shift operations to check the on/off status of your appliances, and take appropriate actions, if necessary.

Logical shift instructions can also be used to perform multiplication and division operations. In fact, both the arithmetic shift left and the logical shift left operations perform the exact same task and can be interchangeably used to multiply unsigned and signed numbers by a factor of 2. The logical shift right instruction, however, should be used to divide unsigned numbers by a factor of 2, while for signed numbers, the arithmetic shift right instruction should be used.

The rotate instruction should be used to retain the original bit order of your data, but requires shifting bits of the data. For example, suppose you want to swap the high order four bits with the low-order four bits of a byte, say 3Eh.[3] We obtain E3h by rotating the original value four times to either left or right. Designers of the MSP430 have combined the rotate and shift instructions as much as possible.

The available rotate and shift instructions of the MSP430 are

- RLA – rotate left arithmetically, arithmetic shift left
- RRA – rotate right arithmetically, arithmetic shift right
- RLC – rotate left through carry
- RRC – rotate right through carry

The last category of the data manipulation instructions contain two instructions that are used to sign extend a byte and swap bytes. The instructions format are:

- SWPB *src* – swaps the upper and lower bytes of a word
- SXT *src* – sign extends the lower byte to the upper byte of a word

The SXT instruction replaces the upper byte of a word either with 1's (lower byte contains a 2's complement negative number) or 0's (lower byte contains a 2's complement positive number).

[3]The h represents a hexadecimal number.

Arithmetic Instructions

The original MSP430 microcontroller only had addition and subtraction instructions and no multiplication or division operations. The MSP430x5xx controller family includes a separate multiplication module called the MPY32, which you saw earlier in this chapter. For addition and subtraction operations, instructions for binary and binary coded decimal (BCD)[4] operations are possible.

For addition, the following instructions use two operands and store the result into the destination (*dst*) location.

- ADD.W/ADD.B *src, dst* – adds the word/byte in *src* and *dst* and stores the result to *dst*
- ADDC.W/ADDC.B *src, dst* – adds the word/byte in *src* and *dst* along with the carry bit of the SR register (R2) and stores the result to *dst*
- SUB.W/SUB.B *src, dst* – subtracts the word/byte in *dst* from *src* and stores the result to *dst*
- SUBC.W/SUBC.B *src, dst* – subtracts the word/byte in *dst* and the carry bit of the SR register (R2) from *src* and stores the result to *dst*
- DADD.W/DADD.B *src, dst* – adds the word/byte in *src* and *dst* in BCD format along with the carry bit of the SR register (R2) and stores the result to *dst*

The addition and subtraction instructions that only use one operand are

- ADC.W/ADC.B *dst* – adds a carry bit of the SR register (R2) to the word/byte in *dst* and stores the result to *dst*
- INC.W/INC.B *dst* – adds one to the word/byte in *dst* and stores the result to *dst*
- INCD.W/INCD.B *dst* – adds two to the word/byte in *dst* and stores the result to *dst*
- SBC.W/SBC.B *dst* – subtracts a carry bit of the SR register (R2) from the word/byte in *dst* and stores the result to *dst*
- DEC.W/DEC.B *dst* – subtracts one from the word/byte in *dst* and stores the result to *dst*
- DECD.W/DECD.B *dst* – subtracts two from the word byte in *dst* and stores the result to *dst*
- DADC.W/DADC.B *dst* – adds a carry bit of the SR register (R2) to the word/byte in *dst* and stores the result in BCD format to *dst*

[4]The BCD format uses four bits to represent decimal numbers 0–9 instead of all possible numbers four bits can represent: 0–15 decimal or 0–F in hexadecimal. Thus, decimal 37 is represented as 0011_0111 binary ($37).

We discuss the multiplication operation, next. Suppose you want to multiply the following two 8-bit numbers: 78h and 03h. Again, the letter *h* at the end of each number represents hexadecimal numbers. The hexadecimal number system uses 16 numbers instead of 10, as in the decimal system, to represent each number. The numerical numbers used in the hexadecimal number system are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. The corresponding numbers in the decimal system are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, and 15. Thus, 78h represents $(7 * 16^1) + (8 * 16^0)$ or decimal value 280.

The following instructions perform the multiplication operation in the MSP430.

```
MOV.B     #78h, MPY_B      ; load 78h to operand one register
MOV.B     #03h, OP2_B      ; load 03h to operand two register
```

Once the second number is loaded into the second operand register, the multiplication starts. Note that both operands are bytes, hence the use of MPY_B and OP2_B registers. A set of 8 bits makes up a byte. The two numbers used in the example are 0111_1000 (78h) and 0000_0011 (03h). The result is stored in the RESLO register.

As another example, suppose now you want to multiply the following two 32-bit signed numbers: 24184219h and F249E201h. One must write the following code segment using the appropriate 32-bit registers.

```
MOV     #4219h, MPYS32L
MOV     #2418h, MPYS32H
MOV     #E201h, OP2L
MOV     #F249h, OP2H
```

Pay special attention to the order of loading numbers in the example above. You must first load the operand one register then the operand two register. For each register, you must also remember to load the low byte first followed by the high byte. If one of the operand is only 16 bits long, you must write to the high byte of the target register followed by the low byte register, which will indicate to the controller to ignore the high byte and that the operand should be considered as a 16-bit (low byte) long number. The result of the above code will be in RES0-RES3 registers where RES3 contains the most significant word (two bytes) and RES0 contains the least significant word of the result.

Logic and Bit Instructions

The MSP430 microcontroller has a simplified set of logic instructions. The MSP430's instruction set architecture does not include any OR instructions or 2's complement instructions. There are

two AND instructions, but one does not affect the flags in the status register (R2). The two AND instructions have the following format.

- AND.W/AND.B *src, dst* – performs bitwise AND operation of words/bytes in *src* and *dst* and stores the result in *dst*
- BIT.W/BIT.B *src, dst* – performs bitwise AND operation of words/bytes in *src* and *dst*, only the SR flags are affected

Instead of implementing OR instructions, MSP430 designers rely on the bit set and bit clear instructions to perform desired OR operations. These instructions have the following format.

- BIS.W/BIS.B *src, dst* – sets (change value to 1) *dst* bit locations which correspond to bit locations with ones in *src*
- BIC.W/BIC.B *src, dst* – clears (change value to 0) *dst* bit locations which correspond to bit locations with ones in *src*

How do we perform an OR operation? We can configure the bits in *src*, say to all 1's, and perform the *bis* instruction to perform an OR instruction. Similarly, we can perform the same OR instruction with all bits in *src* to be zeros and executing the *bic* instruction. The exclusive OR operation is performed using the following instruction.

- XOR *src, dst* – performs bitwise exclusive OR operation with words/bytes in *src* and *dst* and stores the result to *dst*

The MSP430 controller also has an invert instruction that flips all bits in a word or a byte.

- INV.W/INV.B *dst* – invert each bit in *dst*

We now consider MSP430 bit manipulation instructions. Most bit manipulation instructions, besides the *bis* and *bic* instructions, are for dealing with flags in the status register (SR, R2) as shown below. Note that all these instructions do not require any operand.

- SETC – sets the carry bit (flag) in SR
- SETN – sets the negative bit (flag) in SR
- SETZ – sets the zero bit (flag) in SR
- EINT – sets (enables) the interrupt bit (flag) in SR
- CLRC – clears the carry bit (flag) in SR
- CLRN – clears the negative bit (flag) in SR
- CLRZ – clears the zero bit (flag) in SR
- DINT – clears (disables) the interrupt bit (flag) in SR

Data Test Instructions

There are only two test instructions with the MSP430 microcontroller instruction set. The two instructions directly affect the flags in the SR register, allowing instructions that immediately follow the test instructions to use the contents of the SR register to determine the flow of program execution. The two instructions have the format shown below.

- CMP.W/CMP.B *src*, *dst* – compares the values in *src* and *dst* and changes the bits in SR accordingly
- TST.W/TST.B *dst* – compares the word/byte in *dst* with zero and affects the changes to bits in SR accordingly

Flow Control Instructions

The following instructions are designed to allow programmers to change the flow of instruction execution. The flow change can be pre-determined as in the case of calling a subroutine or dynamic as in the result of some computation or inputs during programming execution.

Branch Instructions

Branch instructions are used to implement IF-THEN-ELSE programming constructs. There are 11 such instructions in the MSP430 microcontroller instruction set. Typically, these instructions can be further divided into branch instructions designed to deal with signed numbers and ones for unsigned numbers. Branch instructions for signed numbers are as follows.

- JN *label* – jumps to a location specified by the *label* if the Negative bit in the Status Register (SR) is set
- JGE *label* – jumps to a location specified by the *label* if the test/comparison/operation made immediately before the current instruction execution results in a signed number greater than or equal to zero
- JL(t) *label* – jumps to a location specified by the **label** if the test/comparison/operation made previously results in a signed number less than zero

Branch instructions for unsigned numbers are as follows.

- JHS *label* – jumps to a location specified by the **label** if the test/comparison/operation made previously results in an unsigned number higher than or same as zero
- JLO *label* – jumps to a location specified by the **label** if the test/comparison/operation made previously results in an unsigned number lower than zero

Other available branch instructions are as follows.

- JZ *label* – jumps to a location specified by the **label** if the zero bit in SR is set

- JNZ *label* – jumps to a location specified by the **label** if the zero bit in SR is not set
- JC *label* – jumps to a location specified by the **label** if the carry bit in SR is set
- JNC *label* – jumps to a location specified by the **label** if the carry bit in SR is clear
- JEQ *label* – jumps to a location specified by the **label** if the test/comparison/operation preceding the instruction results in zero
- JNE *label* – jumps to a location specified by the **label** if the test/comparison/operation preceding the instruction results in non-zero value

Function Call Instructions and Others

In this section, we present those instructions associated with calling subroutines and service routines.[5]

- CALL *src* – calls a subroutine with label *src*
- RET – returns from a subroutine
- RETI – returns from an interrupt service routine

The MS430 microcontroller also has a jump instruction that allows the program execution flow to be changed: JMP *label*. The instruction informs the program counter to change its contents to the address specified by *label*. Why bother to use this instruction when we have the branch always instruction *BR*? The JMP instruction is more compact as the instruction fits in a single word, which is important when we want to minimize the use of controller memory. The jump range, however, is limited to 1 K bytes from the location of the JMP instruction.

One more instruction of interest is the no operation instruction, *NOP*, used without any operand. The instruction takes up one clock cycle to execute and is used for debugging of a program or to fill up time to meet timing requirements for an application.

Examples: The JMP and RET instructions are used to jump to a subroutine and return from it. A subroutine is a collection of assembly language instructions used to perform a common task. For example, we might want to develop a subroutine to perform an ADC conversion.

The subroutine uses the following format:

```
 :
 :
JMP          subroutine1
 :
 :
```

[5] A service routine is set of instructions similar to the ones that make up a subroutine except these instructions are written specifically to respond to interrupts (special hardware and software requests).

```
subroutine1:    :
                :
                :
               RET
```

Provided in Figure 3.11 is a summary of the MSP430 assembly language instruction set.

3.7 ADDRESSING MODES

Earlier, we mentioned that each assembly language instruction has an opcode and an operand. The opcode tells hardware which operation should be executed. The operand specifies how the data necessary to execute the particular operation can be found. Addressing modes are different methods used to identify necessary data for assembly instructions. There are seven different addressing modes in the MSP430 microcontroller. We present them next.

3.7.1 REGISTER ADDRESSING MODE

For this addressing mode, the data needed to executean instruction are the contents of registers. For example, once the following instruction is executed,

```
MOV.W  R5,R7
```

the contents of register R5 are moved to register R7.

In the following instruction

```
AND.B     R5,R7
```

to execute the AND operation, the MSP430 takes the contents of R5 and R7 and performs bit wise AND operation and stores the result in R7.

3.7.2 INDEXED ADDRESSING MODE

In this addressing mode, the address of the data necessary for an instruction is found by adding an offset value to the contents of a register. For example, suppose you have the following instruction.

```
ADD.W   5(R4), R5
```

The instruction takes the data located at the address specified by 5 plus the contents of R4, adds the value to the value in R5 and stores the result to R5.

Mnemonic		Description		V	N	Z	C
ADC(.B)†	dst	Add C to destination	dst + C → dst	*	*	*	*
ADD(.B)	src,dst	Add source to destination	src + dst → dst	*	*	*	*
ADDC(.B)	src,dst	Add source and C to destination	src + dst + C → dst	*	*	*	*
AND(.B)	src,dst	AND source and destination	src .and. dst → dst	0	*	*	*
BIC(.B)	src,dst	Clear bits in destination	.not.src .and. dst → dst	–	–	–	–
BIS(.B)	src,dst	Set bits in destination	src .or. dst → dst	–	–	–	–
BIT(.B)	src,dst	Test bits in destination	src .and. dst	0	*	*	*
BR†	dst	Branch to destination	dst → PC	–	–	–	–
CALL	dst	Call destination	PC+2 → stack, dst → PC	–	–	–	–
CLR(.B)†	dst	Clear destination	0 → dst	–	–	–	–
CLRC†		Clear C	0 → C	–	–	–	0
CLRN†		Clear N	0 → N	–	0	–	–
CLRZ†		Clear Z	0 → Z	–	–	0	–
CMP(.B)	src,dst	Compare source and destination	dst – src	*	*	*	*
DADC(.B)†	dst	Add C decimally to destination	dst + C → dst (decimally)	*	*	*	*
DADD(.B)	src,dst	Add source and C decimally to dst.	src + dst + C → dst (decimally)	*	*	*	*
DEC(.B)†	dst	Decrement destination	dst – 1 → dst	*	*	*	*
DECD(.B)†	dst	Double-decrement destination	dst – 2 → dst	*	*	*	*
DINT†		Disable interrupts	0 → GIE	–	–	–	–
EINT†		Enable interrupts	1 → GIE	–	–	–	–
INC(.B)†	dst	Increment destination	dst +1 → dst	*	*	*	*
INCD(.B)†	dst	Double-increment destination	dst+2 → dst	*	*	*	*
INV(.B)†	dst	Invert destination	.not.dst → dst	*	*	*	*
JC/JHS	label	Jump if C set/Jump if higher or same		–	–	–	–
JEQ/JZ	label	Jump if equal/Jump if Z set		–	–	–	–
JGE	label	Jump if greater or equal		–	–	–	–
JL	label	Jump if less		–	–	–	–
JMP	label	Jump	PC + 2 x offset → PC	–	–	–	–
JN	label	Jump if N set		–	–	–	–
JNC/JLO	label	Jump if C not set/Jump if lower		–	–	–	–
JNE/JNZ	label	Jump if not equal/Jump if Z not set		–	–	–	–
MOV(.B)	src,dst	Move source to destination	src → dst	–	–	–	–
NOP†		No operation		–	–	–	–
POP(.B)†	dst	Pop item from stack to destination	@SP → dst, SP+2 → SP	–	–	–	–
PUSH(.B)	src	Push source onto stack	SP – 2 → SP, src → @SP	–	–	–	–
RET†		Return from subroutine	@SP → PC, SP + 2 → SP	–	–	–	–
RETI		Return from interrupt		*	*	*	*
RLA(.B)†	dst	Rotate left arithmetically		*	*	*	*
RLC(.B)†	dst	Rotate left through C		*	*	*	*
RRA(.B)	dst	Rotate right arithmetically		0	*	*	*
RRC(.B)	dst	Rotate right through C		*	*	*	*
SBC(.B)†	dst	Subtract not(C) from destination	dst + 0FFFFh + C → dst	*	*	*	*
SETC†		Set C	1 → C	–	–	–	1
SETN†		Set N	1 → N	–	1	–	–
SETZ†		Set Z	1 → C	–	–	1	–
SUB(.B)	src,dst	Subtract source from destination	dst + .not.src + 1 → dst	*	*	*	*
SUBC(.B)	src,dst	Subtract source and not(C) from dst.	dst + .not.src + C → dst	*	*	*	*
SWPB	dst	Swap bytes		–	–	–	–
SXT	dst	Extend sign		0	*	*	*
TST(.B)†	dst	Test destination	dst + 0FFFFh + 1	0	*	*	1
XOR(.B)	src,dst	Exclusive OR source and destination	src .xor. dst → dst	*	*	*	*

† Emulated Instruction

Figure 3.11: Texas Instruments MSP430 assembly language instruction set [SLAU056L, 2013]. Used with permission of Texas Instruments.

3.7.3 SYMBOLIC ADDRESSING MODE

To those who are familiar with other microcontrollers' addressing modes, this addressing mode is similar to a program counter (PC) relative addressing mode. The data necessary for an instruction is found by finding the relative offset from the current instruction to a destination location. The contents of the source address (contents of PC + X) are moved to the destination address (contents of PC + Y).

3.7.4 ABSOLUTE ADDRESSING MODE

In this addressing mode, the contents of a memory address are used as the data necessary for an instruction. For example, suppose we have

```
MOV.W    ABC, R10
```

The instruction moves the contents of address ABC to register R10.

3.7.5 INDIRECT REGISTER ADDRESSING MODE

In this addressing mode, the contents of an address are used as the address where data for the instruction is found.

For example,

```
MOV.W    @R5, R10
```

moves the contents of address location, whose address value was determined by the contents of R5, are moved to register R10.

3.7.6 INDIRECT AUTOINCREMENT ADDRESSING MODE

In this addressing mode, the instruction first performs its task using the register indirect addressing mode and then increments the contents of the register by either one for a byte operation and two for a word operation.

```
MOV.B    @R4+,R5
```

The above instruction takes the contents of R4, uses it as the address location to find the data, moves the data to R5, then increments the contents of R4 by one.

3.7.7 IMMEDIATE ADDRESSING MODE

In this addressing mode, the actual number specified by symbol # is used by an instruction.

```
mov.w  #78F2h, R4
```

This instruction moves 78F2h to R4 where h represents a hexadecimal number.

Examples: Assume that the following instructions are not sequential. Given that the contents of R5 is $(128)_{10}$ and R6 is CAFEh, describe the results of each instruction.

1. ADD #10, R5

   Answer: Add 10 to the contents of register R5.

2. ADD.B #10, R5

   Answer: Add 10 to the low byte of R5.

3. MOV #AA55h, R4

   Answer: Places AA55h into register R4.

4. AND R5, R4 (use R4 from the previous example)

   Answer: Performs the bit-wise AND of R5 and R4, placing the result in R4 0000h.

5. BIC #FC00h, R6

   Answer: Clears the 6 MSBs of R6 resulting in 02FEh.

6. BIS #FC00h, R6

   Answer: Sets the 6 MSBs of R6 resulting in FEFEh.

7. CLR R5

   Answer: Set R5 to zero.

8. CLR.B R6

   Answer: Clears the low byte of R6 resulting in a value of CA00h.

3.7.8 PROGRAMMING CONSTRUCTS

Programming constructs determine the flow of executing a program. For any program, there can be three principle programming constructs: a sequence, a loop, and a branch. A sequence represents a segment of a program, regardless of the programming language used, where instructions are executed in sequence, one after another in the order shown. A loop represents a segment of a program where the same instructions are executed a number of times specified by some condition. Finally, a branch allows programmers to implement IF-THEN-ELSE decisions in their programs.

The three principle constructs are shown in Figure 3.12. How these three programming constructs are put together to write programs is the key to becoming a good programmer.

Example: Provided below is the basic assembly language construct for a loop.

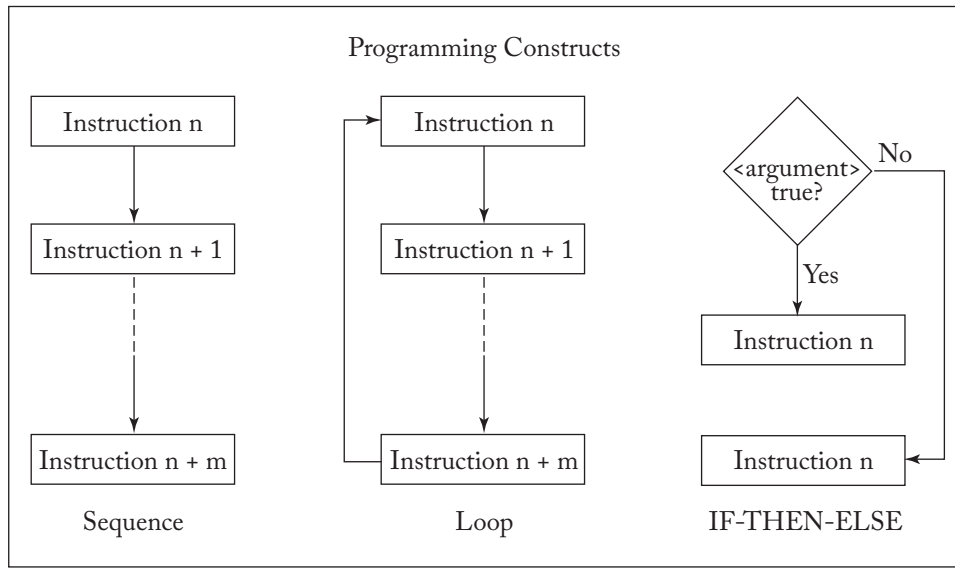


Figure 3.12: Programming constructs.

```
Loop1:          :
                :
                :
                JMP Loop1
```

Example: Provided below is the basic assembly language construct for the if-then statement.

```
                :
                :
                CMP  R6, R7    ;R6 = R7:
                JEQ  EQUAL1
                :

EQUAL1:         :
                :
```

3.7.9 ORTHOGONAL INSTRUCTION SET

Over the history of the computer industry, there have been two different approaches in designing a CPU: the CISC, and the RISC. The rationale for creating a CISC based computer is to increase computer performance by allowing programmers to write compact programs. This means hardware is designed to accommodate a set of specialized instructions, allowing a pro-

grammer to pick and choose appropriate instructions to perform a required task. In contrast, RISC computers are designed to execute only a small number of (thus, reduced) instructions. The related hardware is simple, which allows instructions to be executed in parallel, increasing the performance of a RISC-based computer. Over the past three decades the battle between the two approaches has been intense, but over time, computer designers have extracted advantages of both approaches to create hybrid computers. Most of today's computers incorporate both CISC and RISC features.

The MSP430 architects wanted to design and develop a RISC-based controller. As mentioned, the controller has 27 unique instructions and 9 emulated instructions. In addition to the RISC-based design, MSP430 designers maximized the number of functional instructions by developing instructions that are orthogonal. The orthogonal instruction set means that each opcode (instruction) can use any of the MSP430's available addressing modes. his provides the programmer with tremendous flexibility in writing the code.

3.8 SOFTWARE PROGRAMMING SKILLS

In this section, we present a system design approach called top-down design and bottom-up implementation, which can be used to systemically plan and execute programming tasks. The overall idea of this approach is to break down a given task (your program should be implemented to execute the task) into smaller pieces or subtasks, solve each of the smaller subtasks, and then integrate them together one at a time. Of course, if any smaller piece is too big, you should repeat the process until the subtask at the lowest level is defined by a set of simple operations.

The top-down approach is sometimes called the *divide-and-conquer* method with two immediate benefits. The first one is that you reduce the complexity of the overall task by only concentrating on a simple subtask at a time. The second advantage is the by-product of the first one: test and evaluation of smaller tasks are easy and time saving. In addition to the two advantages, this approach makes it easy to integrate the subtask solutions, making the overall efforts and time spent to perform the original task of writing a program minimal.

We use two powerful tools to implement this approach: structure charts and UML activity diagrams.

- **UML activity diagrams:** A unified modeling language (UML) activity diagram, or flow chart, is a tool to help visualize the different steps required for a control algorithm.
- **Structure chart:** A structure chart is a visual tool used to partition a large project into "doable" smaller parts. The arrows within the structure chart indicate the data flow between different portions of the program.

We conclude this section with a step-by-step procedure to use the top-down design and bottom-Up implementation approach.

1. Given a task statement, write down all subtasks necessary to complete the task.

2. Place the subtasks in a structure chart.
3. For each subtask, repeat step one if the subtask is too big. Stop when the subtasks cannot be further subdivided.
4. Place each subtask on the structure chart in proper hierarchy relative to the other sub tasks.
5. Draw the UML activity diagram for each subtask showing enough detail to write instructions to accomplish the task.
6. Write a program segment to fulfill the subtask, proceeding to the next one only after you have thoroughly tested the functionality of the subtask program segment.
7. Integrate subtask segments one at a time, testing their combined functionality (this may require a test plan).
8. Continue this process until all subtask solution segments are integrated and the overall task is completed.

Example: In Chapter 2 we introduced the Adafruit mini round robot.

In this example we revisit the robot control algorithm to illustrate the use of the structure and UML activity diagram to guide development of a more sophisticated control algorithm.

The UML activity diagram for the robot is provided in Figure 3.14. As you can see, after robot systems are initialized, the robot control system enters a continuous loop to gather data and issue outputs to steer the robot through the maze.

The structure chart for the robot project is provided in Figure 3.13. As you can see, the robot has three main systems: the motor control system, the sensor system, and the digital I/O system. These three systems interact with the main control algorithm to allow the robot to autonomously (by itself) navigate through the maze by sensing and avoiding walls.

3.9 ASSEMBLY VS. C

Due to limited onboard resources, including relatively small memory, microcontrollers do not support the sophisticated software architecture of microprocessors found onboard laptops and PCs. Namely, typical microcontrollers do not have operating systems running onboard.

The actual programs that run onboard used to be written using an assembly programming language rather than high-level languages used for laptop/desktop computers. Since the early 1990s, however, the use of high-level languages, mainly C, increased for microcontrollers due to emerging compiler technologies that allowed programmers to be removed from learning the particular hardware and machine-level software architecture of a microcontroller. The compiler technologies, however, did not produce compact assembly code when compared to an assembly program written by a well-trained assembly programmer. Due to the advantages a high-level

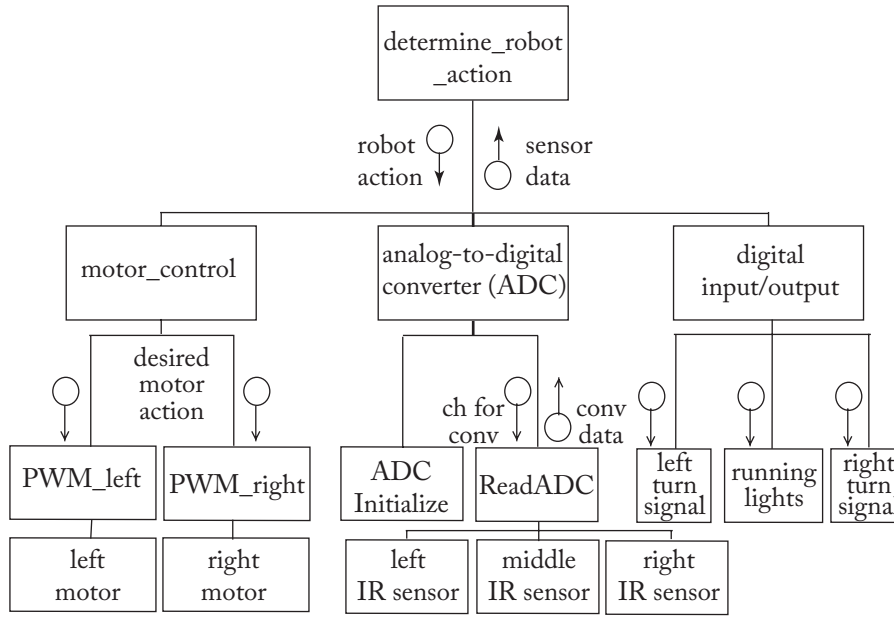


Figure 3.13: Mini round robot structure diagram.

language offers, discussed in the next section, the compiler technologies continued to improve. Today, the difference between a converted assembly program initially written in C and an assembly program written directly is very small.

Advantages/Disadvantages

So why should anyone learn to write assembly language programs? After all, high-level languages, in particular, the C language, are programmer friendly, portable, and compact. Furthermore, most programmers are already familiar with high-level languages, removing the time required to learn a new assembly language. A C program can be machine independent (portability).[6] Although programs usually become more compact when written in an assembly language, the primary reason for most microcontroller programmers who choose to use a high-level language is that a programmer does not need to understand the Instruction Set Architecture (ISA) to write programs for a particular platform, while having access to bit-by-bit level instructions, if necessary.

The proponents of assembly language programs point out the advantage of writing programs with instructions that can directly map to designed functions of a hardware platform. That is, each instruction is written as a specific machine level instruction, allowing a programmer to have full control of the execution of those instructions in the use of time (clock cycle by clock

[6]This statement is marginally true if specific ports and registers for a platform are not used in a program.

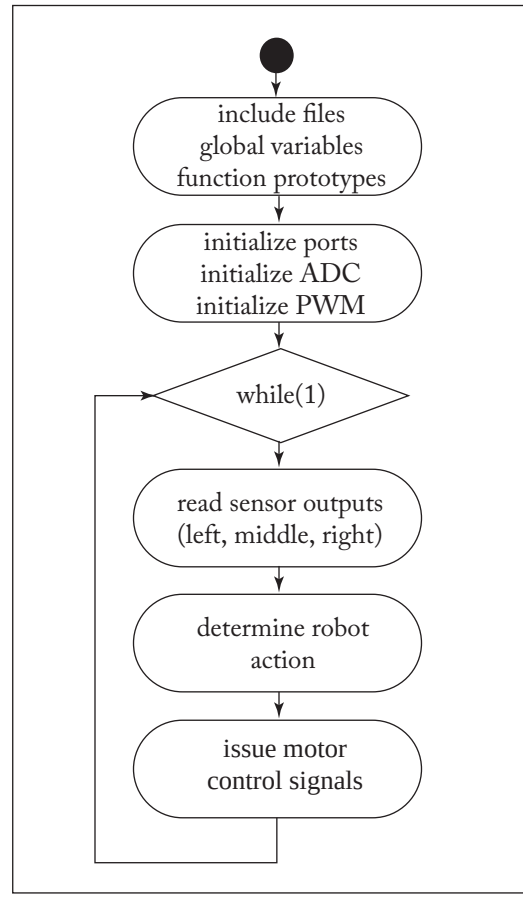


Figure 3.14: Robot UML activity diagram.

cycle) and hardware resources. This leads to writing an efficient program compared to the one written with a high-level language and later converted to assembly code.

Writing programs at the level of machine language also allows programmers control over where his/her programs will reside in memory, optimizing the use of available memory of a microcontroller. Writing assembly language programs also allow programmers to embed pertinent error messages in their programs at the machine level, while high-level language programmers do not have the same option. Finally, programming at the machine specific ISA level allows assembly language programmers to better understand related computer architecture issues, which enables them to take full advantages of the particular hardware and software features of each microcontroller.

3.9.1 OUR APPROACH

Acknowledging the advantages offered by both high-level languages and assembly languages, we use the C language in this book. Once we establish the necessary foundation of the hardware and software systems, we use the C programming language to concentrate on functional capabilities of the MSP430 microcontrollers. Many of the examples provided in the book, written in C, are also available in assembly language within MSPWare (`www.ti.com`).

3.10 ACCESSING AND DEBUGGING TOOLS

There are three different ways to download programs to, interact with, and debug programs on MSP430. The first one uses the built-in eZ-FET Debug Probe resident onboard both the MSP-EXP430FR2433 and the MSP-EXP430FR5994. The second and third methods use joint test action group (JTAG) interface, which uses either four wire signals (conventional) or two wire signals (Spy-Bi-Wire). Most of the MSP430 boards contain JTAG interfaces.

3.11 LABORATORY EXERCISE: PROGRAMMING THE MSP430 IN ASSEMBLY LANGUAGE

In this laboratory exercise we will complete three separate programs:

- flash an LED on the MSP-EXP430FR2433 evaluation board,
- illuminate an LED on the MSP-EXPFR2433 evaluation board when a switch is depressed, and
- perform some mathematical operations and observe the flag values generated.

3.11.1 PART 1: FLASH AN LED VIA ASSEMBLY LANGUAGE

Introduction. In Chapter 2, we illuminated the LEDs on the MSP-EXP430FR2433 evaluation board using a C program. In this laboratory exercise, we employ assembly language to illustrate how to configure, assemble, and execute a program. This basic program may serve as a template for writing future assembly language programs.

Background. The MSP-EXP430FR2433 experimenter board is equipped with two switches (S1 and S2) and two LEDs (LED1 and LED2). These components are hardwired to the following pins on the MSP430 microcontroller (MSP-EXP430FR2433 evaluation board).

- Switch S1, P2.3, switch S1 is active low
- Switch S2, P2.7, switch S2 is active low
- LED1, P1.0, requires logic 1 to illuminate LED

- LED2, P1.1, requires logic 1 to illuminate LED

There are several registers associated with each port.

- PxIN, Port x input. This register is read only and is used to determine the current value of the specified port. For example, to read Port 2, you would read register P2IN.
- PxOUT, Port x output. This is the output register for the specified port. When a value is written to this register, the value written appears at the specified port pins.
- PxDIR, Port x direction register. This is the pin direction register for the specified port. Each port pin has internal interface hardware which is used to configure each pin as an input pin or an output pin. Setting the PxDIR register for a specific port pin to a logic 1 configures the pin as an output pin. A logic 0 configures the pin as an input pin.
- PxSEL, Port x function select. Many pins on the MSP430 microcontroller have alternate functions besides their general purpose I/O function. A logic 0 configures the pin for general purpose I/O, whereas a logic 1 connects the pin to its alternate function.
- PxREN, Port x Pull-up/Pull-down resistor enable register. A logic 0 disables the pull-up/pull-down resistor for the corresponding pin; whereas, a logic 1 connects the pin to a pull-up/pull-down resistor as shown in Figure 3.15.

PxDIR	PxREN	PxOUT	I/O Configuration
0	0	x	Input
0	1	0	Input with pull-down resistor
0	1	1	Input with pull-up resistor
1	x	x	Output

Figure 3.15: MSP430 port configuration registers.

Provided below is an assembly language program to flash an LED. Execute the code on the MSP-EXP430FR2433 evaluation board. To assemble the code, use the same procedure as that used for the C program. An abbreviated version of the steps is provided here for convenience (adapted from SLAU157AP [2017]).

1. Plug the MSP-EXP430FR2433 LaunchPad into the host PC via the USB cable.
2. Start up Code Composer Studio (CCS)
3. Select File – > New – > CCS Project

4. Select Target Family (MSP430FRxxx) and Target (MSP430FR2433).
5. Select TI MSP430 USB1 [Default] and press [Identify].
6. Enter the project name, click Next.
7. Check "Configure as an assembly only project."
8. Click Finish.
9. Select File – > New – > Source File.
10. Enter a file name with the suffix .asm.
11. Type or paste program text into the file.
12. Select Project – > Build Project.
13. Select Run – > Debug to program the MSP430FR2433 memory.
14. Select Run – > Resume to start the program.
15. Important step: Select Run – > Terminate to properly stop the program.

```
; ********************************************************************
;                       MSP430 CODE EXAMPLE DISCLAIMER
;MSP430 code examples are self-contained low-level programs that
;typically demonstrate a single peripheral function or device feature
;in a highly concise manner. For this the code may rely on the
;device's power-on default register values and settings such as the
;clock configuration and care must be taken when combining code from
;several examples to avoid potential side effects. Also see
;www.ti.com/grace for a GUI- and www.ti.com/msp430ware for an API
;functional library-approach to peripheral configuration.
;
; --/COPYRIGHT--
;********************************************************************
;MSP430FR243x Demo - Toggle P1.0 using software
;
;Description: Toggle P1.0 every 0.1s using software.
;By default, FR413x select XT1 as FLL reference.
;If XT1 is present, the XIN and XOUT pin needs to configure.
;If XT1 is absent, switch to select REFO as FLL reference
;automatically.
;  XT1 is considered to be absent in this examples.
```

```
;  ACLK = default REFO ~32768Hz, MCLK = SMCLK = default DCODIV ~1MHz.
;
;           MSP430FR2433
;         ---------------
;     /|\|               |
;      | |               |
;      --|RST            |
;        |           P1.0|-->LED
;
;Cen Fang
;Texas Instruments Inc.
;June 2013
;Built with Code Composer Studio v6.0
;*******************************************************************
            .cdecls C,LIST,"msp430.h"  ;Include device header file
;-------------------------------------------------------------------
            .def    RESET              ;Export program entry-point to
                                       ;make it known to linker.
;-------------------------------------------------------------------
            .global __STACK_END
            .sect   .stack             ;Make stack linker segment known

            .text                      ;Assemble to Flash memory
            .retain                    ;Ensure current section linked
            .retainrefs

RESET       mov.w   #__STACK_END,SP          ;Initialize stack pointer
StopWDT     mov.w   #WDTPW+WDTHOLD,&WDTCTL   ;Stop WDT
SetupP1     bic.b   #BIT0,&P1OUT             ;Clear P1.0 output
            bis.b   #BIT0,&P1DIR             ;P1.0 output
            bic.w   #LOCKLPM5,PM5CTL0        ;Unlock I/O pins

Mainloop    xor.b   #BIT0,&P1OUT             ;Toggle P1.0 every 0.1s
Wait        mov.w   #50000,R15               ;Delay to R15
L1          dec.w   R15                      ;Decrement R15
            jnz     L1                       ;Delay over?
            jmp     Mainloop                 ;Again

;-------------------------------------------------------------------
```

```
;           Interrupt Vectors
;-----------------------------------------------------------------------
            .sect   RESET_VECTOR            ;MSP430 RESET Vector
            .short  RESET                   ;
            .end

;*******************************************************************
```

3.11.2 PART 2: ILLUMINATE A LED VIA ASSEMBLY LANGUAGE

Provided in Figure 3.16 is the UML activity diagram for this portion of the laboratory assignment. Also, the assembly language code is provided that will illuminate LED1 when switch S1 is depressed. Review the UML activity diagram and the assembly language code.

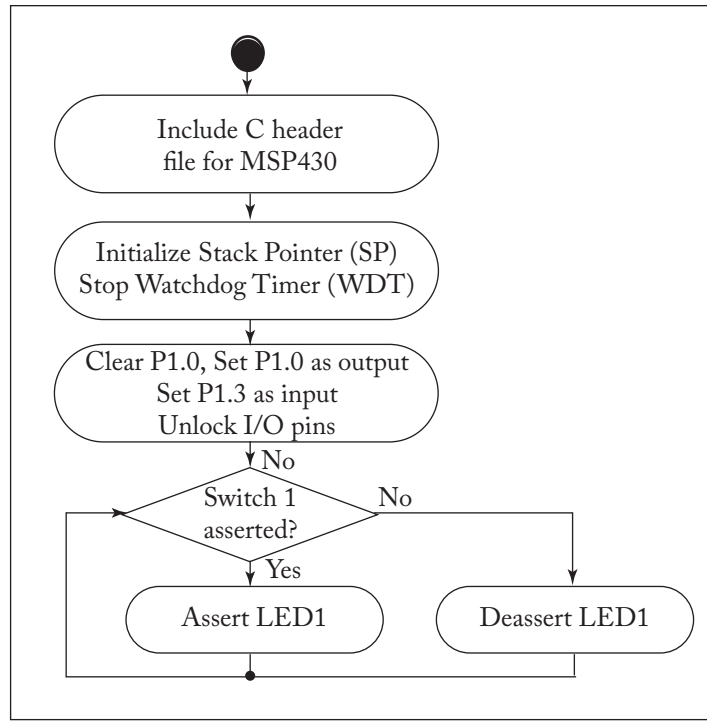


Figure 3.16: UML activity diagram for laboratory part 2.

```
;*******************************************************************
;                       MSP430 CODE EXAMPLE DISCLAIMER
; --/COPYRIGHT--
;*******************************************************************
```

```
;MSP430FR243x Demo - Software Poll P1.3, Set P1.0 if P1.3 = 1
;
;Description: Poll P1.3 in a loop. Set P1.0 if P1.3 = 1, or reset P1.0.
;By default, FR413x select XT1 as FLL reference.
;If XT1 is present, the XIN and XOUT pin needs to be configured.
;If XT1 is absent, REFO is automatically switched for FLL reference.
;XT1 is considered to be absent.
;  ACLK = default REFO ~32768Hz, MCLK = SMCLK = default DCODIV ~1MHz
;
;              MSP430FR2433
;            ---------------
;        /|\|               |
;         | |               |
;         --|RST            |
;     /|\   |               |
;      --o--|P1.3       P1.0|-->LED
;     \|/   |               |
;           |               |
;
;Cen Fang
;Texas Instruments Inc.
;June 2013
;Built with Code Composer Studio v6.0
;*******************************************************************
;-------------------------------------------------------------------
            .cdecls C,LIST,"msp430.h"    ;Include device header file
;-------------------------------------------------------------------
            .def    RESET                ;Export program entry-point
                                         ;to make it known to linker
;-------------------------------------------------------------------
            .global __STACK_END
            .sect   .stack               ;Make stack linker segment
                                         ;

            .text                        ;Assemble to Flash memory
            .retain                      ;Ensure current section gets
                                         ;linked
            .retainrefs
```

```
RESET       mov.w   #__STACK_END,SP       ;Initialize stackpointer
            mov.w    #WDTPW+WDTHOLD,&WDTCTL ;Stop WDT - SET BREAKPOINT HERE

SetupP1     bic.b   #BIT0,&P1OUT          ;Clear P1.0 output
            bis.b   #BIT0,&P1DIR          ;P1.0 output
            bic.b   #BIT3,&P1DIR          ;Set P1.3 as inputs
            bic.w   #LOCKLPM5,PM5CTL0     ;Unlock I/O pins

Mainloop    bit.b    #BIT3,&P1IN
            jz       Clear
            bis.b    #BIT0,&P1OUT
            jmp      Mainloop
Clear       bic.b    #BIT0,&P1OUT
            jmp      Mainloop
            nop

;-------------------------------------------------------------------
;           Interrupt Vectors
;-------------------------------------------------------------------
            .sect   RESET_VECTOR          ;MSP430 RESET Vector
            .short  RESET                 ;
            .end

;*******************************************************************
```

Execute the code on the MSP-EXP430FR2433 evaluation board. Then modify the program to illuminate LED1 when switch S1 is depressed, illuminate LED2 when switch S2 is depressed, and exit the program when switch S1 and S2 are depressed simultaneously. Provide a UML activity diagram of the modified program in your lab notebook.

3.11.3 PART 3: MATHEMATICAL OPERATIONS IN ASSEMBLY LANGUAGE

In this portion of the laboratory, we execute and observe a number of mathematical operations.

Background. In your lab notebook, provide a brief definition of status register bits V, N, Z, and C.

Procedure:

1. For each of the operations listed below, predict the final result and the value of flags V, N, Z, and C.

2. Perform the following operations in assembly language. At least three different addressing modes must be employed.

   (a) Place (CAFE)h into R12. Rotate R12 two bits to the right.
   (b) Place (CAFE)h into R12. Rotate R12 two bits to the left.
   (c) Perform (CAFE)h + (DABA)h.
   (d) Perform (CAFE)h - (DABA)h.
   (e) Perform (CA)h x (FE)h.
   (f) Place (CAFE)h into R12. Clear even bits.
   (g) Place (CAFE)h into R12. Clear odd bits.
   (h) Place (CAFE)h into R12. Set even bits.
   (i) Place (CAFE)h into R12. Set odd bits.
   (j) Place (CAFE)h into R12. Increment R12.

3. Compare predicted and actual results. There are several features with Code Composer Studio (CCS) that will prove very helpful in this step.

   - Register contents may be viewed by selecting **View** – > **Registers**.
   - You can single step through the program by selecting **Target** – > **Assembly Step Into**.
   - The single step progress can be viewed in the **Disassembly** window within the CCS main screen.

3.12 SUMMARY

In this chapter, we introduced a number of fundamental concepts and tools used to program MSP430 microcontrollers. These include the programming model which allows programmers to "see" the current operation status of the controller, a number of power-saving operating modes, the hardware and software organizations of the controller, the instruction set architecture (and available instructions), directives which are the instructions to assemblers, and the overall assembly process. We also presented good software programming skills called top-down design and bottom-up implementation and the use of high-level and assembly languages when programming embedded controllers such as MSP430.

3.13 REFERENCES AND FURTHER READING

68HC12 CPU12 Reference Manual (CPU12 RM/AD Rev 1), Motorala, 1997. 95

Barrett, S. F. *Arduino Microcontroller: Processing for Everyone!*, Morgan & Claypool Publishers, 2010. DOI: 10.2200/s00522ed1v01y201307dcs043.

Code Composer Studio™ v7.x for MSP430™ User's Guide, (SLAU157AP), Texas Instruments, 2017. 129

Hamann, J. C. EE2390 Laboratory Manual, *Department of Electrical and Computer Engineering*, University of Wyoming. 95

Miller, G. H. *Microcomputer Engineering*, Prentice Hall, Englewood Cliffs, NJ, 1995. 95

Morris Mano, M. *Digital Design*, 3rd ed., Prentice Hall, Upper Saddle River, NJ, 2002. 95

MSP430 Assembly Language Tools User's Guide, (SLAU131R), Texas Instruments, 2018. 101

MSP430FR2433 Mixed-Signal Microcontroller, (SLASE59D), Texas Instruments, 2018. 83, 86

MSP430FR4xx and MSP430FR2xx Family User's Guide (SLAU445G), Texas Instruments, 2016. 88

MSP430FR58xx, MSP430FR59xx, and MSP430FR6xx Family User's Guide (SLAU367O), Texas Instruments, 2017. 85, 89

MSP430FR599x, MSP430FR596x Mixed-Signal Microcontrollers, (SLASE54C), Texas Instruments, 2018. 83, 86

Texas Instruments MSP430x4xx Family User's Guide "RISC 16-Bit CPU" (SLAU056L), Texas Instruments, 2013. 96, 100, 120

Texas Instruments MSP430x5xx/MSP430x6xx Family User's Guide, (SLAU208Q), Texas Instruments, 2018. 94, 99

3.14 CHAPTER PROBLEMS

Fundamental

1. What is the function of the SP, PC, and SR registers?
2. Define each bit in the SR register.
3. Perform the following operations. Express the final result in decimal, binary, and hexadecimal.
   (a) (BABA)h + (DADA)h
   (b) (BABA)h - (DADA)h
   (c) (BA)h x (DA)h
   (d) Describe how to observe registers during program execution with Texas Instruments Code Composer Studio.

Advanced

1. Compare and contrast the RISC vs. the CISC approach to computer architecture design.
2. Provide a one-line definition for each system of the MSP430 microcontroller.
3. The registers onboard the MSP430 microcontroller are 16 bits wide. What is the largest unsigned integer that can be stored in 16 bits? Signed integer?
4. What is the fundamental premise of the low-power operating modes of the MSP430 microcontroller?
5. Construct a summary chart of the MSP430 low-power operating modes. Include the status register bit settings to enter a specific mode and the clock status for each mode.
6. Provide an example of each addressing mode.
7. For each of the operations listed below, predict the final result and the value of flags V, N, Z, and C.
   (a) Place (CAFE)h into R12. Rotate R12 three bits to the right.
   (b) Place (CAFE)h into R12. Rotate R12 three bits to the left.
   (c) Perform (CAFE)h + (DABA)h. Add the carry bit to the result. State any assumptions made.
   (d) Perform (CAFE)h - (DABA)h. Subtract the carry bit from the result. State any assumptions made.
   (e) Perform (CA)h x (FE)h.
   (f) Place (CAFE)h into R12. Clear even bits.
   (g) Place (CAFE)h into R12. Clear odd bits.
   (h) Place (CAFE)h into R12. Set even bits.
   (i) Place (CAFE)h into R12. Set odd bits.
   (j) Place (CAFE)h into R12. Increment R12.
   (k) Place (CAFE)h into R12. Swap the high- and low-order bytes.

Challenging

1. Write an assembly language program which increments a variable. If the variable is even, LED1 on the MSP-EXP430FR2433 evaluation board will illuminate, whereas LED2 will illuminate for an even value. Note: You will need to single step through the program to observe program operation.

2. Write an assembly language program which loads a variable and then performs a sequential logical shift right. If the bit shifted out is even, LED1 on the MSP-EXP430FR2433 experimenter board will illuminate, whereas LED2 will illuminate for an even value. Note: You will need to single step through the program to observe program operation.

CHAPTER 4

MSP430 Operating Parameters and Interfacing

Objectives: After reading this chapter, the reader should be able to:

- describe the voltage and current parameters for the Texas Instruments MSP430 microcontroller;
- apply the knowledge of voltage and current parameters toward properly interfacing input and output devices to the MSP430 microcontroller;
- distinguish between low voltage 3.3 VDC and 5.0 VDC microcontroller operations;
- interface the MSP430 microcontroller operating at 3.3 VDC with a peripheral device operating at 5.0 VDC;
- interface a wide variety of input and output devices to the MSP430 microcontroller;
- describe the special concerns that must be followed when the MSP430 microcontroller is used to interface to a high-power DC or AC device;
- describe how to control the speed and direction of a DC motor; and
- describe how to control several types of AC loads.

In this chapter, we introduce the important concepts of the operating envelope for a microcontroller. We begin by reviewing the voltage and current electrical parameters for the MSP430 microcontroller. We use this information to properly interface input and output devices to the MSP430 microcontroller. The MSP430 operates at a low voltage (3.3 VDC and below). There are many compatible low voltage peripheral devices. However, many peripheral devices still operate at 5.0 VDC. We discuss how to interface a 3.3 VDC microcontroller to 5.0 VDC peripherals. We then discuss the special considerations for controlling a high-power DC or AC load such as a motor. Throughout the chapter, we provide several detailed examples to illustrate concepts.

4.1 OPERATING PARAMETERS

A microcontroller is an electronic device which has precisely defined operating conditions. If the microcontroller is used within its defined operating parameter limits, it should continue to operate correctly. However, if the allowable conditions are violated, spurious results or microcontroller damage may result.

4.1.1 MSP430 3.3 VDC OPERATION

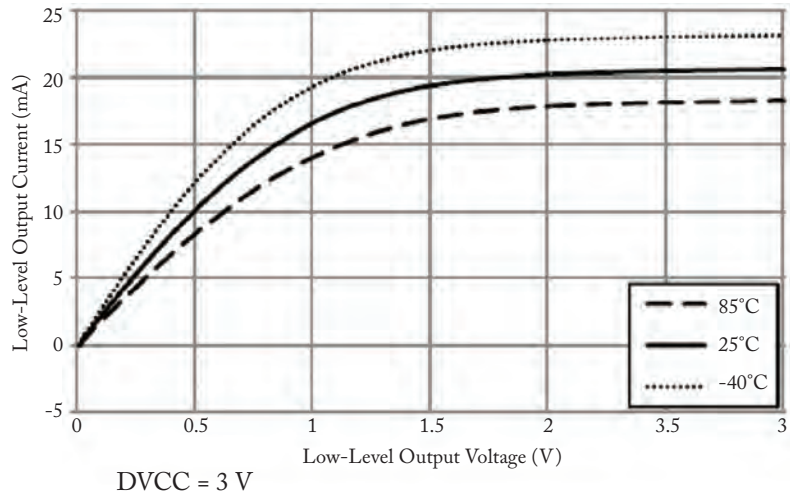
Any time a device is connected to a microcontroller, careful interface analysis must be performed. The MSP430 is a low operating voltage microcontroller. It may be operated with a supply voltage between 2.2 and 3.6 VDC. To perform the interface analysis, there are eight different electrical specifications we must consider. The electrical parameters are:

- V_{OH}: the lowest guaranteed output voltage for a logic high;
- V_{OL}: the highest guaranteed output voltage for a logic low;
- I_{OH}: the output current for a V_{OH} logic high;
- I_{OL}: the output current for a V_{OL} logic low;
- V_{IH}: the lowest input voltage guaranteed to be recognized as a logic high;
- V_{IL}: the highest input voltage guaranteed to be recognized as a logic low;
- I_{IH}: the input current for a V_{IH} logic high; and
- I_{IL}: the input current for a V_{IL} logic low.

These drive parameters cannot be used at their face value. To properly interface a peripheral device to the microcontroller, the parameters provided in Figure 4.1 must be used. As shown in the figure, operating parameter curves for the MSP430 microcontroller operating at 3.0 VDC change under varying conditions.

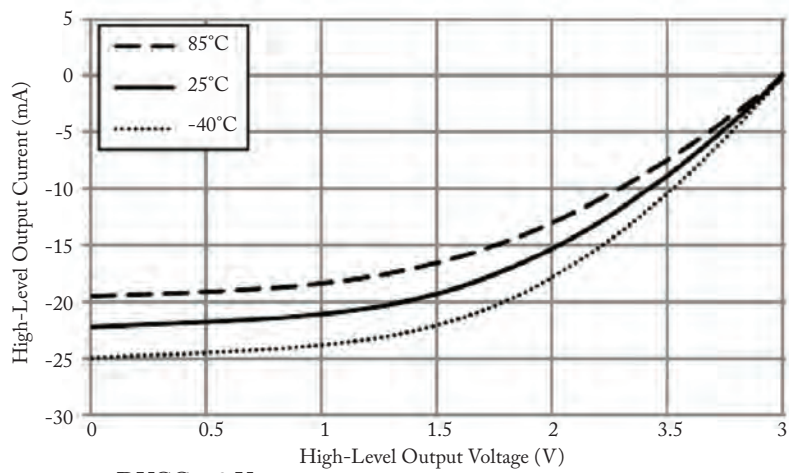
It is important to realize that these are static values taken under specific operating conditions. If external circuitry is connected such that the microcontroller acts as a current source (current leaving microcontroller) or current sink (current entering microcontroller), the voltage parameters listed above will also be affected.

In the current source case, an output voltage V_{OH} is provided at the output pin of the microcontroller when the load connected to this pin draws a current of I_{OH}. If a load draws more current from the output pin than the I_{OH} specification, the value of V_{OH} is reduced. If the load current becomes too high, the value of V_{OH} falls below the value of V_{IH} for the subsequent logic circuit stage, and it will not be recognized as an acceptable logic high signal. When this situation occurs, erratic and unpredictable circuit behavior results.



DVCC = 3 V

Typical Low-Level Output Current vs. Low-Level Output Voltage



DVCC = 3 V

Typical High-Level Output Current vs. High-Level Output Voltage

Figure 4.1: MSP430 drive current parameters [SLASE59C, 2018]. Illustration used with permission of Texas Instruments (www.ti.com).

In the sink case, an output voltage V_{OL} is provided at the output pin of the microcontroller when the load connected to this pin delivers a current of I_{OL} to this logic pin. If a load delivers more current to the output pin of the microcontroller than the I_{OL} specification, the value of V_{OL} increases. If the load current becomes too high, the value of V_{OL} rises above the value of V_{IL} for the subsequent logic circuit stage, and it will not be recognized as an acceptable logic low signal. When this situation occurs, erratic and unpredictable circuit behavior results.

You must also ensure that total current limits for an entire microcontroller port and overall bulk port specifications are observed. For planning purposes, the sum of current sourced or sinked from a port should not exceed 48 mA. As before, if these guidelines are not complied with, erratic microcontroller behavior may result. Several examples are provided in an upcoming section.

4.1.2 COMPATIBLE 3.3 VDC LOGIC FAMILIES

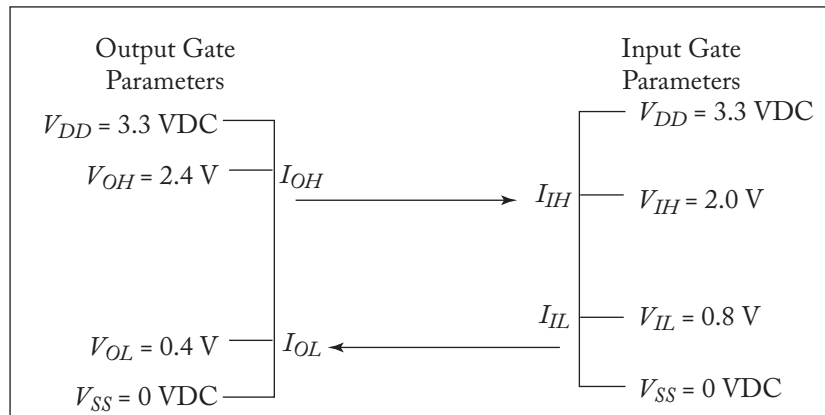
For the rest of this chapter, we limit our discussion to the MSP430 operating with a 3.3 VDC supply voltage. There are several compatible logic families that operate at 3.3 VDC. These families include the LVC, LVA, and the LVT logic families. Key parameters for the low voltage compatible families are provided in Figure 4.2 [SDYU001AB, 2017].

4.1.3 MICROCONTROLLER OPERATION AT 5.0 VDC

The MSP430 operates at 3.3 VDC and below. However, many HC CMOS microcontroller families and peripherals operate at a supply voltage of 5.0 VDC. For completeness, we provide operating parameters for these 5.0 VDC devices. This information is essential should the MSP430 be interfaced to a 5 VDC CMOS device or peripheral.

Typical values for a microcontroller in the HC CMOS family, assuming $V_{DD} = 5.0$ volts and $V_{SS} = 0$ volts, are provided below. The minus sign on several of the currents indicates a current flowing out of the device. A positive current indicates current flowing into the device.

- V_{OH} = 4.2 volts,
- V_{OL} = 0.4 volts,
- I_{OH} = −0.8 milliamps,
- I_{OL} = 1.6 milliamps,
- V_{IH} = 3.5 volts,
- V_{IL} = 1.0 volt,
- I_{IH} = 10 microamps, and
- I_{IL} = −10 microamps.



(a) Voltage and current electrical parameters

	LVC	LVA	LVT
Vcc	1.65–3.6 V	2.0–5.5 V	2.7–3.6 V
tpd	5.5 ns	14 ns	3.5 ns
loc	10 uA	20 uA	190 uA

(b) LV parameters

Figure 4.2: Low voltage compatible logic families [SDYU001AB, 2017].

4.1.4 INTERFACING 3.3 VDC LOGIC DEVICES WITH 5.0 VDC LOGIC FAMILIES

Although there are a wide variety of available 3.3 VDC peripheral devices available for the MSP430, you may find a need to interface the controller with 5.0 VDC devices. If bidirectional information exchange is required between the microcontroller and a peripheral device, a bidirectional level shifter should be used. The level shifter translates the 3.3 VDC signal up to 5 VDC for the peripheral device and back down to 3.3 VDC for the microcontroller. There are a wide variety of unidirectional and bidirectional level shifting devices available. Texas Instruments level shifting options include: unidirectional, bidirectional, and direction-controlled level shifters. For example, the LSF0101, LSF0102, LSF0204, and LSF0108 level shifters are available in the LSF010XEVM-001 Bi-Directional Multi-Voltage Level Translator Evaluation Module (LSFEVM) (www.ti.com), see Figure 4.3. Later in the chapter we show how the LSF010XEVM module is used to interface the MSP430 with a LED special effects cube.

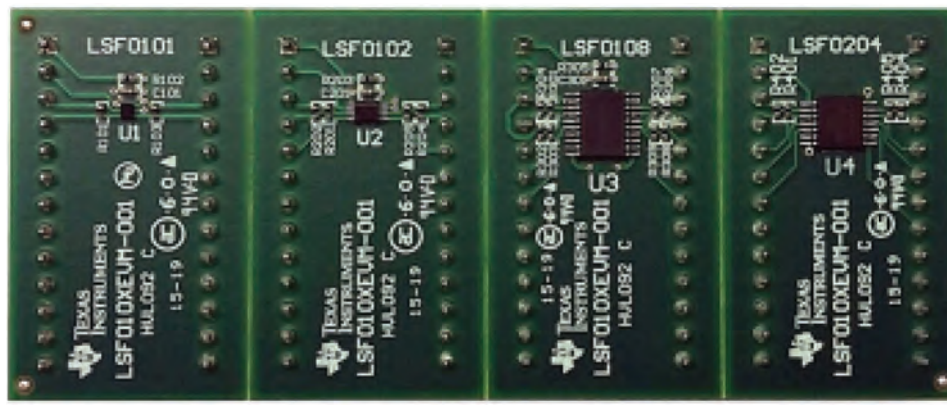


Figure 4.3: LSF010XEVM-001 bi-directional multi-voltage level translator evaluation module (LSFEVM). Illustration used with permission of Texas Instruments SDLU003A [2015].

Example: Large LED displays. Large seven-segments displays with character heights of 6.5 inches are available from SparkFun Electronics (www.sparkfun.com). Multiple display characters may be daisy chained together to form a display panel of desired character length. Only four lines from the MSP430 are required to control the display panel (ground, latch, clock, and serial data). Each character is controlled by a Large Digit Driver Board (#WIG-13279) equipped with the Texas Instruments TPIC6C596 IC Program Logic 8-bit Shifter Register. The shift register requires a 5 VDC supply and has a V_{IH} value of 4.25 VDC. The MSP430, when supplied at 3.3 VDC, has a maximum V_{OH} value of 3.3 VDC. Since the output signal levels from the MSP430 are not high enough to control the TPIC6C596, a level shifter (e.g.,

LSF010XEVM module) is required to up convert the MSP430 signals to be compatible to the ones for the TPIC6C596 [SLIS093D, 2015].

4.2 INPUT DEVICES

In this section, we discuss how to properly interface input devices to a microcontroller. We start with the most basic input component, a simple on/off switch.

4.2.1 SWITCHES

Switches come in a variety of types. As a system designer, it is up to you to choose the appropriate switch for a specific application. Switch varieties commonly used in microcontroller applications are illustrated in Figure 4.4a. Provided below is a brief summary of the different types.

- **Slide switch:** A slide switch has two different positions: on and off. The switch is manually moved to one position or the other. For microcontroller applications, slide switches are available that fit in the profile of a common integrated circuit size dual inline package (DIP). A bank of four or eight DIP switches in a single package is commonly available.
- **Momentary contact pushbutton switch:** A momentary contact pushbutton switch comes in two varieties: normally closed (NC) and normally open (NO). A normally open switch, as its name implies, does not normally provide an electrical connection between its contacts. When the pushbutton portion of the switch is depressed, the connection between the two switch contacts is made. The connection is held as long as the switch is depressed. When the switch is released, the connection is opened. The converse is true for a normally closed switch. For microcontroller applications, pushbutton switches are available in a small Tactile (tact) type switch configuration. The MSP-EXP430FR2433 and the MSP-EXP430FR5994 LaunchPads are each equipped with two pushbutton tactile (tact) switches designated S1 and S2.
- **Push on/push off switches:** This switch type of is also available in a normally open or normally closed configuration. For the normally open configuration, the switch is depressed to make connection between the two switch contacts. The pushbutton must be depressed again to release the connection.
- **Hexadecimal rotary switches:** Small profile rotary switches are available for microcontroller applications. These switches commonly have sixteen rotary switch positions. As the switch is rotated to each position, a unique four-bit binary code is provided at the switch contacts.

A common switch interface is shown in Figure 4.4b. This interface allows a logic one or zero to be properly introduced to a microcontroller input port pin. The basic interface consists of the switch in series with a current limiting resistor. The node between the switch and the

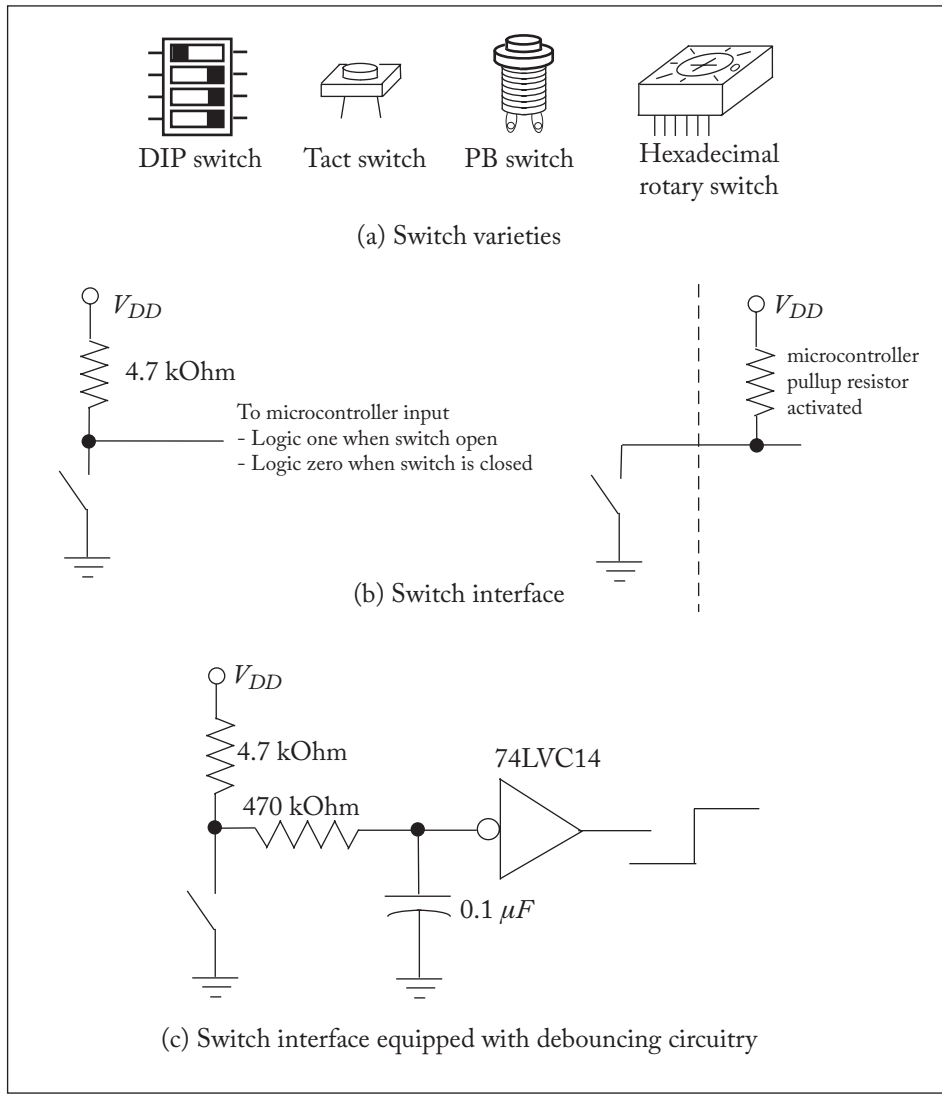


Figure 4.4: Switch interface.

resistor is provided to the microcontroller input pin. In the configuration shown, the resistor pulls the microcontroller input up to the supply voltage V_{DD}. When the switch is closed, the node is grounded and a logic zero is detected by the microcontroller input pin. To reverse the logic of the switch configuration, the position of the resistor and the switch is simply reversed.

4.2.2 SWITCH DEBOUNCING

Mechanical switches do not make a clean transition from one position (on) to another (off). When a switch is moved from one position to another, it makes, and breaks, contact multiple times. This activity may go on for tens of milliseconds. A microcontroller is relatively fast as compared to the action of the switch. Therefore, the microcontroller is able to recognize each switch bounce as a separate and erroneous transition.

To correct the switch bounce phenomena, additional external hardware components may be used or software techniques may be employed. A hardware debounce circuit is illustrated in Figure 4.4c. The node between the switch and the limiting resistor of the basic switch circuit is fed to a low-pass filter (LPF), formed by the 470 kOhm resistor and the capacitor. The LPF prevents abrupt changes (bounces) in the input signal from the microcontroller. The LPF is followed by a 74LVC14 Schmitt trigger which is simply an inverter equipped with hysteresis. The Schmitt trigger has a different value of threshold voltage to trigger an output change for a positive transitioning (logic low to high) signal vs. a negative transitioning (logic high to low) signal. This difference in threshold voltages results in typical hysteresis values of 0.3 VDC. The hysteresis further limits the switch bouncing.

Switches may also be debounced using software techniques. This is accomplished by inserting a 30–50 ms lockout delay in the function responding to port pin changes. The delay prevents the microcontroller from responding to the multiple switch transitions related to bouncing.

You must carefully analyze a given design to determine if hardware or software switch debouncing techniques should be used. It is important to remember that all switches exhibit bounce phenomena and therefore must be debounced.

4.2.3 KEYPADS

A keypad is an extension of the simple switch configuration. A typical keypad configuration and interface are shown in Figure 4.5. As you can see, the keypad contains multiple switches in a two-dimensional array configuration. The switches in the array share common row and column connections. The common column connections are pulled up to Vcc by external 10 K resistors or by pull-up resistors within the MSP430.

To determine if a switch has been depressed, a single row of keypad switches is first asserted by the microcontroller, followed by a reading of the host keypad column inputs. If a switch has been depressed, the keypad pin corresponding to the column the switch is in will also be asserted. The combination of a row and a column assertion can be decoded to determine which key has been pressed. The keypad rows are sequentially asserted. Since the keypad is a collection

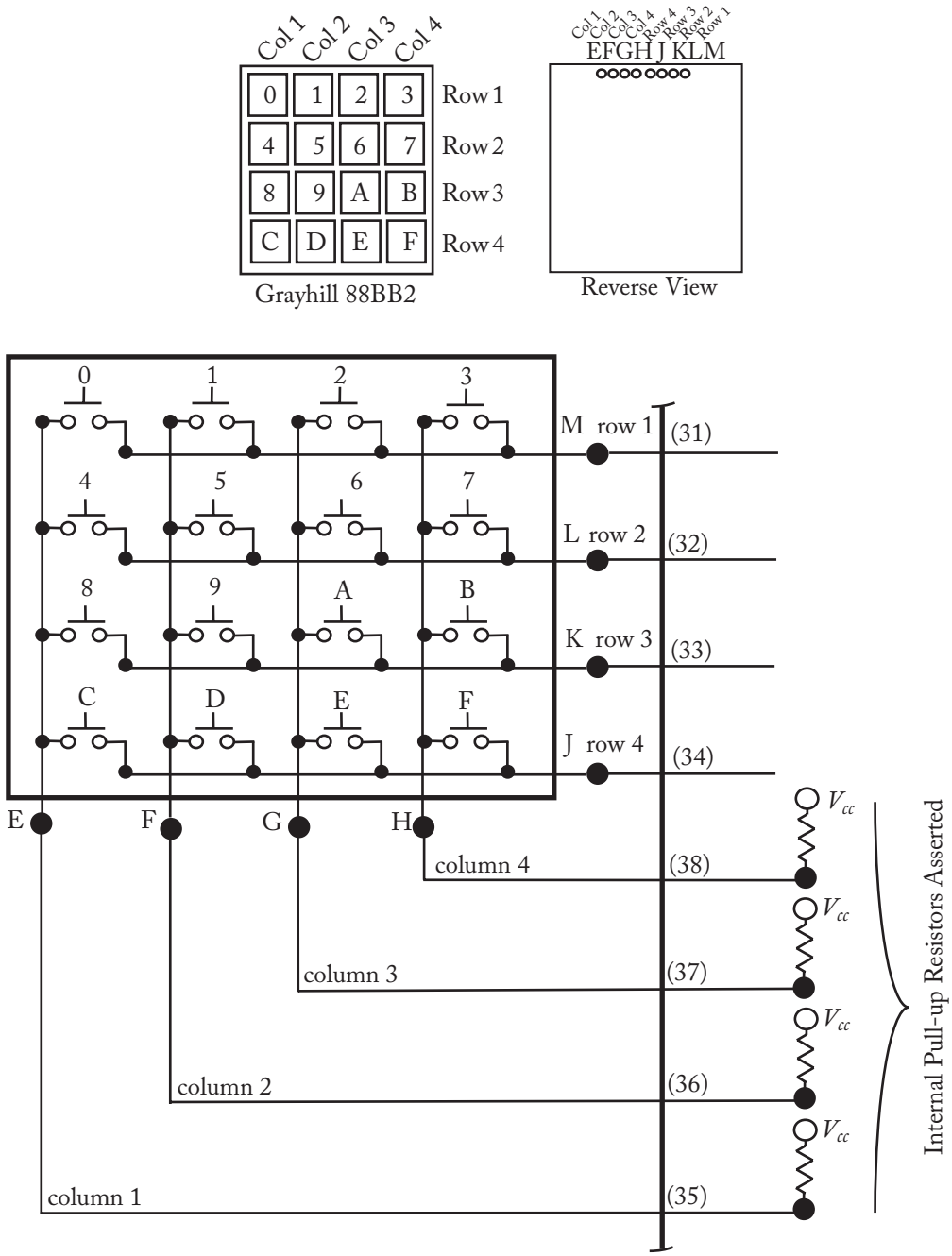


Figure 4.5: Keypad interface.

of switches, debounce techniques must also be employed. In the example code provided, a 200 ms delay is provided to mitigate switch bounce. In the keypad shown, the rows are sequentially asserted active low (0).

The keypad is typically used to capture user requests to a microcontroller. A standard keypad with alphanumeric characters may be used to provide alphanumeric values to the microcontroller such as providing your personal identification number (PIN) for a financial transaction. However, some keypads are equipped with removable switch covers such that any activity can be associated with a key press.

Example: Keypad. In this example a Grayhill 88BB2 4-by-4 matrix keypad is interfaced to the MSP-EXP430FR5994 LaunchPad. The example shows how a specific switch depression can be associated with different activities by using a "switch" statement.

```
//*********************************************************************
//keypad_4X4
//Specified pins are for the MSP-EXP430FR5994 LuanchPad
//This code is in the public domain.
//*********************************************************************

#define row1  31
#define row2  32
#define row3  33
#define row4  34

#define col1  35
#define col2  36
#define col3  37
#define col4  38

unsigned char  key_depressed = '*';

void setup()
  {
  //start serial connection to monitor
  Serial.begin(9600);

  //configure row pins as ouput
  pinMode(row1, OUTPUT);
  pinMode(row2, OUTPUT);
  pinMode(row3, OUTPUT);
  pinMode(row4, OUTPUT);
```

```
  //configure column pins as input and assert pullup resistors
  pinMode(col1, INPUT_PULLUP);
  pinMode(col2, INPUT_PULLUP);
  pinMode(col3, INPUT_PULLUP);
  pinMode(col4, INPUT_PULLUP);
  }


void loop()
   {
   //Assert row1, deassert row 2,3,4
   digitalWrite(row1, LOW);   digitalWrite(row2, HIGH);
   digitalWrite(row3, HIGH);  digitalWrite(row4, HIGH);

   //Read columns
   if (digitalRead(col1) == LOW)
     key_depressed = '0';
   else if (digitalRead(col2) == LOW)
     key_depressed = '1';
   else if (digitalRead(col3) == LOW)
     key_depressed = '2';
   else if (digitalRead(col4) == LOW)
     key_depressed = '3';
   else
     key_depressed = '*';


   if (key_depressed == '*')
     {
     //Assert row2, deassert row 1,3,4
     digitalWrite(row1, HIGH);   digitalWrite(row2, LOW);
     digitalWrite(row3, HIGH);  digitalWrite(row4, HIGH);

     //Read columns
     if (digitalRead(col1) == LOW)
       key_depressed = '4';
     else if (digitalRead(col2) == LOW)
       key_depressed = '5';
```

```
  else if (digitalRead(col3) == LOW)
    key_depressed = '6';
  else if (digitalRead(col4) == LOW)
    key_depressed = '7';
  else
    key_depressed = '*';
  }

 if (key_depressed == '*')
  {
  //Assert row3, deassert row 1,2,4
  digitalWrite(row1, HIGH);   digitalWrite(row2, HIGH);
  digitalWrite(row3, LOW);  digitalWrite(row4, HIGH);

  //Read columns
  if (digitalRead(col1) == LOW)
    key_depressed = '8';
  else if (digitalRead(col2) == LOW)
    key_depressed = '9';
  else if (digitalRead(col3) == LOW)
    key_depressed = 'A';
  else if (digitalRead(col4) == LOW)
    key_depressed = 'B';
  else
    key_depressed = '*';
  }

if (key_depressed == '*')
  {
  //Assert row4, deassert row 1,2,3
  digitalWrite(row1, HIGH);   digitalWrite(row2, HIGH);
  digitalWrite(row3, HIGH);  digitalWrite(row4, LOW);

  //Read columns
  if (digitalRead(col1) == LOW)
    key_depressed = 'C';
  else if (digitalRead(col2) == LOW)
    key_depressed = 'D';
  else if (digitalRead(col3) == LOW)
```

```
      key_depressed = 'E';
    else if (digitalRead(col4) == LOW)
      key_depressed = 'F';
    else
      key_depressed = '*';
    }

  if(key_depressed != '*')
     {
     Serial.write(key_depressed);
     Serial.write(' ');

     switch(key_depressed)
       {
       case '0' : Serial.println("Do something associated with case 0");
                  break;

       case '1' : Serial.println("Do something associated with case 1");
                  break;

       case '2' : Serial.println("Do something associated with case 2");
                  break;

       case '3' : Serial.println("Do something associated with case 3");
                  break;

       case '4' : Serial.println("Do something associated with case 4");
                  break;

       case '5' : Serial.println("Do something associated with case 5");
                  break;

       case '6' : Serial.println("Do something associated with case 6");
                  break;

       case '7' : Serial.println("Do something associated with case 7");
                  break;

       case '8' : Serial.println("Do something associated with case 8");
```

```
                break;

        case '9' : Serial.println("Do something associated with case 9");
                break;

        case 'A' : Serial.println("Do something associated with case A");
                break;

        case 'B' : Serial.println("Do something associated with case B");
                break;

        case 'C' : Serial.println("Do something associated with case C");
                break;

        case 'D' : Serial.println("Do something associated with case D");
                break;

        case 'E' : Serial.println("Do something associated with case E");
                break;

        case 'F' : Serial.println("Do something associated with case F");
                break;
        }
      }
   //limit switch bounce
   delay(200);
   }
//*********************************************************************
```

4.2.4 SENSORS

A microcontroller is typically used in control applications where data is collected. The data is assimilated and processed by the host algorithm, and a control decision and accompanying signals are provided by the microcontroller. Input data for the microcontroller is collected by a complement of input sensors. These sensors may be digital or analog in nature.

Digital Sensors

Digital sensors provide a series of digital logic pulses with sensor data encoded. The sensor data may be encoded in any of the parameters associated with the digital pulse train such as duty cycle,

frequency, period, or pulse rate. The input portion of the timing system may be configured to measure these parameters.

An example of a digital sensor is the optical encoder. An optical encoder consists of a small plastic transparent disk with opaque lines etched into the disk surface. A stationary optical emitter and detector pair are placed on either side of the disk. As the disk rotates, the opaque lines break the continuity between the optical source and detector. The signal from the optical detector is monitored to determine disk rotation, as shown in Figure 4.6.

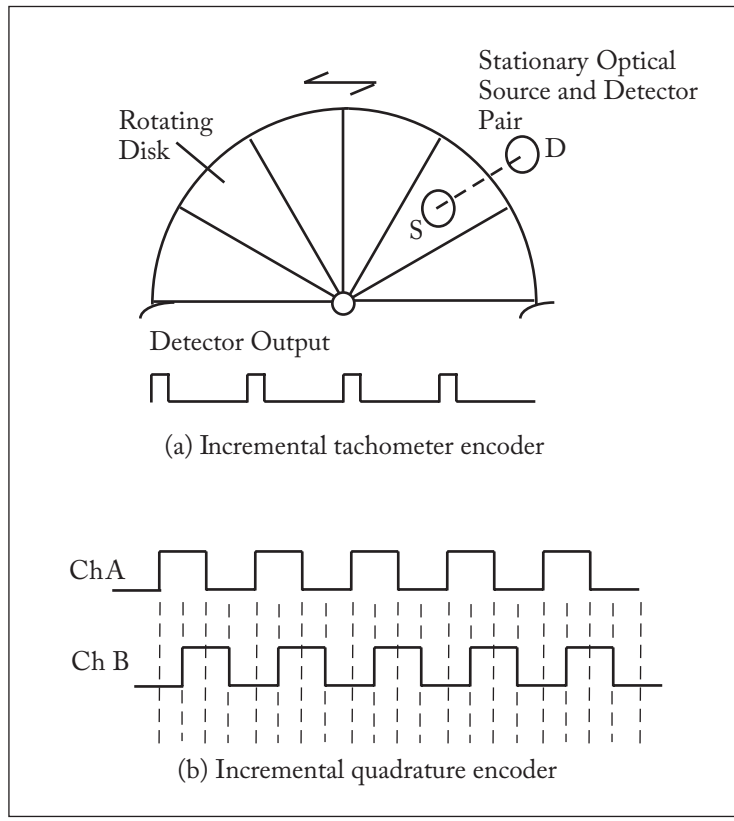


Figure 4.6: Optical encoder.

There are two major types of optical encoders: incremental encoders and absolute encoders. An absolute encoder is used when it is required to retain position information when power is lost. For example, if you were using an optical encoder in a security gate control system, an absolute encoder would be used to monitor the gate position. The absolute encoder is equipped with multiple data tracks to determine the precise location of the encoder disk [Stegmann].

An incremental encoder is used in applications where velocity and/or direction information is required. The incremental encoder types may be further subdivided into tachometers and quadrature encoders. An incremental tachometer encoder consists of a single track of etched opaque lines as shown in Figure 4.6a. It is used when the velocity of a rotating device is required. To calculate velocity, the number of detector pulses is counted in a fixed amount of time. Since the number of pulses per encoder revolution is known, velocity may be calculated. The quadrature encoder contains two tracks shifted in relationship to one another by 90°. This allows the calculation of both velocity and direction. To determine direction, one would monitor the phase relationship between Channel A and Channel B, as shown in Figure 4.6b [Stegmann].

Example: Optical encoder. An optical encoder provides 200 pulses per revolution. The encoder is connected to a rotating motor shaft. If 80 pulses are counted in a 100 ms span, what is the speed of the motor in revolutions per minute (RPM)?

$$(1\ rev/200\ pulses)\ \times\ (80\ pulses/0.100\ s)\ \times\ (60\ s/min)\ =\ 240\ RPM.$$

Analog Sensors and Transducers

Analog sensors or transducers provide a DC voltage that is proportional to the physical parameter being measured. The analog signal may be first preprocessed by external analog hardware such that it falls within the voltage references of the conversion subsystem. In the case of the MSP430 microcontroller, the transducer output must fall between 0 and 3.3 VDC. The analog voltage is then converted to a corresponding binary representation.

An example of an analog sensor is the flex sensor shown in Figure 4.7a. The flex sensor provides a change in resistance for a change in sensor flexure. At 0° flex, the sensor provides 10 kOhms of resistance. For 90° flex, the sensor provides 30–40 kOhms of resistance. Since the microcontroller cannot measure resistance directly, the change in flex sensor resistance is converted to a change in a DC voltage. This is accomplished using the voltage divider network shown in Figure 4.7c. For increased flex, the DC voltage will increase. The voltage can be measured using the MSP430's analog to digital converter subsystem.

The flex sensor may be used in applications such as virtual reality data gloves, robotic sensors, biometric sensors, and in science and engineering experiments [Images Company]. One of the co-authors used the circuit provided in Figure 4.7 to help a colleague in zoology monitor the movement of a newt salamander during a scientific experiment.

Example: Joystick. The thumb joystick is used to select a desired direction in an X–Y plane as shown in Figure 4.9. The thumb joystick contains two built-in potentiometers (horizontal and vertical). A reference voltage of 3.3 VDC is applied to the VCC input of the joystick. As the joystick is moved, the horizontal (HORZ) and vertical (VERT) analog output voltages will change to indicate the joystick position. The joystick is also equipped with a digital select (SEL) button.

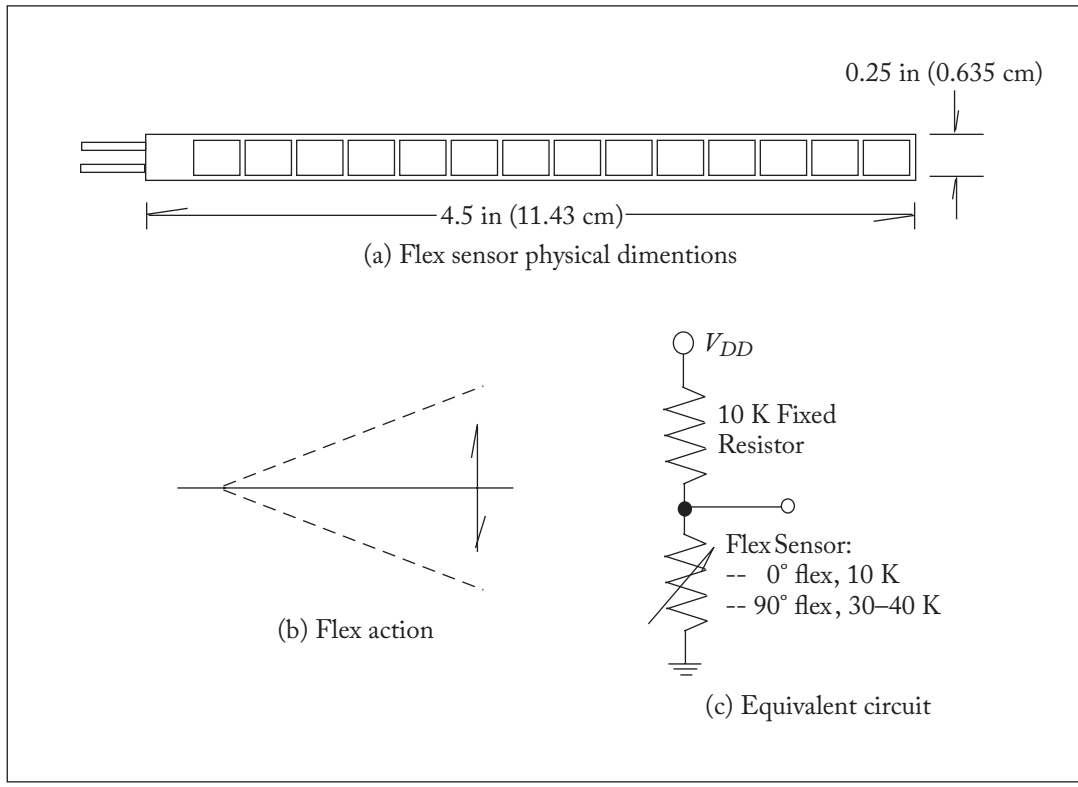


Figure 4.7: Flex sensor.

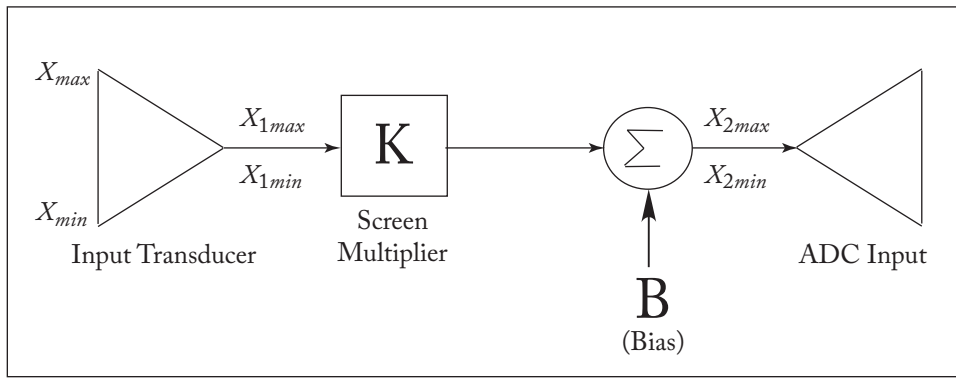
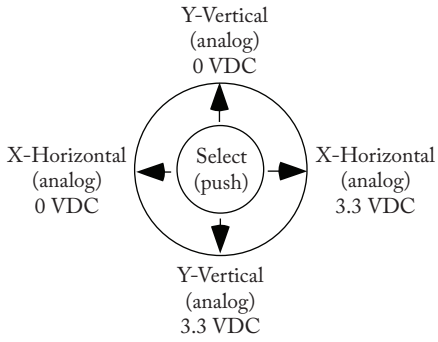
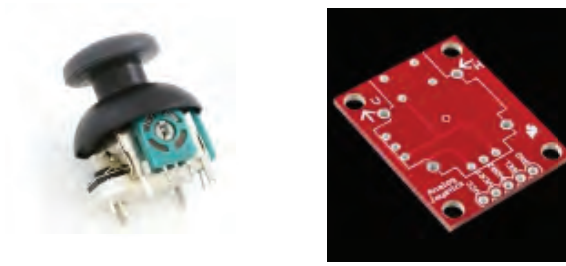


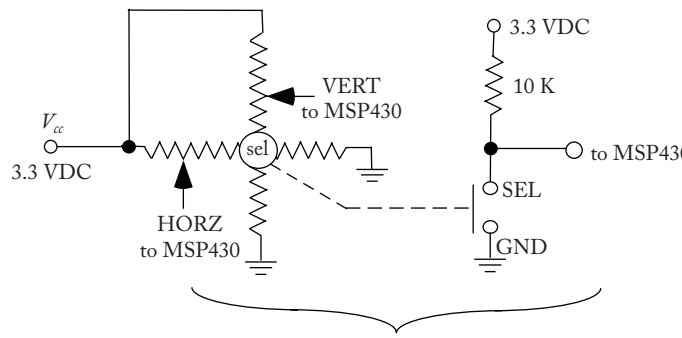
Figure 4.8: A block diagram of the signal conditioning for an analog-to-digital converter. The range of the sensor voltage output is mapped to the analog-to-digital converter input voltage range. The scalar multiplier maps the magnitudes of the two ranges and the bias voltage is used to align two limits.



(a) Joystick operation



(b) Sparkfun joystick (COM-09032) and breakout board (BOB-09110)



(c) Thumb joystick circuit

Figure 4.9: Thumb joystick. Images used with permission of Sparkfun (www.sparkfun.com).

Example: IR sensor. In Chapter 2, a Sharp IR sensor is used to sense the presence of maze walls. In this example, we use the Sharp GP2Y0A21YKOF IR sensor to control the intensity of an LED. The profile of the Sharp IR sensor is provided in Figure 4.10.

```
//*****************************************************************
//IR_sensor
//
//The circuit:
//- For the MSP-EXP430FR2433 LaunchPad, the IR sensor signal pin is
//  connected to analog pin 0 (2).
//- The sensor power and ground pins are connected to 5 VDC and
//  ground respectively.
//- The analog output is designated as the onboard red LED.
//
```

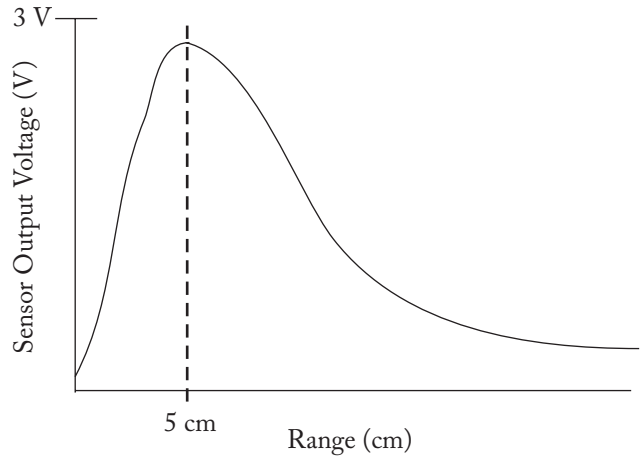


Figure 4.10: Sharp GP2Y0A21YKOF IR sensor profile.

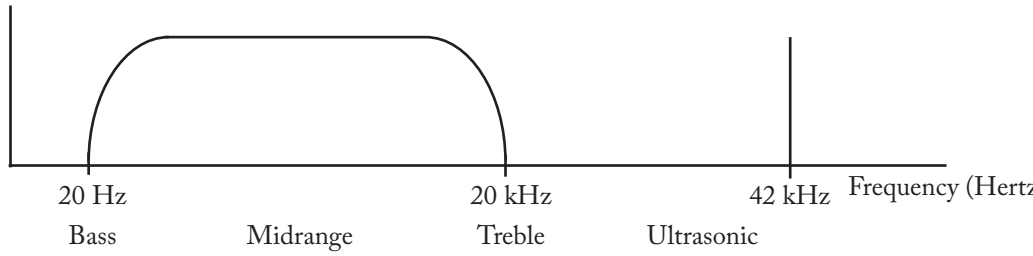
```
//Created:  Dec 29, 2008
//Modified: Aug 30, 2011
//Author:   Tom Igoe
//
//This example code is in the public domain.
//****************************************************************

const int analogInPin  = 2;       //Energia analog input pin A0
const int analogOutPin = RED_LED; //Energia onboard red LED pin

int sensorValue = 0;          //value read from the OR sensor
int outputValue = 0;          //value output to the PWM (red LED)

void setup()
{
// initialize serial communications at 9600 bps:
Serial.begin(9600);
}

void loop()
{
//read the analog in value:
sensorValue = analogRead(analogInPin);
```

```
// map it to the range of the analog out:
outputValue = map(sensorValue, 0, 1023, 0, 255);

// change the analog out value:
analogWrite(analogOutPin, outputValue);

// print the results to the serial monitor:
Serial.print("sensor = " );
Serial.print(sensorValue);
Serial.print("\t output = ");
Serial.println(outputValue);

// wait 10 milliseconds before the next loop
// for the analog-to-digital converter to settle
// after the last reading:
delay(10);
}

//*************************************************************
```

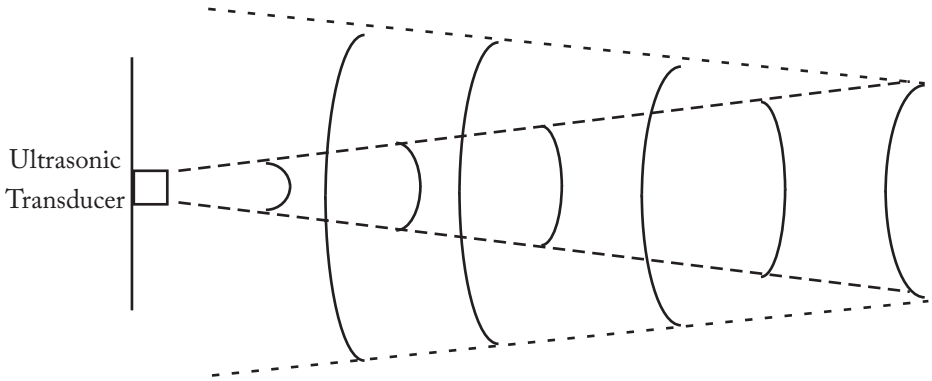
Example: Ultrasonic sensor. The ultrasonic sensor pictured in Figure 4.11 is an example of an analog-based sensor. The sensor is based on the concept of ultrasound or sound waves that are at a frequency above the human range of hearing (20 Hz to 20 kHz). The ultrasonic sensor pictured in Figure 4.11c emits a sound wave at 42 kHz. The sound wave reflects from a solid surface and returns to the sensor. The amount of time for the sound wave to transit from the surface and back to the sensor may be used to determine the range from the sensor to the wall. Figure 4.11c,d show an ultrasonic sensor manufactured by Maxbotix (LV-EZ3). The sensor provides an output that is linearly related to range in three different formats: (a) a serial RS-232 compatible output at 9600 bits per second, (b) a pulse output which corresponds to 147 us/in width, and (c) an analog output at a resolution of 10 mV/in. The sensor is powered from a 2.5–5.5 VDC source (`www.sparkfun.com`).

Example: Inertial measurement unit. Pictured in Figure 4.12 is an inertial measurement unit (IMU) which consists of an IDG5000 dual-axis gyroscope and an ADXL335 triple axis accelerometer. This sensor may be used in unmanned aerial vehicles (UAVs), autonomous helicopters and robots. For robotic applications the robot tilt may be measured in the X and Y directions as shown in Figure 4.12c,d (`www.sparkfun.com`).

Example: Level sensor. Milone Technologies manufacture a line of continuous fluid level sensors. The sensor resembles a ruler and provides a near liner response, as shown in Figure 4.13.



(a) Sound spectrum



(b) Ultrasonic range finding

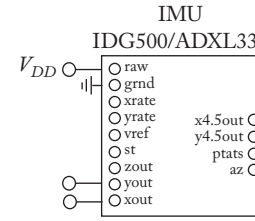
(c) Ultrasonic range finder Maxbotix LV-EZ3 (SparkFun SEN-08501)

O1: leave open
O2: PW
O3: analog output
O4: RX
O5: TX
O6: V+(3.3–5.0V)
O7: gnd

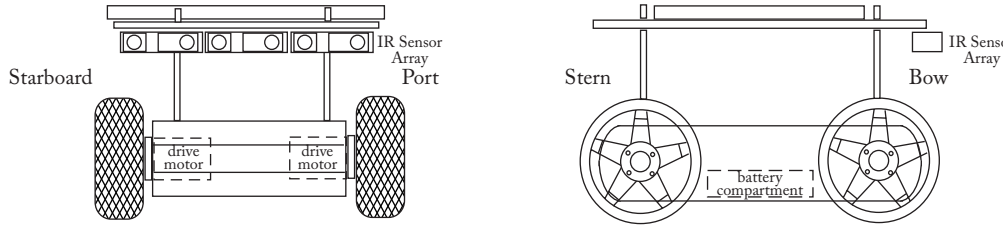
(d) Pinout

Figure 4.11: Ultrasonic sensor. (Sensor image used courtesy of SparkFun, Electronics (www.sparkfun.com).)

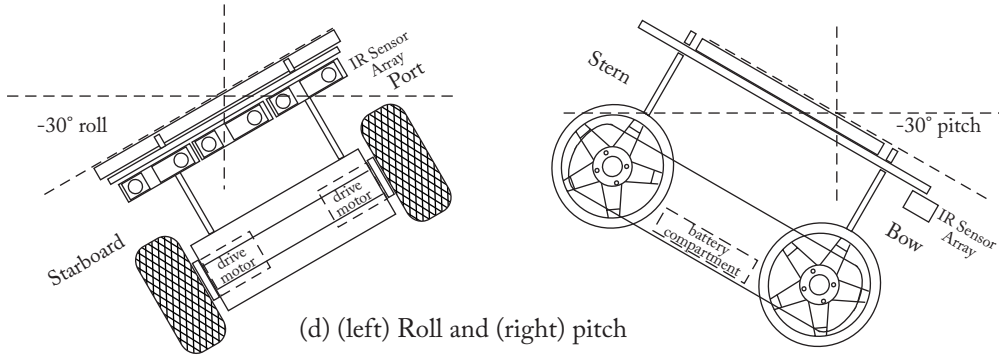
(a) SparkFun IMU Analog Combo Board
5 Degrees of Freedom IDF500/ADXL335 SEN



(b) IDG500/ADXL335 pinout



(c) (left) Robot front view and (right) side view



(d) (left) Roll and (right) pitch

Figure 4.12: Inertial measurement unit. (IMU image used courtesy of SparkFun, Electronics (www.sparkfun.com).)

The sensor reports a change in resistance to indicate the distance from sensor top to the fluid surface. A wide resistance change occurs from 700 ohms at a 1-in fluid level to 50 ohms at a 12.5-in fluid level (www.milonetech.com). To covert the resistance change to a voltage change measurable by the MSP430, a voltage divider circuit as shown in Figure 4.13 may be used. With a supply voltage (V_{DD}) of 3.3 VDC, a V_{TAPE} voltage of 0.855 VDC results for a one inch fluid level. Whereas, a fluid of 12.5 in provides a V_{TAPE} voltage level of 0.080 VDC.

4.2.5 TRANSDUCER INTERFACE DESIGN (TID) CIRCUIT

In addition to transducers, we also need a signal conditioning circuitry before we can apply the signal for analog-to-digital conversion. The signal conditioning circuitry is called the transducer interface. The objective of the transducer interface circuit is to scale and shift the electrical signal range to map the output of the input transducer to the input range of the analog-to-digital converter, which is typically 0–3.3 VDC. Figure 4.8 shows the transducer interface circuit using an input transducer.

The transducer interface consists of two steps: scaling and shifting via a DC bias. The scale step allows the span of the transducer output to match the span of the ADC system input range. The bias step shifts the output of the scale step to align with the input of the ADC system. In general, the scaling and bias process may be described by two equations:

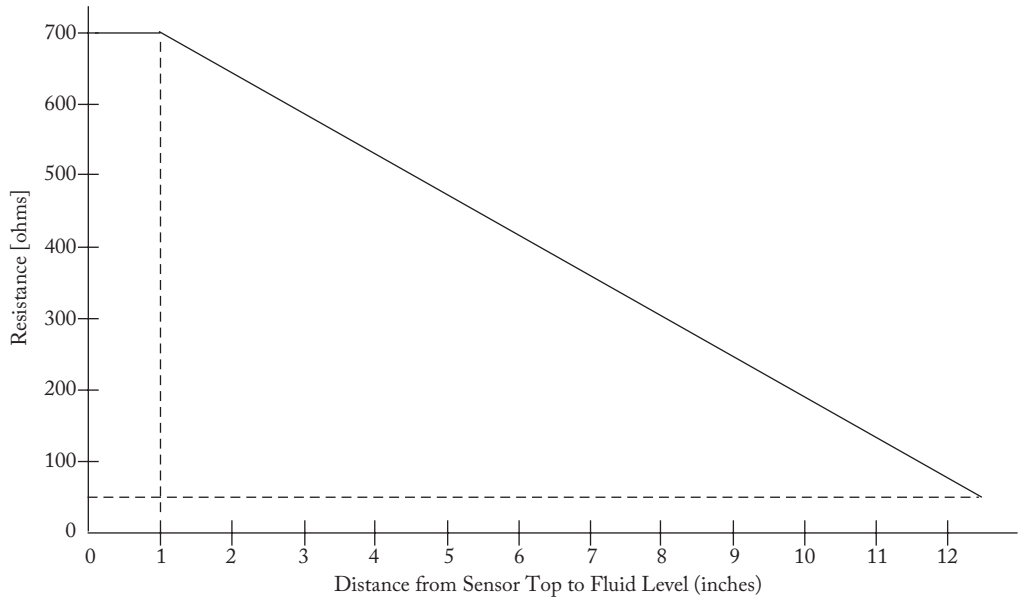
$$V_{2max} = (V_{1max} \times K) + B$$
$$V_{2min} = (V_{1min} \times K) + B.$$

The variable V_{1max} represents the maximum output voltage from the input transducer. This voltage occurs when the maximum physical variable (X_{max}) is presented to the input transducer. This voltage must be scaled by the scalar multiplier (K) and then have a DC offset bias voltage (B) added to provide the voltage V_{2max} to the input of the ADC converter [USAFA].

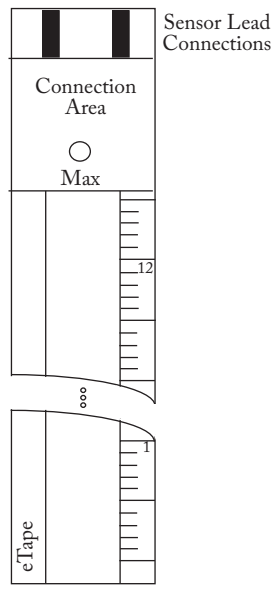
Similarly, the variable V_{1min} represents the minimum output voltage from the input transducer. This voltage occurs when the minimum physical variable (X_{min}) is presented to the input transducer. This voltage must be scaled by the scalar multiplier (K) and then have a DC offset bias voltage (B) added to produce voltage V_{2min}, the input of the ADC converter.

Usually the values of V_{1max} and V_{1min} are provided with the documentation for the transducer. Also, the values of V_{2max} and V_{2min} are known. They are the high and low reference voltages for the ADC system (usually 3.3 VDC and 0 VDC for the MSP430 microcontroller). We thus have two equations and two unknowns to solve for K and B. The circuits to scale by K and add the offset B are usually implemented with operational amplifiers.

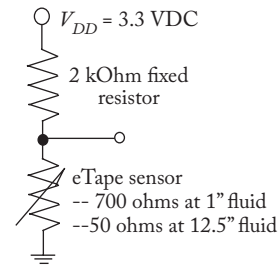
Example: A photodiode is a semiconductor device that provides an output current, corresponding to the light impinging on its active surface. The photodiode is used with a transimpedance amplifier to convert the output current to an output voltage. A photodiode/transimpedance amplifier provides an output voltage of 0 volt for maximum rated light intensity and -2.50 VDC



(a) Characteristics for Milone Technologies eTape TM fluid level sensor



(b) eTape Sensor



(c) Equivalent Circuit

Figure 4.13: Milone technologies fluid level sensor (www.milonetech.com).

output voltage for the minimum rated light intensity. Calculate the required values of K and B for this light transducer, so it may be interfaced to a microcontroller's ADC system.

$$\begin{aligned} V_{2max} &= (V_{1max} \times K) + B \\ V_{2min} &= (V_{1min} \times K) + B \\ 3.3V &= (0V \times K) + B \\ 0V &= (-2.50V \times K) + B. \end{aligned}$$

The values of K and B may then be determined to be 1.3 and 3.3 VDC, respectively.

4.2.6 OPERATIONAL AMPLIFIERS

In the previous section, we discussed the transducer interface design (TID) process. Going through this design process yields a required value of gain (K) and DC bias (B). Operational amplifiers (op amps) are typically used to implement a TID interface. In this section, we briefly introduce operational amplifiers including ideal op amp characteristics, classic op amp circuit configurations, and an example to illustrate how to implement a TID with op amps. Op amps are also used in a wide variety of other applications, including analog computing, analog filter design, and a myriad of other applications. The interested reader is referred to the References section at the end of the chapter for pointers to some excellent texts on this topic.

The Ideal Operational Amplifier

A generic ideal operational amplifier is shown in Figure 4.14. An ideal operational does not exist in the real world. However, it is a good first approximation for use in developing op amp application circuits.

The op amp is an active device (requires power supplies) equipped with two inputs, a single output, and several voltage source inputs. The two inputs are labeled Vp, or the non-inverting input, and Vn, the inverting input. The output of the op amp is determined by taking the difference between Vp and Vn and multiplying the difference by the open loop gain (Avol) of the op amp, which is typically a large value much greater than 50,000. Due to the large value of A_{vol}, it does not take much of a difference between Vp and Vn before the op amp will saturate. When an op amp saturates, it does not damage the op amp, but the output is limited to $\pm V_{cc}$. This will clip the output, and hence distort the signal, at levels slightly less than $\pm V_{cc}$. Due to this reason, op amps are typically used in a closed loop, negative feedback configuration. A sample of classic operational amplifier configurations with negative feedback are provided in Figure 4.15 [Faulkenberry, 1977].

It should be emphasized that the equations provided with each operational amplifier circuit are only valid if the circuit configurations are identical to those shown. Even a slight variation in the circuit configuration may have a dramatic effect on circuit operation. To analyze each operational amplifier circuit, use the following steps.

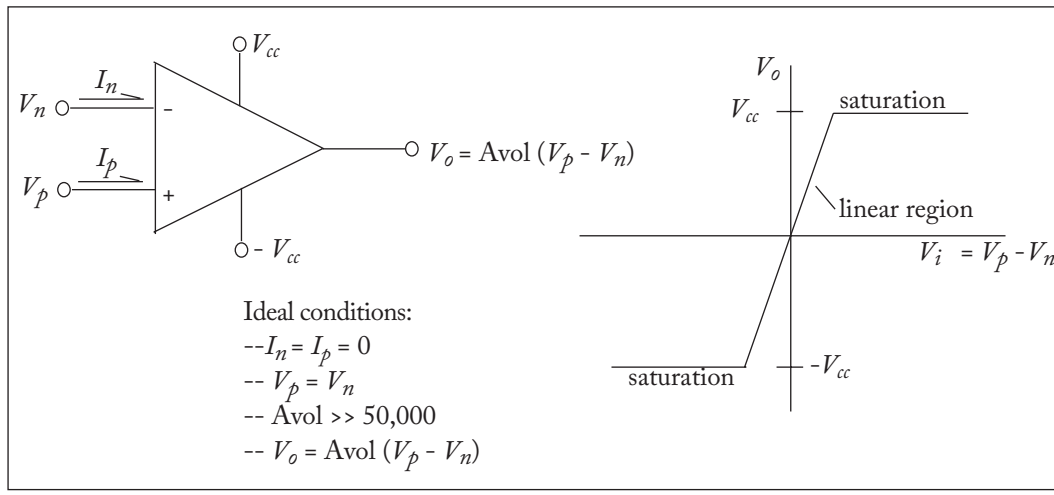


Figure 4.14: Ideal operational amplifier characteristics.

- Write the node equation at Vn for the circuit.
- Apply ideal op amp characteristics to the node equation.
- Solve the node equation for Vo.

As an example, we provide the analysis of the non-inverting amplifier circuit in Figure 4.16. This same analysis technique may be applied to the remaining circuits in Figure 4.15 to arrive at the equations for V_{out} provided.

Example: In the previous section, it was determined that the values of K and B were 1.3 and 3.3 VDC, respectively. The two-stage op amp circuitry in Figure 4.17 implements these values of K and B. The first stage provides an amplification of -1.3 due to the use of the non-inverting amplifier configuration. In the second stage, a summing amplifier is used to add the output of the first stage with a bias of 3.3 VDC. Since this stage also introduces a minus sign to the result, the overall result of a gain of 1.3 and a bias of +3.3 VDC is achieved.

Low-voltage operational amplifiers, operating in the 2.7–5 VDC range, are readily available from Texas Instruments.

4.3 OUTPUT DEVICES

An external device should not be connected to a microcontroller without first performing careful interface analysis to ensure the voltage, current, and timing requirements of the microcontroller and the external device are met. In this section, we describe interface considerations for a wide variety of external devices. We begin with the interface for a single LED.

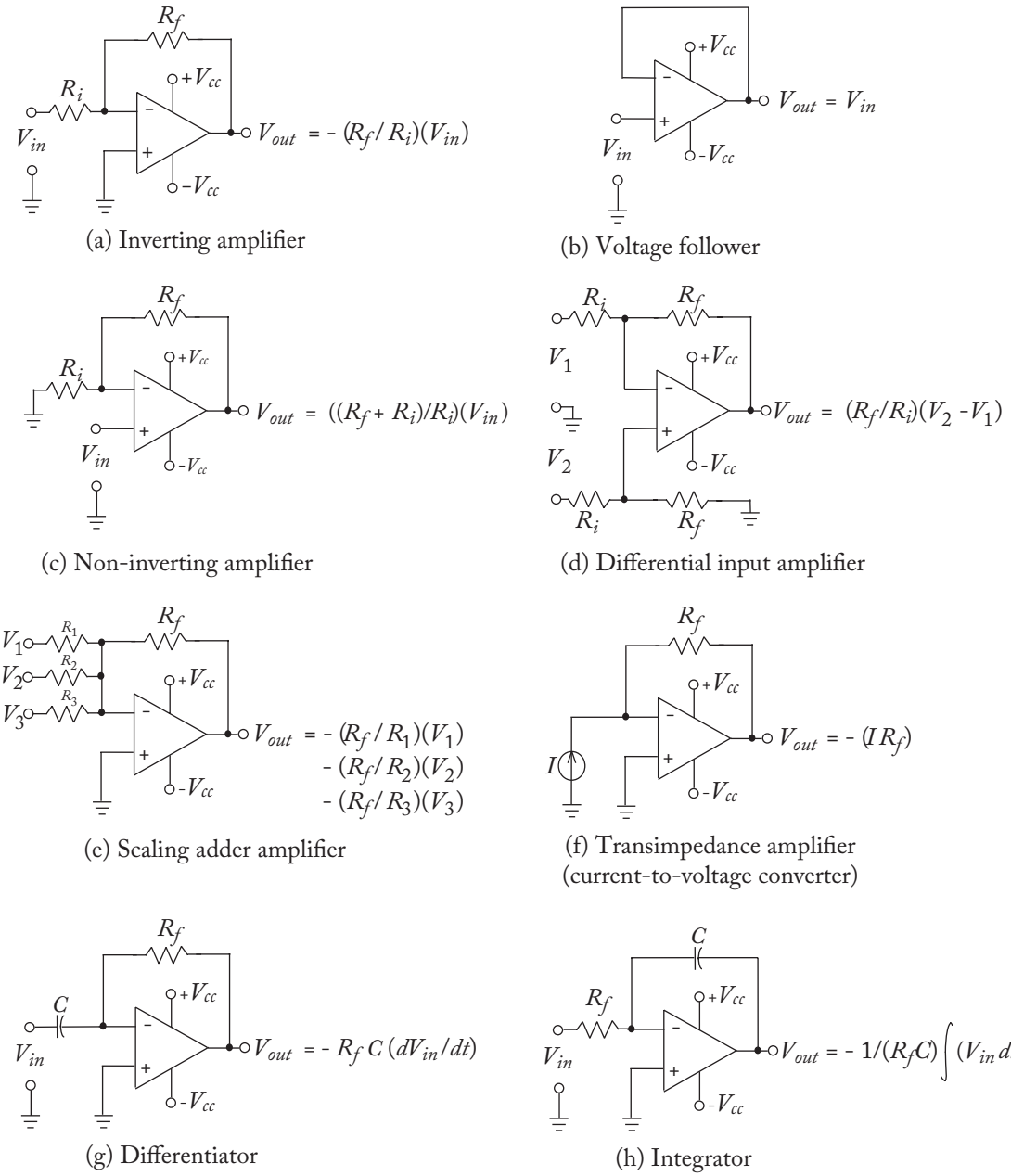
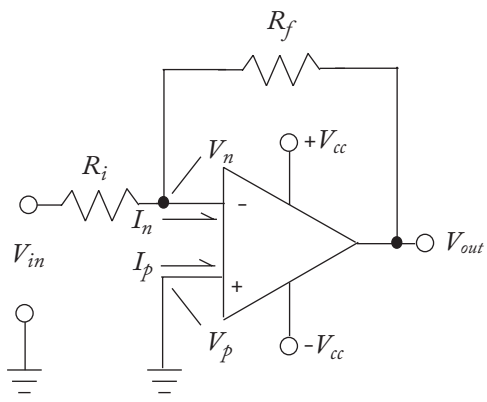


Figure 4.15: Classic operational amplifier configurations. (Adapted from Faulkenberry [1977].)



Node equation at V_n:

$(V_n - V_{in})/ R_i + (V_n - V_{out})/R_f + I_n = 0$

Apply ideal conditions

$I_n = I_p = 0$

$V_n = V_p = 0$ (since V_p is grounded)

Solve node equation for V_{out}:

$V_{out} = - (R_f / R_i)(V_{in})$

Figure 4.16: Operational amplifier analysis for the non-inverting amplifier. (Adapted from Faulkenberry [1982].)

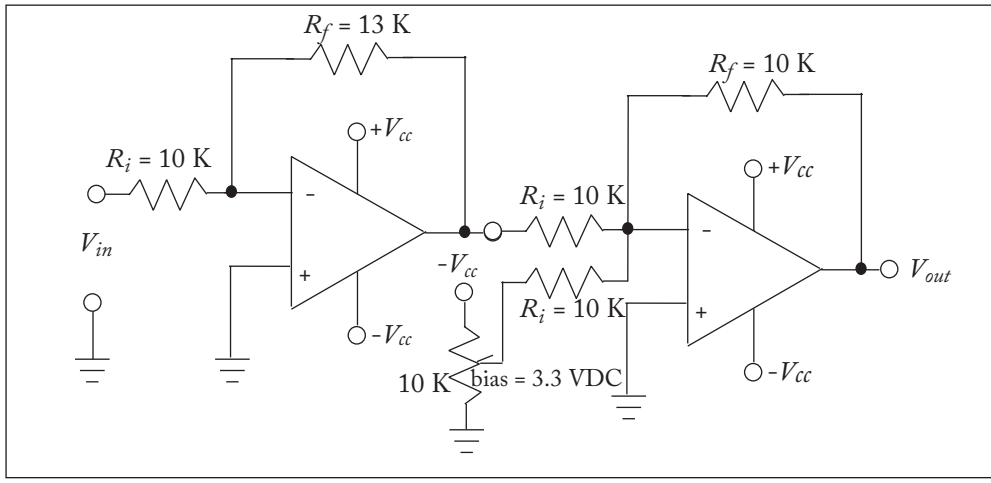


Figure 4.17: Operational amplifier implementation of the TID example circuit.

4.3.1 LIGHT-EMITTING DIODES (LEDS)

A LED is typically used as a logic indicator to inform the presence of a logic one or a logic zero at a specific pin of a microcontroller. An LED has two leads: the anode or positive lead and the cathode or negative lead. To properly bias an LED, the anode lead must be biased at a level approximately 1.7–2.2 volts higher than the cathode lead. This specification is known as the forward voltage (V_f) of the LED. The LED current must also be limited to a safe current level known as the forward current (I_f). The diode voltage and current specifications are usually provided by the manufacturer.

An example of various LED biasing circuit is provided in Figure 4.18. In Figure 4.18a, a logic one asserted by the microcontroller provides the voltage to forward bias the LED. The

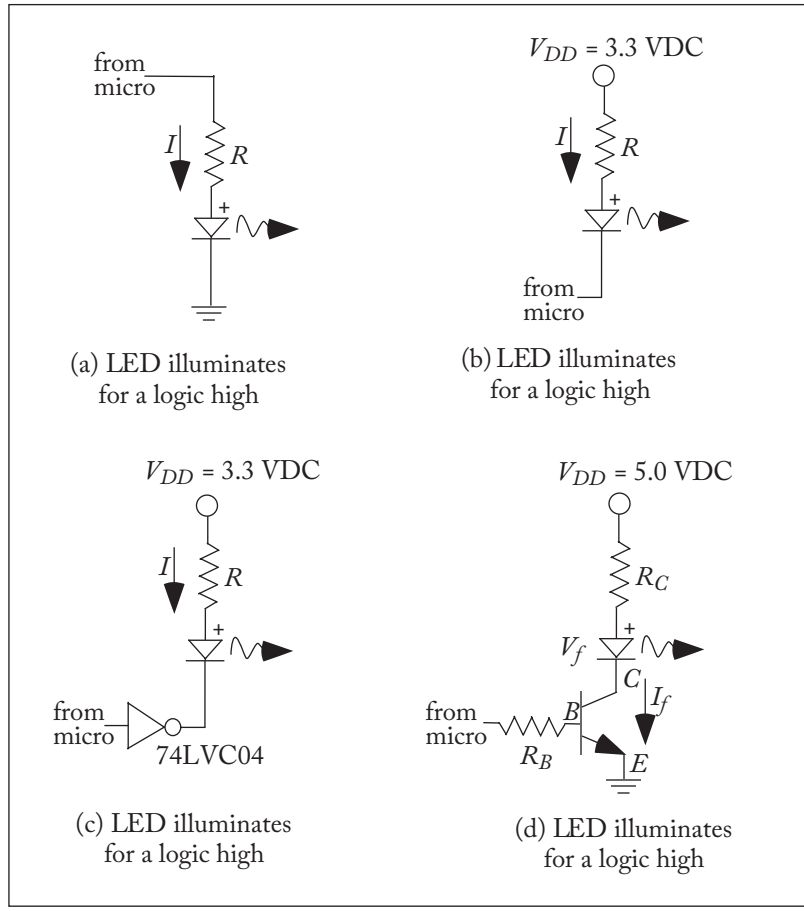


Figure 4.18: Interfacing an LED.

microcontroller also acts as the source for the forward current through the LED. To properly bias the LED, the value of the limit resistor (R) is chosen. Also, we must insure the microcontroller can safely supply the voltage and current to the LED using Figure 4.1.

Example: A red (635 nm) LED is rated at 1.8 VDC with a forward operating current of 10 mA. Design a proper bias for the LED using the configuration of 4.18a.
Answer: In the configuration of Figure 4.18a, the MSP430 microcontroller pin can be viewed as an unregulated power supply. That is, the pin's output voltage is determined by the current supplied by the pin as shown in Figure 4.1. The current flows out of the microcontroller pin through the LED and resistor combination to ground (current source). In this example, we use the MSP430 high-level output voltage characteristics provided at Figure 4.1b. When supplying 10 mA in the logic high case, the high-level output voltage drops to approximately 2.25 VDC. The value of R may be calculated using Ohm's Law. The voltage drop across the resistor is the difference between the 2.25 VDC supplied by the microcontroller pin and the LED forward voltage of 1.8 VDC. The current flowing through the resistor is the LED's forward current (10 mA). This renders a resistance value of approximately 45 ohms. The nearest standard resistor value is 47 ohms. The resistor's required power rating is determined using $P = V \times I$. This yields a value of 4 mW. An eighth watt rated resistor may be used.

For the LED interface provided in Figure 4.18b, the LED is illuminated when the microcontroller provides a logic low. In this case, the current flows from the power supply back into the microcontroller pin (current sink). As before, the MSP microcontroller parameters provided in Figure 4.1 must be used. For the logic low case, the characteristic curve for the low-level output voltage must be used (Figure 4.1, upper subfigure).

If LEDs with higher forward voltages and currents are used, alternative interface circuits may be employed. Figures 4.18c,d provide two more LED interface circuits. In Figure 4.18c, a logic one is provided by the microcontroller to the input of the inverter. The inverter generates a logic zero at its output, which provides a virtual ground at the cathode of the LED. Therefore, the proper voltage biasing for the LED is provided. The resistor (R) limits the current through the LED. A proper resistor value can be calculated using $R = (V_{DD} - V_{DIODE})/I_{DIODE}$. It is important to note that the inverter used must have sufficient current sink capability (I_{OL}) to safely handle the forward current requirements of the LED. As in previous examples, the characteristic curves of the inverter must be carefully analyzed.

An NPN transistor such as a 2N2222 (PN2222 or MPQ2222) may be used in place of the inverter as shown in Figure 4.18d. In this configuration, the transistor is used as a switch. When a logic low is provided by the microcontroller, the transistor is in the cutoff region. When a logic one is provided by the microcontroller, the transistor is driven into the saturation region. To properly interface the microcontroller to the LED, resistor values R_B and R_C must be chosen. The resistor R_B is chosen to limit the base current.

Example: Using the interface configuration of Figure 4.18d, design an interface for an LED with V_f of 2.2 VDC and I_f of 20 mA.

Answer: In this example, we use the current vs. voltage characteristics of the MSP430 (reference Figure 4.1b). If we choose an I_{OH} value of 2 mA, the V_{OH} value will be approximately 2.8 VDC. A loop equation, which includes these parameters, may be written as:

$$V_{OH} = (I_B \times R_B) + V_{BE}.$$

The transistor V_{BE} is typically 0.7 VDC. Therefore, all equation parameters are known except R_B. Solving for R_B yields a value of 1050 ohms. The closest standard value is 1 kOhm.

In this interface configuration, resistor R_C is chosen to safely limit the forward LED current to prescribed values. A loop equation may be written that includes R_C:

$$V_{CC} - (I_f \times R_C) - V_f - V_{CE(sat)} = 0.$$

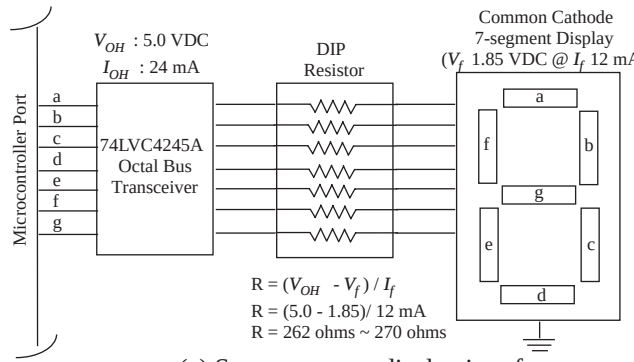
A typical value for $V_{CE(sat)}$ is 0.2 VDC. All equation values are known except R_C. The equation may be solved rendering an R_C value of 130 ohms. For the PN2222 transistor, β is typically 100, insuring the transistor is driven into saturation when a logic one is provided by the MSP430 [Sedra and Smith, 2004].

4.3.2 SEVEN-SEGMENT LED DISPLAYS

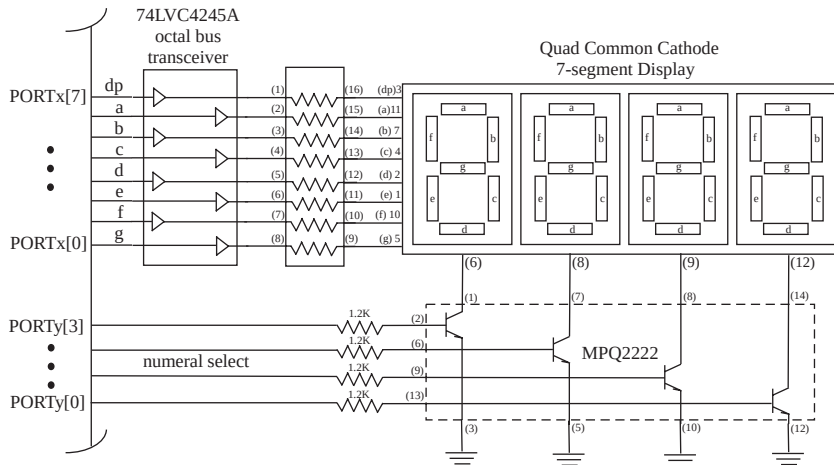
To display numeric data, seven-segment LED displays are available as shown in Figure 4.19b. Different numerals can be displayed by asserting the proper LED segments. For example, to display the number five, segments a, c, d, f, and g would be illuminated; see Figure 4.19a. Seven segment displays are available in common cathode (CC) and common anode (CA) configurations. As the CC designation implies, all seven individual LED cathodes on the display are tied together.

As shown in Figure 4.19b, an interface circuit is required between the microcontroller and the seven-segment LED. We use a 74LVC4245A octal bus transceiver circuit to translate the 3.3 VDC output from the microcontroller up to 5 VDC and also provide a maximum I_{OH} value of 24 mA. A limiting resistor is required for each segment to limit the current to a safe value for the LED. Conveniently, resistors are available in DIP packages of eight for this type of application.

Seven segment displays are available in multi-character panels. In this case, separate microcontroller ports are not used to provide data to each seven-segment character. Instead, a single port is used to provide character data. A portion of another port is used to sequence through each of the characters as shown in Figure 4.19b. An NPN (for a CC display) transistor is connected to the common cathode connection of each individual character. As the base contact of each transistor is sequentially asserted, the specific character is illuminated. If the microcontroller sequences through the display characters at a rate greater than 30 Hz, the display will have steady illumination.



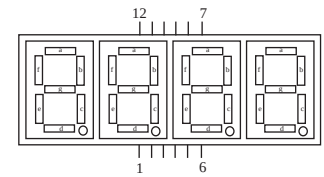
(a) Seven segment display interface



(b) Quad seven segment display interface

Numeral	dp PORTx[7]	a PORTx[6]	b PORTx[5]	c PORTx[4]	d PORTx[3]	e PORTx[2]	f PORTx[1]	g PORTx[0]	hex rep
0		1	1	1	1	1	1	0	0x7E
1		0	1	1	0	0	0	0	0x30
2		1	1	0	1	1	0	1	0x6D
3		1	1	1	1	0	0	1	0x79
4		0	1	1	0	0	1	1	0x33
5		1	0	1	1	0	1	1	0x5D
6		0	0	1	1	1	1	1	0x1F
7		1	1	1	0	0	0	0	0x70
8		1	1	1	1	1	1	1	0x7F
9		1	1	1	0	0	1	1	0x73

(c) Numeral to segment converion



(d) Quad seven segment display pinout
UN(M)5624-11 EWRS

Figure 4.19: LED display devices.

4.3.3 TRI-STATE LED INDICATOR

A tri-state LED indicator is shown in Figure 4.20. It is used to provide the status of an entire microcontroller port. The indicator bank consists of eight green and eight red LEDs. When an individual port pin is logic high the green LED is illuminated. When logic low, the red LED is illuminated. If the port pin is at a tri-state, high impedance state, no LED is illuminated. Tri-state logic is used to connect a number of devices to a common bus. When a digital circuit is placed in the Hi-z (high impedance) state it is electrically isolated from the bus.

The NPN/PNP transistor pair at the bottom of the figure provides a 2.5 VDC voltage reference for the LEDs. When a specific port pin is logic high, the green LED will be forward biased since its anode will be at a higher potential than its cathode. The 47 ohm resistor limits current to a safe value for the LED. Conversely, when a specific port pin is at a logic low (0 VDC), the red LED will be forward biased and illuminate. For clarity, the red and green LEDs are shown as being separate devices. LEDs are available that have both LEDs in the same device. The 74LVC4245A octal bus transceiver translates the output voltage of the microcontroller from 3.3–5.0 VDC.

4.3.4 DOT MATRIX DISPLAY

The dot matrix display consists of a large number of LEDs configured in a single package. A typical 5 x 7 LED arrangement is a matrix of five columns of LEDs with seven LEDs per row, as shown in Figure 4.21. Display data for a single matrix column [R6-R0] is provided by the microcontroller. That specific row is then asserted by the microcontroller using the column select lines [C2-C0]. The entire display is sequentially built up a column at a time. If the microcontroller sequences through each column fast enough (greater than 30 Hz), the matrix display appears to be stationary to a viewer.

In Figure 4.21, we have provided the basic configuration for the dot matrix display for a single display device. However, this basic idea can be expanded in both dimensions to provide a multi-character, multi-line display. A larger display does not require a significant number of microcontroller pins for the interface. The dot matrix display may be used to display alphanumeric data as well as graphics data. Several manufacturers provide 3.3 VDC compatible dot matrix displays with integrated interface and control circuitry.

4.3.5 LIQUID CRYSTAL DISPLAY (LCD)

An LCD is an output device to display text information as shown in Figure 4.22. LCDs come in a wide variety of configurations including multi-character, multi-line format. A 16 x 2 LCD format is common. That is, it has the capability of displaying two lines of 16 characters each. The characters are sent to the LCD via American Standard Code for Information Interchange (ASCII) format a single character at a time. For a parallel configured LCD, an 8-bit data path and two lines are required between the microcontroller and the LCD as shown in Figure 4.22a. Many parallel configured LCDs may also be configured for a 4-bit data path thus saving several

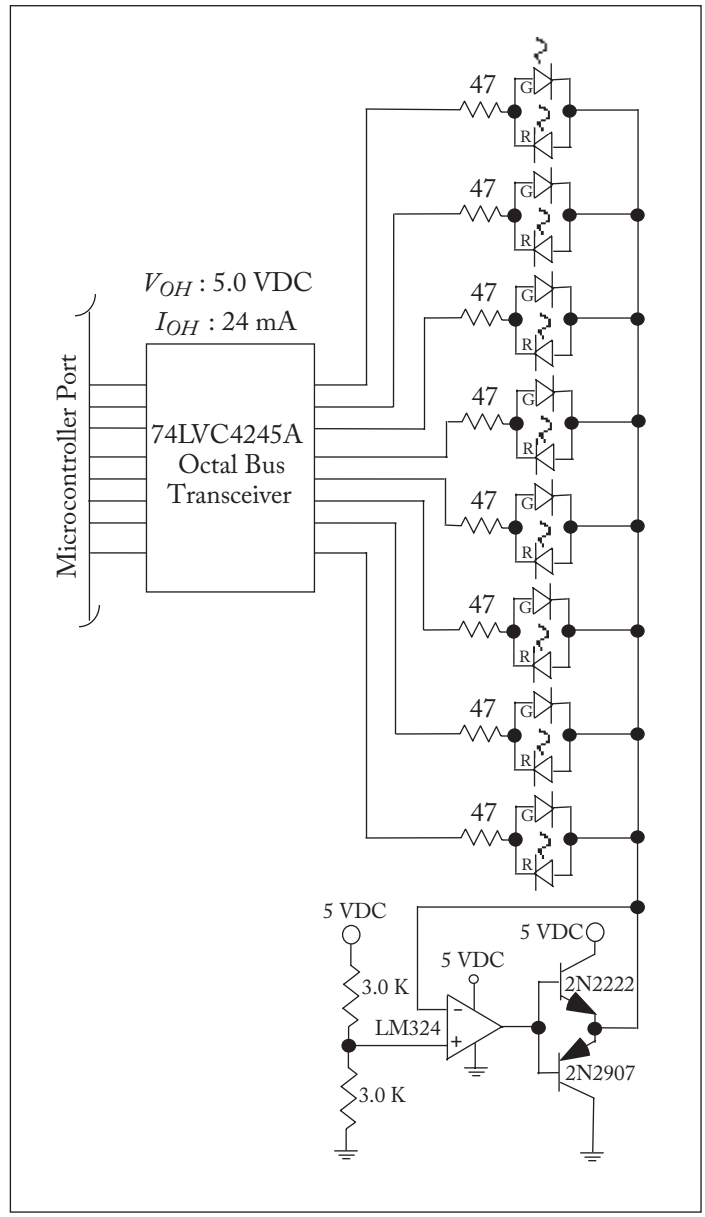


Figure 4.20: Tri-state LED display.

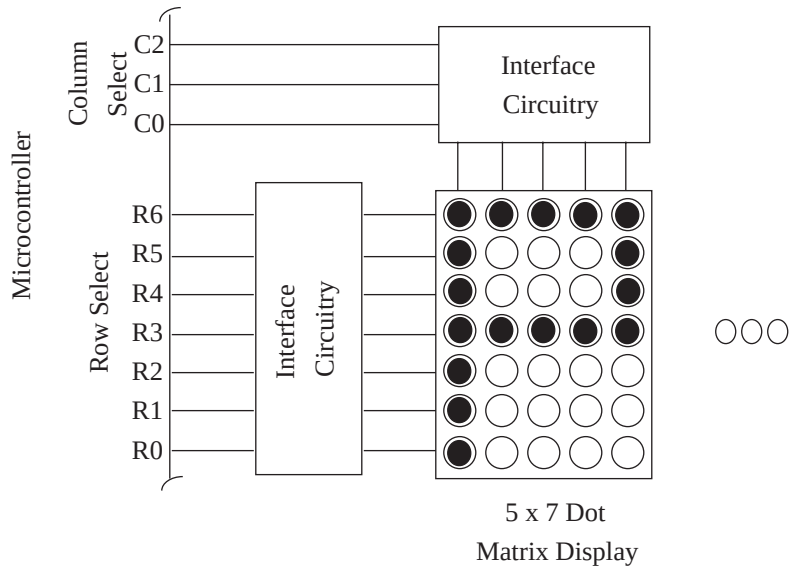


Figure 4.21: Dot matrix display.

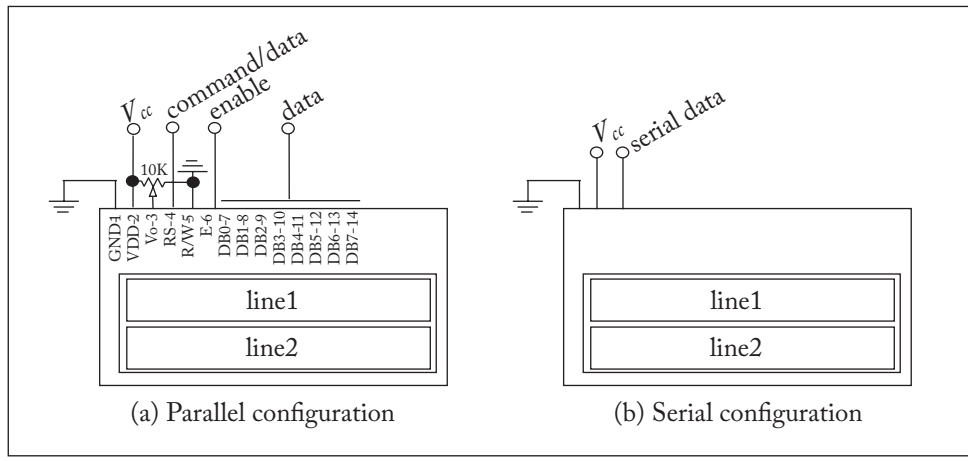


Figure 4.22: LCD display with (a) parallel interface and (b) serial interface.

precious microcontroller pins. A small microcontroller mounted to the back panel of the LCD translates the ASCII data characters and control signals to properly display the characters. Several manufacturers provide 3.3 VDC compatible displays.

To conserve precious, limited microcontroller I/O pins, a serial configured LCD may be used. A serial LCD reduces the number of required microcontroller pins for interface, from ten down to one, as shown in Figure 4.22b. Display data and control information is sent to the LCD via an asynchronous UART serial communication link (8 data bits, 1 stop bit, no parity, 9600 Baud). A serial configured LCD costs slightly more than a similarly configured parallel LCD.

Example: LCD. In this example a Sparkfun LCD-09067, 3.3 VDC, serial, 16 by 2 character, black on white LCD display is connected to the MSP430. Communication between the MSP430 and the LCD is accomplished by a single 9600 bits per second (baud) connection using the onboard universal asynchronous receiver transmitter (UART). The UART is configured for 8 data bits, no parity, and one stop bit (8-N-1). The MSP-EXP430FR5994 LaunchPad is equipped with two UART channels. One is the back channel UART connection to the PC. The other is accessible by pin 3 (RX, P6.1) and pin 4 (TX, P6.0). Provided below is the sample Energia code to print a test message to the LCD. Note the UART is designated "Serial1" in the program. The back channel UART for the Energia serial monitor display is designated "Serial."

```
//*****************************************************************
//Serial_LCD_energia
//Serial 1 accessible at:
// - RX: P6.1, pin 3
// - TX: P6.0, pin 4
//
//This example code is in the public domain.
//*****************************************************************

void setup()
  {
  //Initialize serial channel 1 to 9600 baud and wait for port to open
  Serial1.begin(9600);
  }

void loop()
  {
  Serial1.print("Hello World");
  delay(500);
  Serial1.println("...Hello World");
  delay(500);
  }
```

```
//*******************************************************************
```

4.4 HIGH-POWER DC INTERFACES

There are a wide variety of DC motor types that may be controlled by a microcontroller. To properly interface a motor to the microcontroller, we must be familiar with the different types of motor technologies. Motor types are illustrated in Figure 4.23.

General categories of DC motor types include the following.

- **DC motor:** A DC motor has a positive and a negative terminal. When a DC power supply of suitable current rating is applied to the motor, it will rotate. If the polarity of the supply is switched with reference to the motor terminals, the motor will rotate in the opposite direction. The speed of the motor is roughly proportional to the applied voltage up to the rated voltage of the motor.

- **Servo motor:** A servo motor provides a precision angular rotation for an applied pulse width modulation duty cycle. As the duty cycle of the applied signal is varied, the angular displacement of the motor also varies. This type of motor is used to change mechanical positions such as the steering angle of a wheel.

- **Stepper motor:** A stepper motor, as its name implies, provides an incremental step change in rotation (typically 2.5° per step) for a step change in control signal sequence. The motor is typically controlled by a two or four wire interface. For the four wire stepper motor, the microcontroller provides a 4-bit control sequence to rotate the motor clockwise. To turn the motor counterclockwise, the control sequence is reversed. The low-power control signals are interfaced to the motor via metal oxide semiconductor field effect transistors (MOSFETs) or power transistors to provide for the proper voltage and current requirements of the pulse sequence.

- **Linear actuator:** A linear actuator translates the rotation motion of a motor to linear forward and reverse movement. The actuators are used in a number of different applications where precisely controlled linear motion is required. The control software and interface for linear actuators are very similar to DC motors.

Example: DC motor interface. A general-purpose DC motor interface is provided in Figure 4.24. This interface allows the low-voltage (3.3 VDC), low-current control signal to be interfaced to a higher voltage, higher current motor. This interface provides for unidirectional control. To control motor speed, PWM techniques may be used. The control signal from the MSP430 is fed to the TIP 120 NPN Darlington transistor. The Darlington configuration allows high current gain to drive the motor. Diodes are placed in series with the motor to reduce

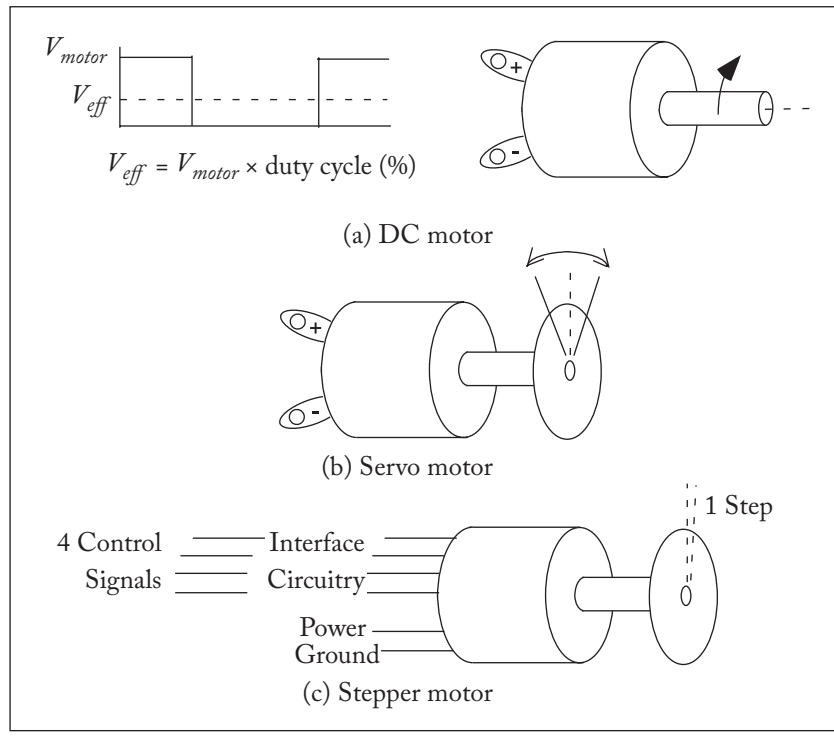


Figure 4.23: Motor types.

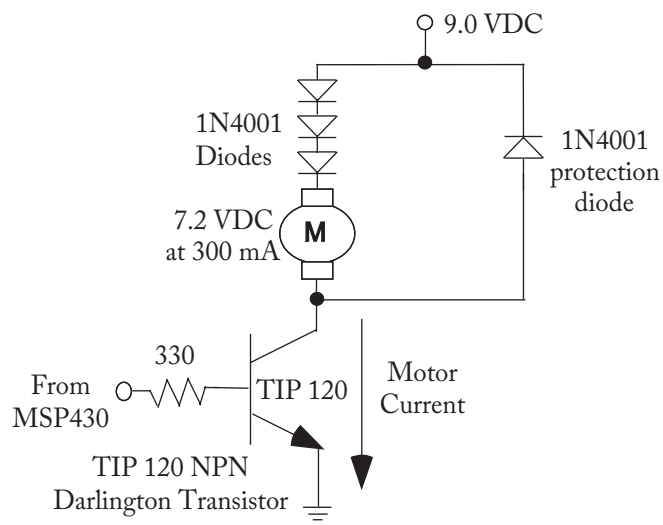


Figure 4.24: General-purpose motor interface.

the motor supply voltage to the required motor voltage. Each diode provides a drop of approximately 0.7 VDC. A reverse biased diode is placed across the motor and diode string to allow a safe path for reverse current. This configuration may be adjusted for many types of DC motors by appropriately adjusting supply voltage, number of series diodes, and the value of the base resistance.

Example: Inexpensive laser light show. An inexpensive laser light show can be constructed using two servos. This application originally appeared in the third edition of *Arduino Microcontroller Processing for Everyone!* The example has been adapted with permission for compatibility with the MSP430 [Barrett, 2013]. In this example we use two Futaba 180° range servos (Parallax 900-00005, available from Jameco #283021) mounted as shown in Figure 4.25. The servos operates from 4–6 VDC. The servos expect a pulse every 20 ms (50 Hz). The pulse length determines the degree of rotation from 1000 microseconds (5% duty cycle, –90° rota-

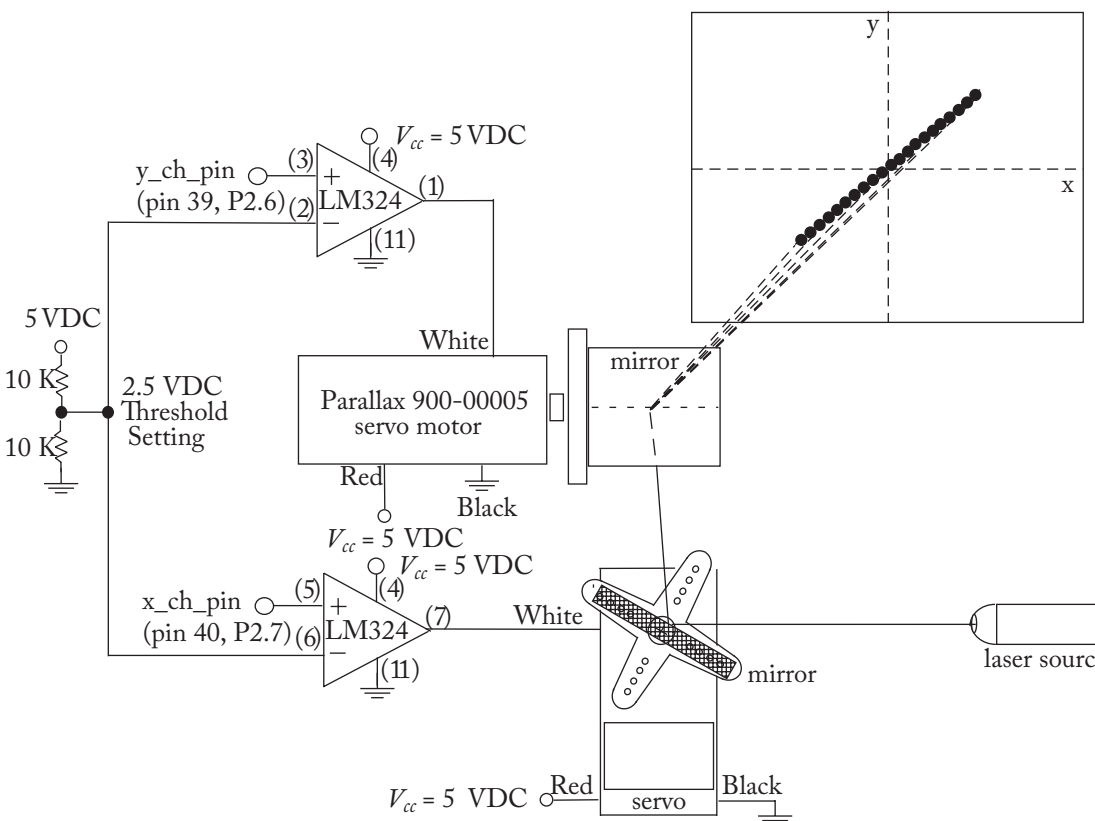


Figure 4.25: Inexpensive laser light show.

tion) to 2000 μs (10% duty cycle, +90° rotation). The X and Y control signals are provided by the MSP430. The X and Y control signals are interfaced to the servos via LM324 operational amplifiers. The 3.3 VDC control signals from the MSP430 are up converted to 5.0 VDC by the op-amps. The op-amps serve as voltage comparators with a 2.5 VDC threshold. The laser source is provided by an inexpensive laser pointer.

Energia contains useful servo configuration and control functions. The "attach" function initializes the servo at the specified pin. The MSP430-EXP430FR5994 LaunchPad has pulse width modulated output features available on pins 19 (P5.7), 37 (P3.4), 38 (P3.5), 39 (P3.6), and 40 (P3.7). The "write" function rotates the servo the specified number of degrees. The program sends the same signal to both channel outputs (x_ch_pin, y_ch_pin) and traces a line with the laser. Any arbitrary shape may be traced by the laser using this technique.

```
//*************************************************************
//X-Y ramp
//
//This example code is in the public domain.
//*************************************************************

#include <Servo.h>          //Use Servo library, included with IDE

Servo myServo_x;            //Create Servo objects to control the
Servo myServo_y;            //X and Y servos

void setup()
  {
  myServo_x.attach(40);     //Servo is connected to PWM pin 40
  myServo_y.attach(39);     //Servo is connected to PWM pin 39
  }

void loop()
{
int i = 0;
for(i=0; i<=180; i++)       //Rotates servo 0 to 180 degrees
  {
  myServo_x.write(i);       //Rotate servo counter clockwise
  myServo_y.write(i);       //Rotate servo counter clockwise
  delay(20);                //Wait 20 milliseconds
  if(i==180)
    delay(5000);
  }
```

```
}

//************************************************************
```

4.4.1 DC MOTOR INTERFACE, SPEED, AND DIRECTION CONTROL

Interface. A number of direct current load devices are controlled with an electronic switching device such as a MOSFET. Specifically, an N-channel enhancement MOSFET may be used to switch a high current load on and off (such as a motor) using a low-current control signal from a microcontroller, as shown in Figure 4.26. The low current control signal from the microcontroller is connected to the gate of the MOSFET via a MOSFET driver. As shown in Figure 4.26, an LTC 1157 MOSFET driver is used to boost the control signal from the microcontroller to be compatible with an IRLR024 power MOSFET. The IRLR024 is rated at 60 VDC V_{DS} and a continuous drain current I_D of 14 amps. The IRLR024 MOSFET switches the high current load on and off consistent with the control signal. In a low-side connection, the high current load is connected between the MOSFET source and ground.

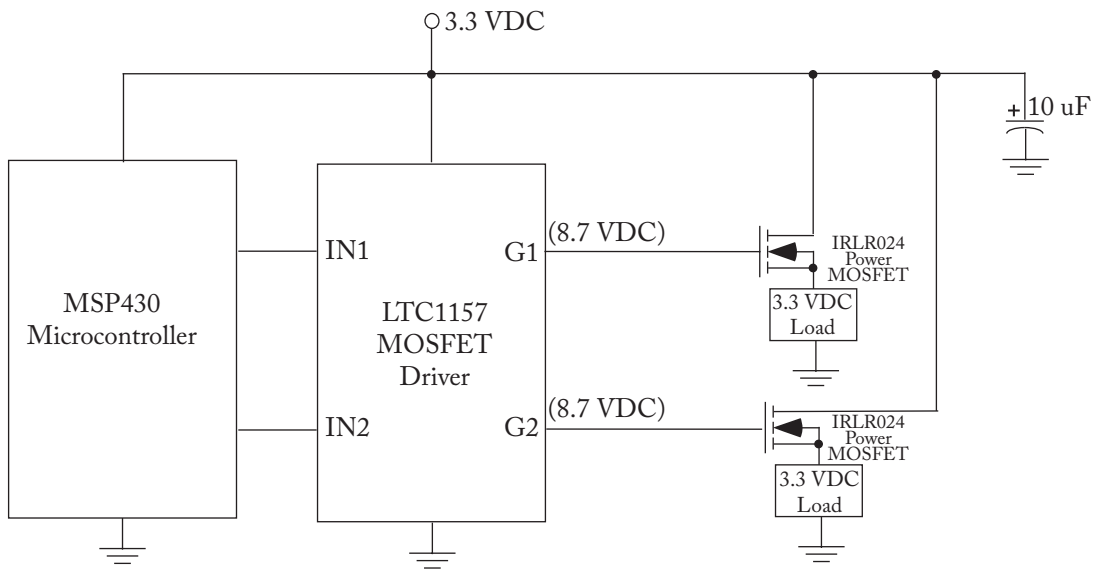


Figure 4.26: MOSFET drive circuit (adapted from Linear Technology).

Speed. As previously mentioned, DC motor speed may be varied by changing the applied voltage. This is difficult to do with a digital control signal. However, PWM techniques combined with a MOSFET interface circuit may be used to precisely control motor speed. The duty cycle of the PWM signal governs the percentage of the motor supply voltage applied to the motor and hence the percentage of rated full speed at which the motor will rotate. The interface circuit

to accomplish this type of control is shown in Figure 4.27. It is a slight variation of the control circuit provided in Figure 4.26. In this configuration, the motor supply voltage may be different than the microcontroller's 3.3 VDC supply. For an inductive load, a reverse biased protection diode should be connected across the load. The interface circuit allows the motor to rotate in a given direction.

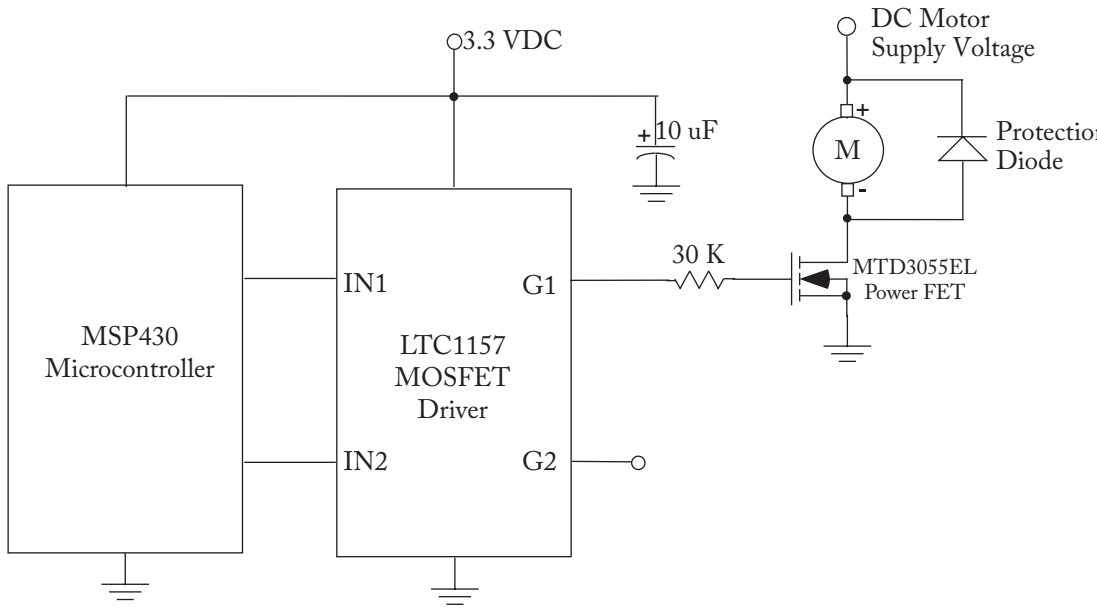


Figure 4.27: DC motor interface.

Direction. For a DC motor to operate in both the clockwise and counterclockwise directions, the polarity of the DC motor supplied must be changed. To operate the motor in the forward direction, the positive battery terminal must be connected to the positive motor terminal while the negative battery terminal must be attached to the negative motor terminal. To reverse the motor direction, the motor supply polarity must be reversed. An H-bridge is a circuit employed to perform this polarity switch. An H-bridge may be constructed from discrete components as shown in Figure 4.28. If PWM signals are used to drive the base of the transistors, both motor speed and direction may be controlled by the circuit. The transistors used in the circuit must have a current rating sufficient to handle the current requirements of the motor during start and stall conditions.

Texas Instruments provides a self-contained H-bridge motor controller integrated circuit, the DRV8829. Within the DRV8829 package is a single H-bridge driver. The driver may control DC loads with supply voltages from 8–45 VDC with a peak current rating of 5 amps. The

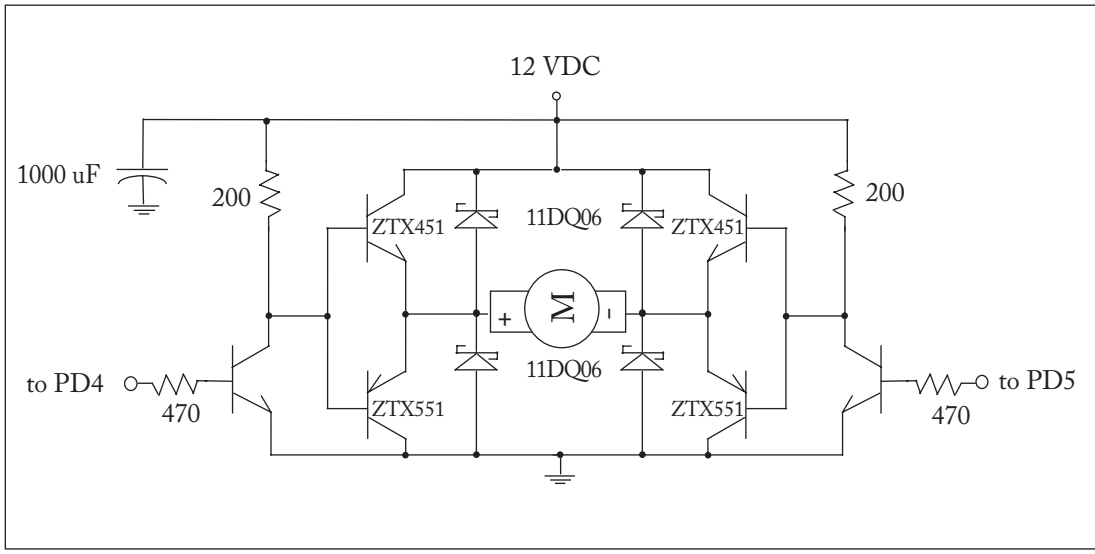


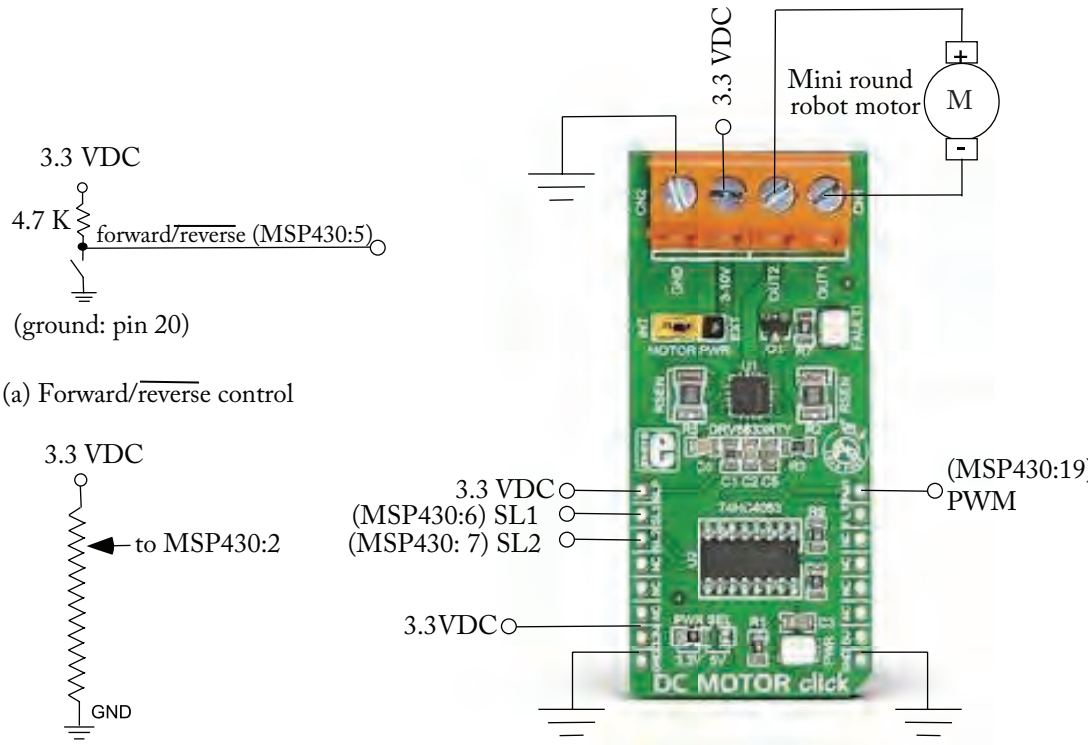
Figure 4.28: H-bridge control circuit.

single H-bridge driver may be used to control a DC motor or one winding of a bipolar stepper motor [DRV8829].

Example: MIKROE-1526 DC MOTOR click. MikroElectronica (www.mikroe.com) manufactures a number of motor interface products including the MIKROE-1526 DC MOTOR click motor driver board. The board features the T.I. DRV8833RTY H-bridge motor driver. A test circuit to control a DC motor's speed and direction is provided in Figure 4.29. We use one of the motors from the Mini Round Autonomous Maze Navigating Robot (Chapter 2).

In the test circuit, a tach switch is used to determine motor direction and a potentiometer for motor speed control. These two inputs are read by the Energia program and proper control signals are issued to the MIKROE-1526 (SL1, SL2, and PWM) for motor speed and direction. The nSLP pin on the MIKROE-1526 must be logic high to enable the devive. For the test circuit, the pin is tied to Vcc (3.3 VDC) (www.mikroe.com).

```
//*****************************************************************
//MIKROE-1526 DC Motor click
//Sketch demonstrates operation of the MIKROE-1526 DC Motor click
//
//The circuit:
//  - Motor speed control potentiometer.
//    Potentiometer connected to analog pin 0 (2).  The center wiper
//    pin of the potentiometer goes to the analog pin.  The side
```



(a) Forward/reverse control

(b) Speed control

(c) MIKROE-1526 DC MOTOR click

Figure 4.29: MIKROE-1526 DC MOTOR click. (Illustration used with permission (www.mikroe.com).)

```
//    pins of the potentiometer go to +3.3 VDC and ground.
//  - Motor forward/reverse control.
//    Tact switch connected to pin 5 of MSP430.
//  - MIKROE-1526 connections:
//    -- Select 1 (SL1) to MSP430 pin 6
//    -- Select 2 (SL2) to MSP430 pin 7
//    -- PWM to MSP430 pin 19
//    -- nSLEEP to Vcc (3.3 VDC) to enable device
//
//This example code is in the public domain.
//*****************************************************************

int analog_in = 2;                  //analog input (0 to 1023)
int analog_out = 19;                 //analog output (0 to 255)
int forward_reverse = 5;            //direction control
int select1 = 6;                    //motor direction control
int select2 = 7;                    //SL1, SL2
int speed_value;                    //potentiometer input value
int switch_value;
int output_value;

void setup()
{
pinMode(forward_reverse, INPUT);
pinMode(select1, OUTPUT);
pinMode(select2, OUTPUT);
}

void loop()
{
//Deteremine motor direction
switch_value = digitalRead(forward_reverse);
if(switch_value == HIGH)   //forward direction
  {
  digitalWrite(select1, LOW);
  digitalWrite(select2, LOW);
  }
else                       //reverse direction
  {
```

```
  digitalWrite(select1, LOW);
  digitalWrite(select2, HIGH);
  }

//read analog in value
speed_value = analogRead(analog_in);

//map to analog out range
output_value = map(speed_value, 0, 1023, 0, 255);

//update analog out value
analogWrite(analog_out, output_value);

delay(50);
}

//*************************************************************
```

4.4.2 DC SOLENOID CONTROL

The interface circuit for a DC solenoid is shown in Figure 4.30. A solenoid is used to activate a mechanical insertion (or extraction). As in previous examples, we employ the LTC1157 MOSFET driver between the microcontroller and the power MOSFET used to activate the solenoid. A reverse biased diode is placed across the solenoid. Both the solenoid power supply and the MOSFET must have an appropriate voltage and current rating to support the solenoid requirements.

4.4.3 STEPPER MOTOR CONTROL

Stepper motors are used to provide a discrete angular displacement in response to a control signal step. There are a wide variety of stepper motors including bipolar and unipolar types with different configurations of motor coil wiring. Due to space limitations we only discuss the unipolar, 5-wire stepper motor. The internal coil configuration for this motor is shown in Figure 4.31b.

Often, a wiring diagram is not available for the stepper motor. Based on the wiring configuration (Figure 4.31b), one can find out the common line for both coils. It has a resistance that is one-half of all of the other coils. Once the common connection is found, one can connect the stepper motor into the interface circuit. By changing the other connections, one can determine the correct connections for the step sequence. To rotate the motor either clockwise

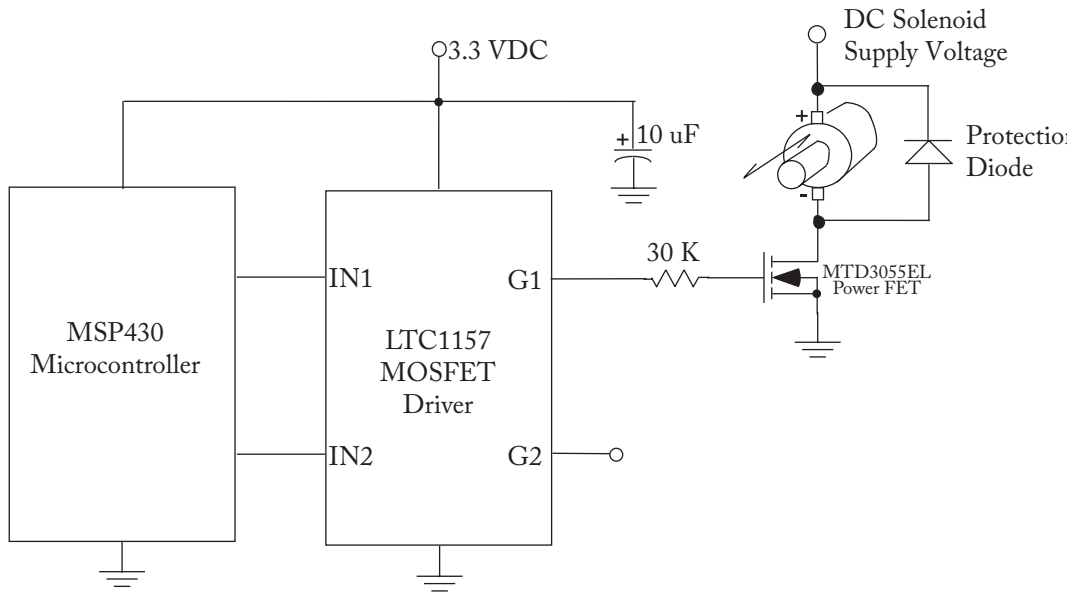
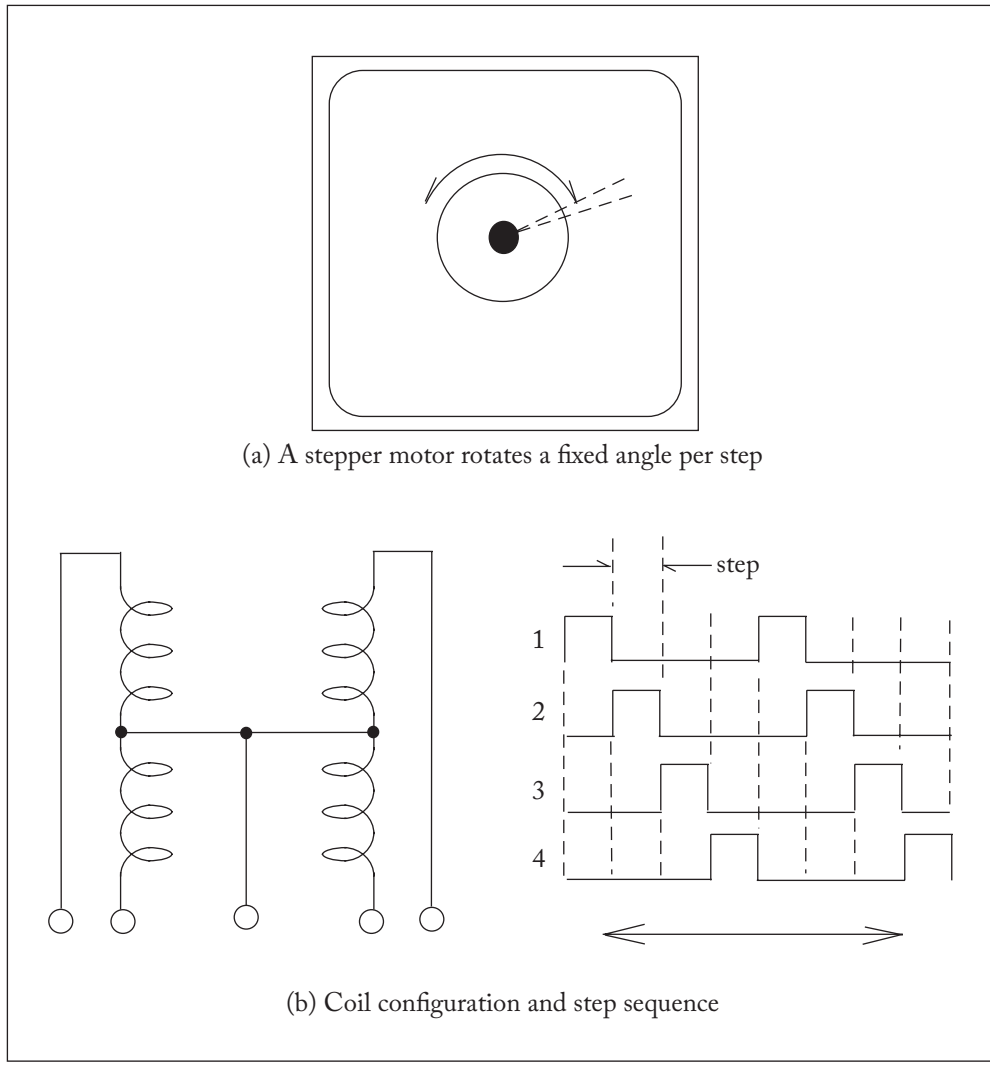


Figure 4.30: Solenoid interface circuit.

or counterclockwise, a specific step sequence must be sent to the motor control wires as shown in Figure 4.31b.

The microcontroller does not have sufficient capability to drive the motor directly. Therefore, an interface circuit is required as shown in Figure 4.32. The speed of motor rotation is determined by how fast the control sequence is completed.

```
//*************************************************************
//stepper
//
//This example code is in the public domain.
//*************************************************************

//external switches
#define ext_sw1  36
#define ext_sw2  35

//stepper channels
#define stepper_ch1  31
#define stepper_ch2  32
#define stepper_ch3  33
#define stepper_ch4  34
```



(a) A stepper motor rotates a fixed angle per step

(b) Coil configuration and step sequence

Figure 4.31: Unipolar stepper motor. (Illustration used with permission of Texas Instruments (www.ti.com).)

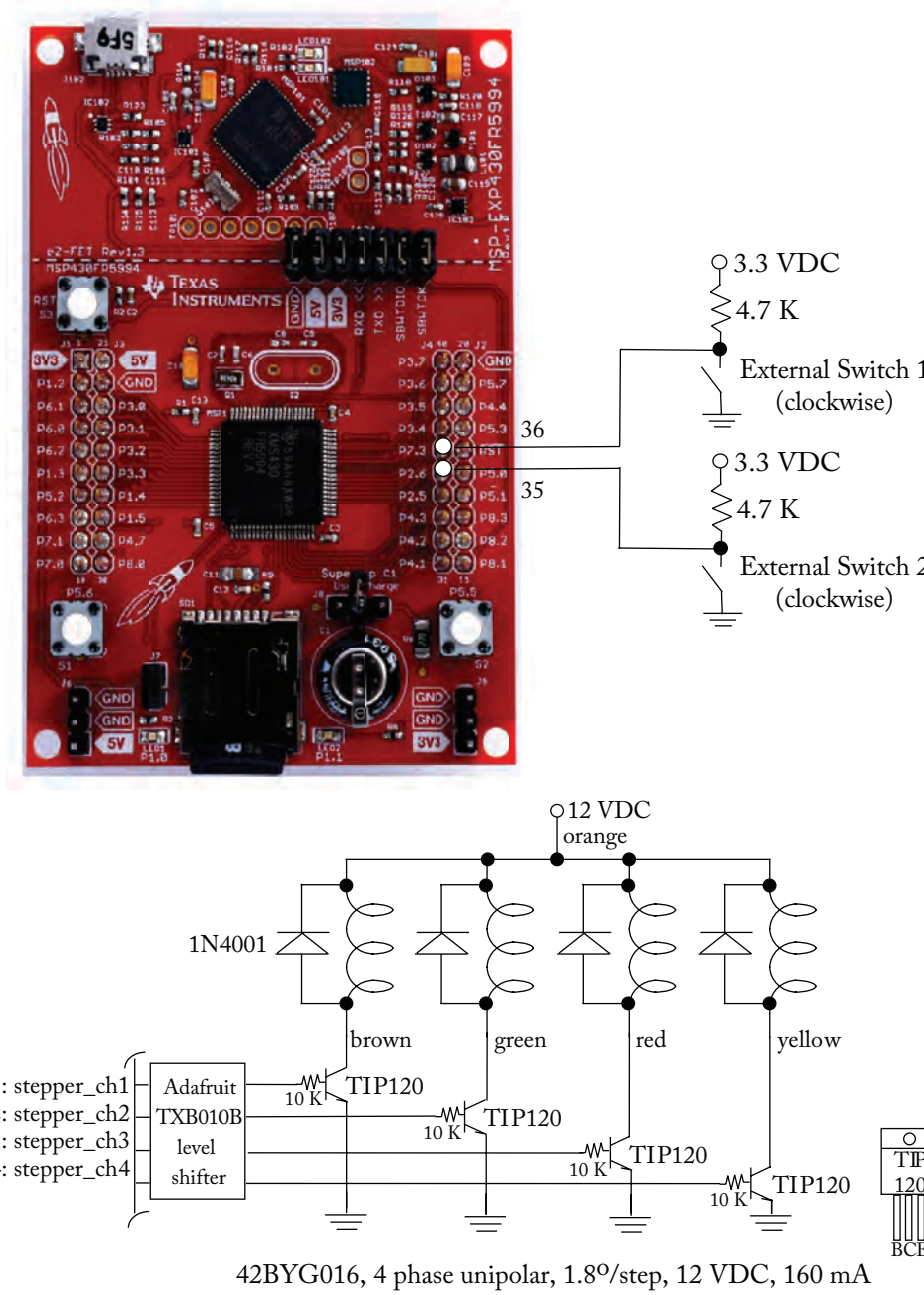


Figure 4.32: Unipolar stepper motor interface circuit.

```
int switch_value1, switch_value2;
int motor_speed = 1000;           //motor increment time in ms
int last_step = 1;
int next_step;

void setup()
{
//Screen
Serial.begin(9600);

//external switches
pinMode(ext_sw1,  INPUT);
pinMode(ext_sw2,  INPUT);

//stepper channel
pinMode(stepper_ch1, OUTPUT);
pinMode(stepper_ch2, OUTPUT);
pinMode(stepper_ch3, OUTPUT);
pinMode(stepper_ch4, OUTPUT);

}

void loop()
{
switch_value1 = digitalRead(ext_sw1);
switch_value2 = digitalRead(ext_sw2);

if(switch_value1 == LOW)          //switch1 asserted
  {
  while(switch_value1 == LOW)     //clockwise
    {
    if(last_step == 1)
      {
      Serial.println("Switch 1: low, step 1");
      digitalWrite(stepper_ch1, HIGH);
      digitalWrite(stepper_ch2, LOW);
      digitalWrite(stepper_ch3, LOW);
      digitalWrite(stepper_ch4, LOW);
```

```
    next_step = 2;
    }
  else if(last_step == 2)
    {
    Serial.println("Switch 1: low, step 2");
    digitalWrite(stepper_ch1, LOW);
    digitalWrite(stepper_ch2, HIGH);
    digitalWrite(stepper_ch3, LOW);
    digitalWrite(stepper_ch4, LOW);
    next_step = 3;
    }
  else if(last_step == 3)
    {
    Serial.println("Switch 1: low, step 3");
    digitalWrite(stepper_ch1, LOW);
    digitalWrite(stepper_ch2, LOW);
    digitalWrite(stepper_ch3, HIGH);
    digitalWrite(stepper_ch4, LOW);
    next_step = 4;
    }
  else if(last_step == 4)
    {
    Serial.println("Switch 1: low, step 4");
    digitalWrite(stepper_ch1, LOW);
    digitalWrite(stepper_ch2, LOW);
    digitalWrite(stepper_ch3, LOW);
    digitalWrite(stepper_ch4, HIGH);
    next_step = 1;
    }
  else
    {
    ;
    }
  last_step = next_step;
  delay(motor_speed);
  switch_value1 = digitalRead(ext_sw1);
  }//end while
 }//end if
```

```
else if(switch_value2 == LOW)      //switch2 asserted
  {
  while(switch_value2 == LOW)      //counter clockwise
    {
    if(last_step == 1)
      {
      Serial.println("Switch 2: low, step 1");
      digitalWrite(stepper_ch1, HIGH);
      digitalWrite(stepper_ch2, LOW);
      digitalWrite(stepper_ch3, LOW);
      digitalWrite(stepper_ch4, LOW);
      next_step = 4;
      }
    else if(last_step == 2)
      {
      Serial.println("Switch 2: low, step 2");
      digitalWrite(stepper_ch1, LOW);
      digitalWrite(stepper_ch2, HIGH);
      digitalWrite(stepper_ch3, LOW);
      digitalWrite(stepper_ch4, LOW);
      next_step = 1;
      }
    else if(last_step == 3)
      {
      Serial.println("Switch 2: low, step 3");
      digitalWrite(stepper_ch1, LOW);
      digitalWrite(stepper_ch2, LOW);
      digitalWrite(stepper_ch3, HIGH);
      digitalWrite(stepper_ch4, LOW);
      next_step = 2;
      }
    else if(last_step == 4)
      {
      Serial.println("Switch 2: low, step 4");
      digitalWrite(stepper_ch1, LOW);
      digitalWrite(stepper_ch2, LOW);
      digitalWrite(stepper_ch3, LOW);
      digitalWrite(stepper_ch4, HIGH);
      next_step = 3;
```

```
      }
    else
      {
      ;
      }
    last_step = next_step;
    delay(motor_speed);
    switch_value2 = digitalRead(ext_sw2);
    }//end while
   }//end if

   else
     {
     digitalWrite(stepper_ch1, LOW);
     digitalWrite(stepper_ch2, LOW);
     digitalWrite(stepper_ch3, LOW);
     digitalWrite(stepper_ch4, LOW);
     }
}
//*********************************************************
```

Example. Adafruit (www.adafruit.com) manufactures a DC stepper motor breakout board (#3297) based on the Texas Instruments DRV8833RTY H-bridge motor driver. The board can provide up to 1.2 A per channel for motors from 2.7–10.8 VDC. In this example, we use the board to drive a Jameco (www.jameco.com) #238538 unipolar stepper motor rated at 12 VDC, 0.4 A. We power the motor at 10 VDC. The interface between the MSP-EXP430FR2433 LaunchPad, the breakout board, and the stepper motor in shown in Figure 4.33. Two external switches are used to select motor direction.

The code used in the previous stepper motor example is modified with the step sequence required by the driver/motor combination.

```
//*********************************************************
//stepper2
//
//This example code is in the public domain.
//*********************************************************

//external switches
#define ext_sw1  2
#define ext_sw2  5
```

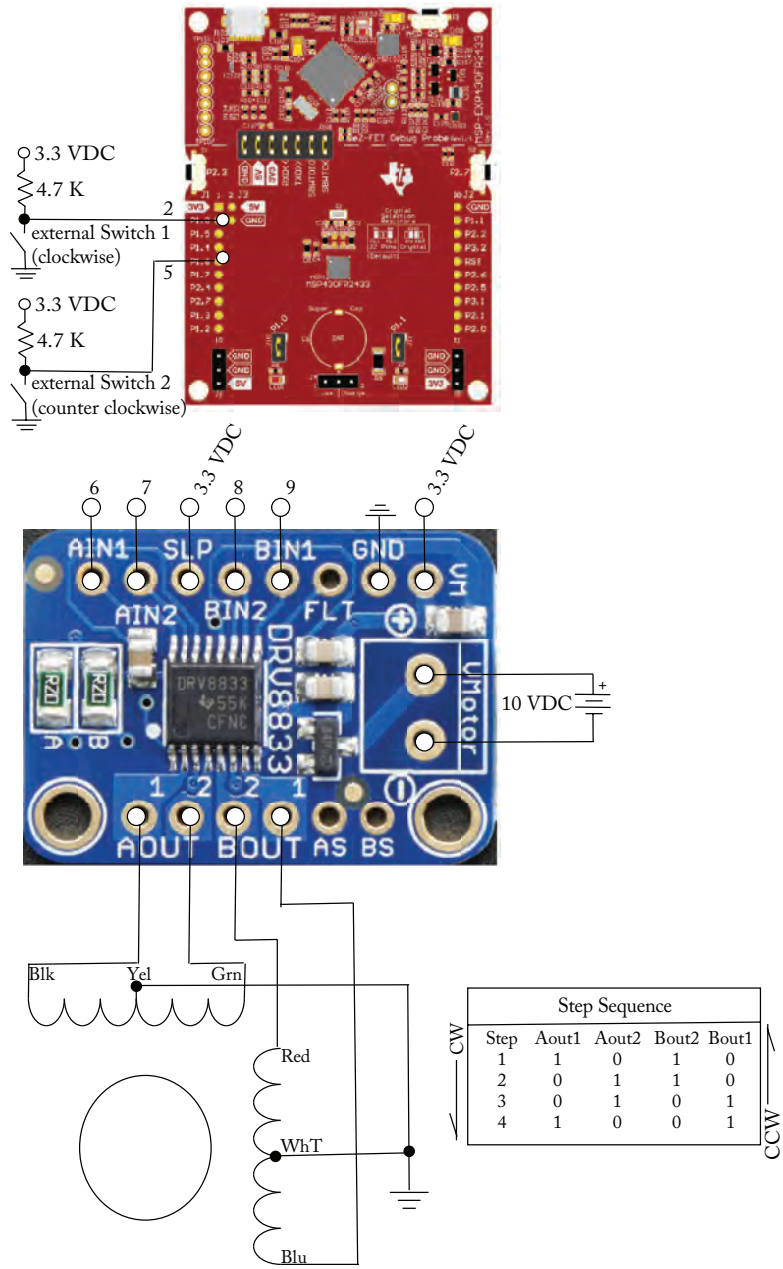


Figure 4.33: Unipolar stepper motor with DRV8833 breakout board. (Illustration used with permission of Texas Instruments (www.ti.com).)

```
//stepper channels
#define stepper_ch1  6
#define stepper_ch2  7
#define stepper_ch3  8
#define stepper_ch4  9

int switch_value1, switch_value2;
int motor_speed = 10;           //motor increment time in ms
int last_step = 1;
int next_step;

void setup()
{
//Screen
Serial.begin(9600);

//external switches
pinMode(ext_sw1,  INPUT);
pinMode(ext_sw2,  INPUT);

//stepper channel
pinMode(stepper_ch1, OUTPUT);
pinMode(stepper_ch2, OUTPUT);
pinMode(stepper_ch3, OUTPUT);
pinMode(stepper_ch4, OUTPUT);

}

void loop()
{
switch_value1 = digitalRead(ext_sw1);
switch_value2 = digitalRead(ext_sw2);

if(switch_value1 == LOW)          //switch1 asserted
  {
  while(switch_value1 == LOW)     //clockwise
    {
    if(last_step == 1)
```

```
  {
  Serial.println("Switch 1: low, step 1");
  digitalWrite(stepper_ch1, HIGH);
  digitalWrite(stepper_ch2, LOW);
  digitalWrite(stepper_ch3, HIGH);
  digitalWrite(stepper_ch4, LOW);
  next_step = 2;
  }
else if(last_step == 2)
  {
  Serial.println("Switch 1: low, step 2");
  digitalWrite(stepper_ch1, LOW);
  digitalWrite(stepper_ch2, HIGH);
  digitalWrite(stepper_ch3, HIGH);
  digitalWrite(stepper_ch4, LOW);
  next_step = 3;
  }
else if(last_step == 3)
  {
  Serial.println("Switch 1: low, step 3");
  digitalWrite(stepper_ch1, LOW);
  digitalWrite(stepper_ch2, HIGH);
  digitalWrite(stepper_ch3, LOW);
  digitalWrite(stepper_ch4, HIGH);
  next_step = 4;
  }
else if(last_step == 4)
  {
  Serial.println("Switch 1: low, step 4");
  digitalWrite(stepper_ch1, HIGH);
  digitalWrite(stepper_ch2, LOW);
  digitalWrite(stepper_ch3, LOW);
  digitalWrite(stepper_ch4, HIGH);
  next_step = 1;
  }
else
  {
  ;
  }
```

```
    last_step = next_step;
    delay(motor_speed);
    switch_value1 = digitalRead(ext_sw1);
    }//end while
   }//end if

else if(switch_value2 == LOW)      //switch2 asserted
  {
  while(switch_value2 == LOW)      //counter clockwise
    {
    if(last_step == 1)
      {
      Serial.println("Switch 2: low, step 1");
      digitalWrite(stepper_ch1, HIGH);
      digitalWrite(stepper_ch2, LOW);
      digitalWrite(stepper_ch3, HIGH);
      digitalWrite(stepper_ch4, LOW);
      next_step = 4;
      }
    else if(last_step == 2)
      {
      Serial.println("Switch 2: low, step 2");
      digitalWrite(stepper_ch1, LOW);
      digitalWrite(stepper_ch2, HIGH);
      digitalWrite(stepper_ch3, HIGH);
      digitalWrite(stepper_ch4, LOW);
      next_step = 1;
      }
    else if(last_step == 3)
      {
      Serial.println("Switch 2: low, step 3");
      digitalWrite(stepper_ch1, LOW);
      digitalWrite(stepper_ch2, HIGH);
      digitalWrite(stepper_ch3, LOW);
      digitalWrite(stepper_ch4, HIGH);
      next_step = 2;
      }
    else if(last_step == 4)
      {
```

```
      Serial.println("Switch 2: low, step 4");aq  1q
      digitalWrite(stepper_ch1, HIGH);
      digitalWrite(stepper_ch2, LOW);
      digitalWrite(stepper_ch3, LOW);
      digitalWrite(stepper_ch4, HIGH);
      next_step = 3;
      }
    else
      {
      ;
      }
    last_step = next_step;
    delay(motor_speed);
    switch_value2 = digitalRead(ext_sw2);
    }//end while
   }//end if

   else
     {
     digitalWrite(stepper_ch1, LOW);
     digitalWrite(stepper_ch2, LOW);
     digitalWrite(stepper_ch3, LOW);
     digitalWrite(stepper_ch4, LOW);
     }
}
//*********************************************************
```

4.4.4 OPTICAL ISOLATION

It is a good design practice to provide optical isolation between a motor control circuit and the motor. A typical optical isolator (e.g., 4N25) consists of an LED and an optical transistor in a common package, as shown in Figure 4.34. The LED is driven by a low voltage control signal from the MSP430, whereas the optical transistor provides the control signal to the motor interface circuit. The link between the MSP430 to the motor interface circuit is now enabled by light rather than an electrical link. This provides a high level of noise isolation between the processor and the motor interface circuit. Many optical isolators also provide a signal inversion.

4.5 INTERFACING TO MISCELLANEOUS DC DEVICES

In this section, we present a potpourri of interface circuits to connect a microcontroller to a wide variety of DC peripheral devices.

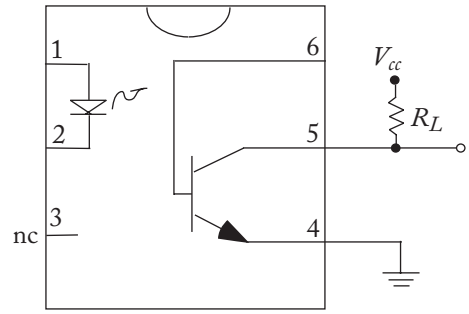
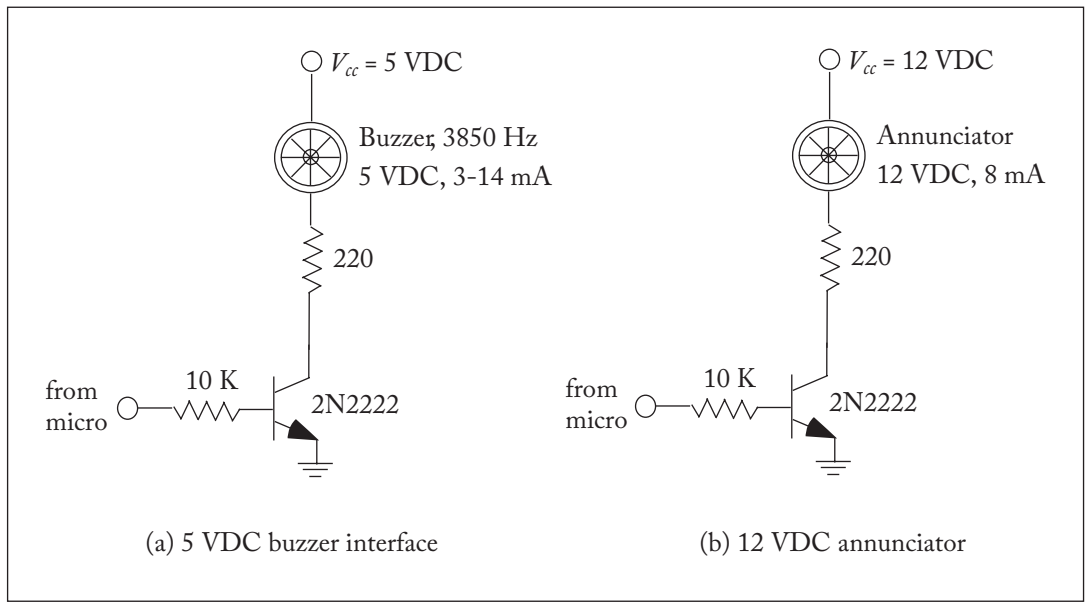


Figure 4.34: 4N25 optical isolator (www.vishay.com).

4.5.1 SONALERTS, BEEPERS, AND BUZZERS

In Figure 4.35, we show several circuits used to interface a microcontroller to a buzzer, beeper, or other types of annunciator devices such as a sonalert. It is important that the interface transistor and the supply voltage are matched to the requirements of the sound producing device.



(a) 5 VDC buzzer interface

(b) 12 VDC annunciator

Figure 4.35: Sonalert, beepers, and buzzers.

4.5.2 VIBRATING MOTOR

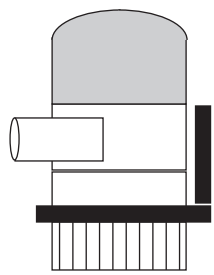
A vibrating motor is often used to gain one's attention as in a cell phone. These motors are typically rated at 3 VDC and a high current. The interface circuit shown in Figure 4.26 is used to drive the low voltage motor.

4.5.3 DC FAN

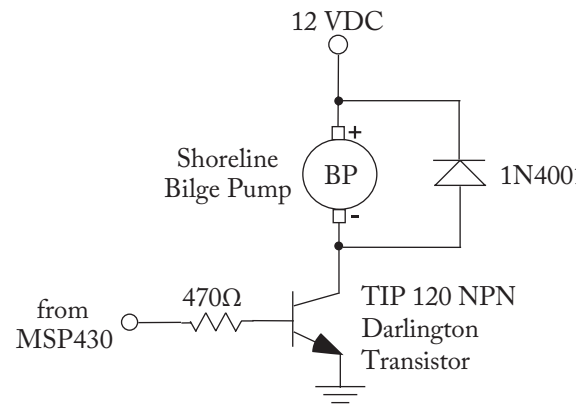
The interface circuit shown in Figure 4.24 may also be used to control a DC fan. As before, a reverse biased diode is placed across the DC fan motor.

4.5.4 BILGE PUMP

A bilge pump is a pump specifically designed to remove water from the inside of a boat. The pumps are powered from a 12 VDC source and have typical flow rates from 360 to over 3,500 gallons per minute. They range in price from U.S. $20–$80 (`www.shorelinemarinedevelopment.com`). An interface circuit to control a bilge pump from MSP430 is shown in Figure 4.36. The interface circuit consists of a 470 ohm resistor, a power NPN Darlington transistor (TIP 120) and a 1N4001 diode. The 12 VDC supply should have sufficient current capability to supply the needs of the bilge pump.



(a) Shoreline Bilge Pump



(b) MSP430 to bilge pump interface

Figure 4.36: Bilge pump interface.

4.6 AC DEVICES

A high-power alternating current (AC) load may be switched on and off using a low power control signal from the microcontroller. In this case, a Solid State Relay is used as the switching

device. Solid state relays are available to switch a high power DC or AC load [Crydom]. For example, the Crydom 558-CX240D5R is a printed circuit board-mounted, air-cooled, single-pole single-throw (SPST), normally open (NO) solid-state relay. It requires a DC control voltage of 3-15 VDC at 15 mA. This microcontroller compatible DC control signal is used to switch 12-280 VAC loads rated from 0.06–5 amps [Crydom].

To vary the direction of an AC motor, you must use a bi-directional AC motor. A bi-directional motor is equipped with three terminals: common, clockwise, and counterclockwise. To turn the motor clockwise, an AC source is applied to the common and clockwise connections. In like manner, to turn the motor counterclockwise, an AC source is applied to the common and counterclockwise connections. This may be accomplished using two of the Crydom SSRs.

PowerSwitch manufacturers an easy-to-use AC interface the PowerSwitch Tail II. The device consists of a control module with attached AC connections rated at 120 VAC, 15 A. The device to be controlled is simply plugged inline with the PowerSwitch Tail II. A digital control signal from MSP430 (3 VDC at 3 mA) serves as the on/off control signal for the controlled AC device. The controlled signal is connected to the PowerSwitch Tail II via a terminal block connection. The PowerSwitch II may be configured as either normally closed (NC) or normally open (NO) (`www.powerswitchtail.com`).

Example: PowerSwitch Tail II. In this example, we use an IR sensor to detect someone's presence. If the IR sensor's output reaches a predetermined threshold level, an AC desk lamp is illuminated, as shown in Figure 4.37.

```
//*****************************************************************
//switch_tail
//
//The circuit:
//  - The IR sensor signal pin is connected to analog pin 3 (6).
//    The sensor power and ground pins are connected to 5 VDC and
//    ground respectively.
//  - The analog output is designated as the onboard red LED.
//  - The switch tail control signal is connected to P6.2 (pin 5)
//
//Adapted for code originally written by Tom Igoe
//Created:  Dec 29, 2008
//Modified: Aug 30, 2011
//Author:   Tom Igoe
//
//This example code is in the public domain.
//*****************************************************************

const int analogInPin  = 6;       //Energia analog input pin A3
```

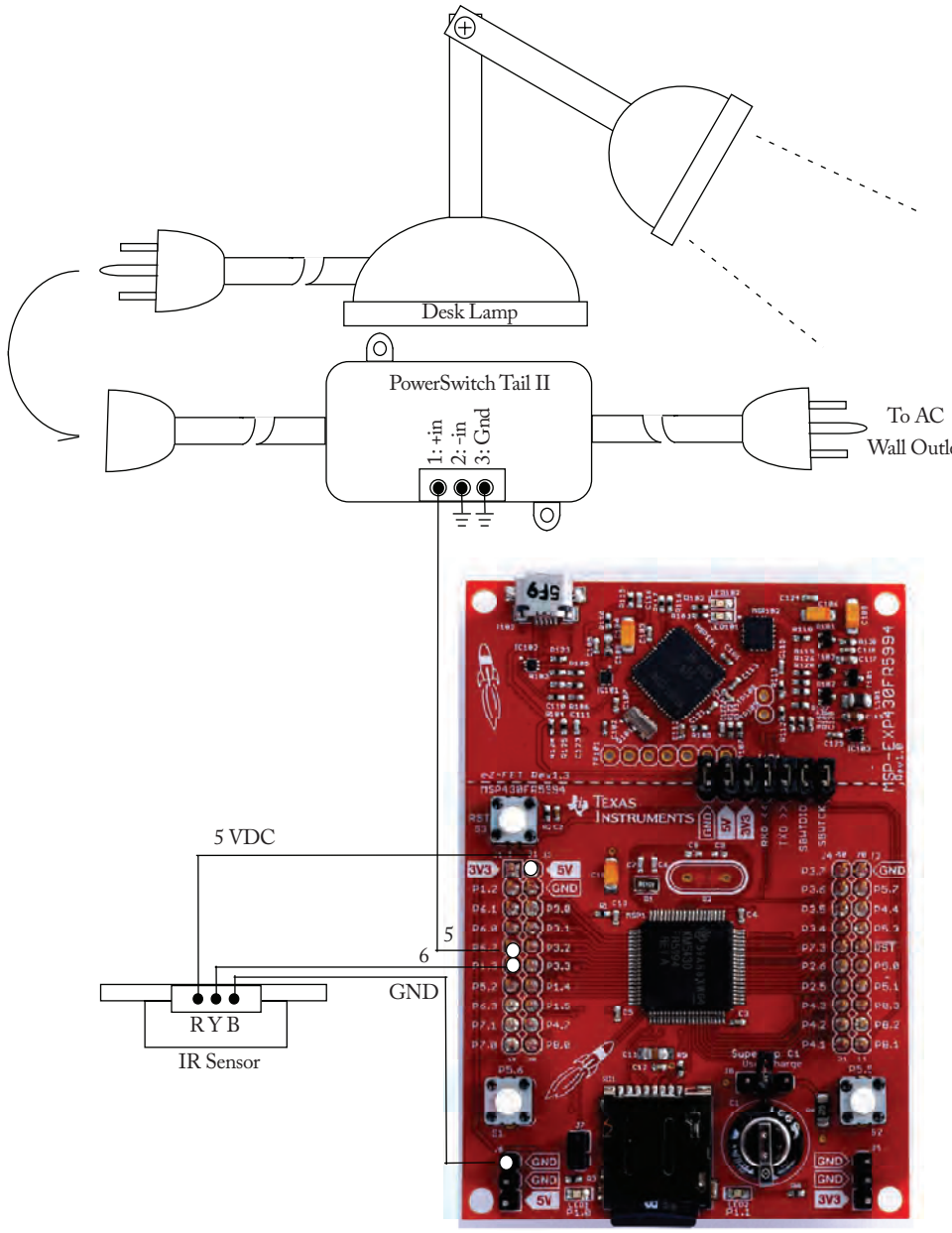


Figure 4.37: PowerSwitch Tail II. (Illustration used with permission of Texas Instruments (www.ti.com).)

```
const int analogOutPin = RED_LED; //Energia onboard red LED pin
const int switch_tail_control =5zz; //Switch Tail control signal

int sensorValue = 0;              //value read from the OR sensor
int outputValue = 0;              //value output to the PWM (red LED)

void setup()
  {
  //initialize serial communications at 9600 bps:
  Serial.begin(9600);

  //configure Switch Tail control pin
  pinMode(switch_tail_control, OUTPUT);
  }

void loop()
  {
  //read the analog in value:
  sensorValue = analogRead(analogInPin);

  //map it to the range of the analog out:
  outputValue = map(sensorValue, 0, 1023, 0, 255);

  //change the analog out value:
  analogWrite(analogOutPin, outputValue);

  //Switch Tail control signal
  if(outputValue >= 128)
    {
    digitalWrite(switch_tail_control, HIGH);
    Serial.print("Light on");
    }
  else
    {
    digitalWrite(switch_tail_control, LOW);
    Serial.print("Light off");
    }

  // print the results to the serial monitor:
```

```
  Serial.print("sensor = " );
  Serial.print(sensorValue);
  Serial.print("\t output = ");
  Serial.println(outputValue);

  // wait 10 milliseconds before the next loop
  // for the analog-to-digital converter to settle
  // after the last reading:
  delay(10);
}

//*************************************************************
```

4.7 MSP430FR5994: EDUCATIONAL BOOSTER PACK MKII

The Educational Booster Pack MkII allows rapid prototyping of designs. Shown in Figure 4.38, it is equipped with a variety of transducers and output devices. The MkII may be mounted to the MSP-EXP430FR5994 LaunchPad. Energia provides a variety of test programs for the MkII.

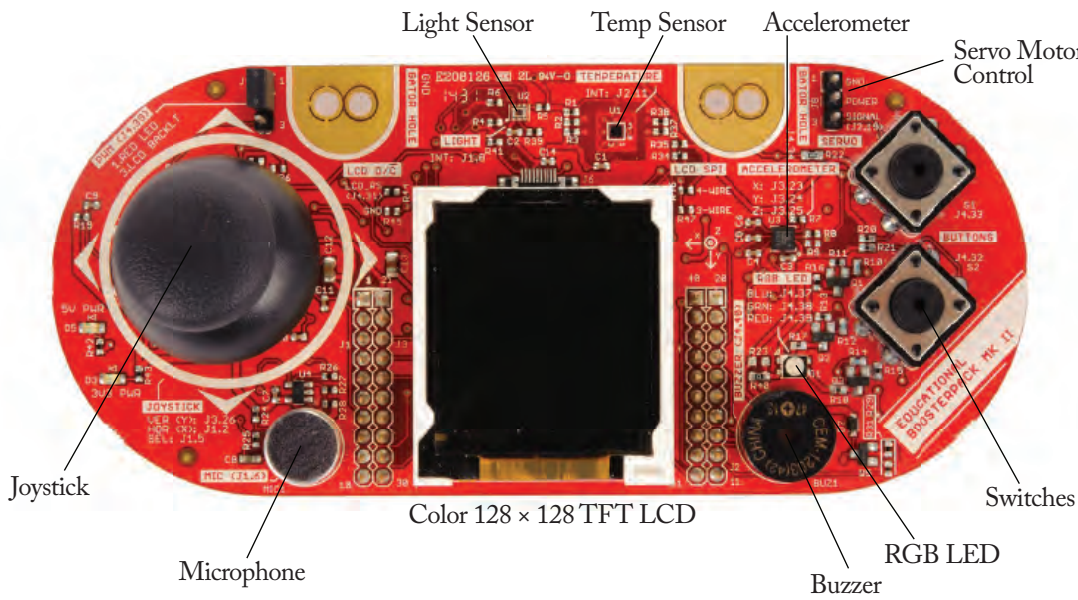


Figure 4.38: Educational Booster Pack MkII. (Illustration used with permission of Texas Instruments (www.ti.com).)

In the list of features below, pin numbers provided refer to the MkII pin placement. The MkII is equipped with a [SLAU599].

- **Two-axis joystick.** The ITEAD Studio IM130330001 is a two-axis analog joystick equipped with a pushbutton. The two analog signals are generated by x and y oriented potentiometers. As the joystick is moved the analog signals relay the joystick position to the MSP430 via the J1.2 (X) and J3.26 (Y) header pins. The joystick select pushbutton is connected to pin J1.5.

- **Microphone.** The MkII is equipped with the CUI CMA-4544PW-W electret microphone. The microphone signal is amplified via an OPA344 operational amplifier. The microphone has a frequency response of 20 Hz to 20 kHz. The microphone is connected to MSP430 pin J1.6.

- **Light sensor.** The light sensor aboard the MkII is the OPT3001 digital ambient light sensor. The sensor measures ambient light intensity and it is tuned to the light response of the human eye. It also has filters to reject infrared (IR) light. It detects light intensity in the range from 0.01–83 lux. The I2C compatible output of the sensor is provided to MSP430 via pins J1.9 (I2C SCL), J1.10 (I2C SDA), and J1.8 (sensor interrupt).

- **Temperature sensor.** The temperature sensor is also I2C compatible. The TMP006 is a noncontact sensor that passively absorbs IR wavelengths from 4–16 μm. The I2C compatible output is provided to the MSP430 via pins J1.9 (I2C SCL), J1.10 (I2C SDA), and J2.11 (sensor interrupt).

- **Servo motor controller.** The MkII is equipped with a convenient connector for a servo motor. The servo motor control signal is provided by MSP430 signal pin J2.19.

- **Three-axis accelerometer.** Aboard the MkII is a Kionix KXTC9-2050 three-axis accelerometer that measures acceleration in the X, Y, and Z directions. The three-channel analog output corresponds to acceleration from $\pm$1.5 g to $\pm$6 g. The three channels of analog output are provided to the MSP430 via pins J3.23 (X), J3.24 (Y), and J3.25 (Z).

- **Pushbuttons.** The MkII is equipped with two pushbuttons designated S1 and S2. They are connected to the MSP430 via pins J4.33 (S1) and J4.32 (S2).

- **Red-Green-Blue (RGB) LED.** The RGB LED aboard the MkII is the Cree CLV1A--FKB RGB multicolor LED. The three-color components are accessible via pins J4.39 (red), J4.38 (green), and J4.37 (blue). The intensity of each component may be adjusted using PWM techniques.

- **Buzzer.** The piezo buzzer aboard the MkII is the CUI CEM-1230. The buzzer will operate at various frequencies using PWM techniques. The buzzer is accessible via pin J4.40.

- **Color TFT LCD.** The color 2D LCD aboard the MkII is controlled via the serial peripheral interface (SPI) system. The Crystalfontz CFAF 128128B-0145T is a color 128 by 128 pixel display.

Provided in Energia is considerable software support for the MkII. This software will be explored in the Laboratory Exercise, Section 4.10. It is important to note that not all of the Educational Booster Pack MkII examples included within Energia are compatible with the MSP430FR5994.

4.8 GROVE STARTER KIT FOR LAUNCHPAD

The Seeed company provides a Grove Starter Kit for the MSP430 LaunchPad shown in Figure 4.39. It consists of a BoosterPack configured breakout board for a number of sensors and output devices including (`www.seeedstudio.com`):

- buzzer
- four-digit seven-segment LED display
- relay
- proximity infrared sensor (PIR) sensor
- ultrasonic ranger
- light sensor
- rotary angle sensor
- sound sensor
- moisture sensor
- temperature and humidity sensor

The Grove Starter Kit is enhanced by considerable software support we explore in the Laboratory Exercise, Section 4.11.

4.9 APPLICATION: SPECIAL EFFECTS LED CUBE

To illustrate some of the fundamentals of MSP430 interfacing, we construct a three-dimensional LED cube. This design was inspired by an LED cube kit available from Jameco (`www.jameco.com`). This application originally appeared in the third edition of *Arduino Microcontroller Processing for Everyone!* The LED cube example has been adapted with permission for compatibility with the MSP430 [Barrett, 2013].

The MSP430-EXP430FR5994 LaunchPad is a 3.3 VDC system. With this in mind, we take two different design approaches:

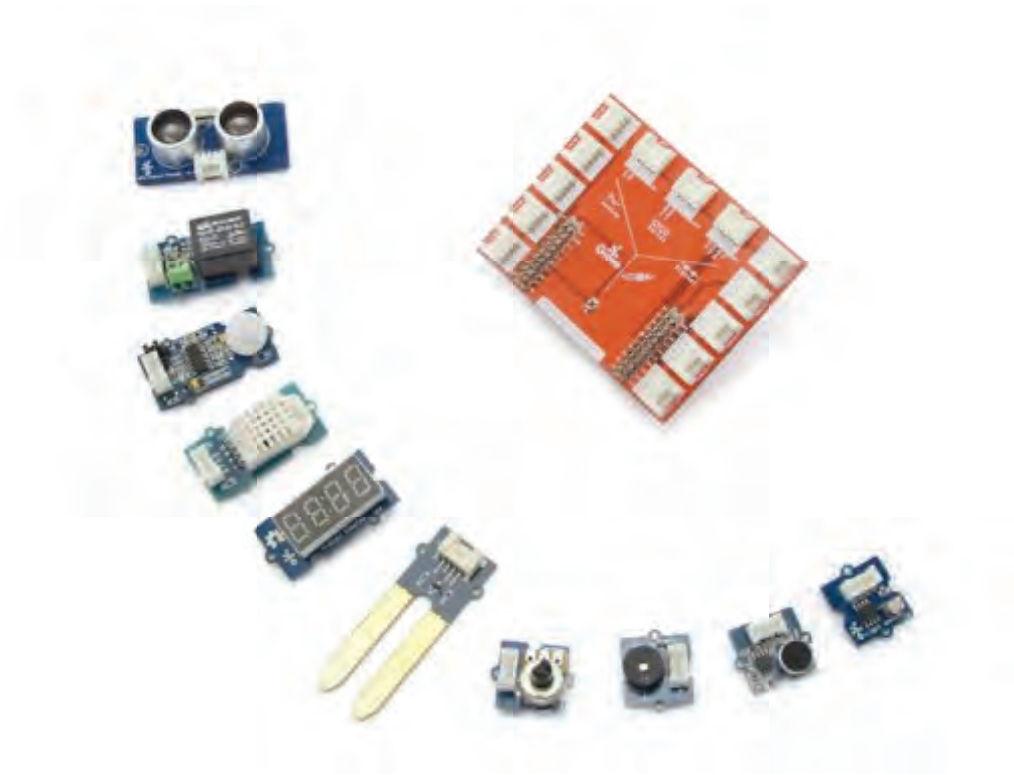


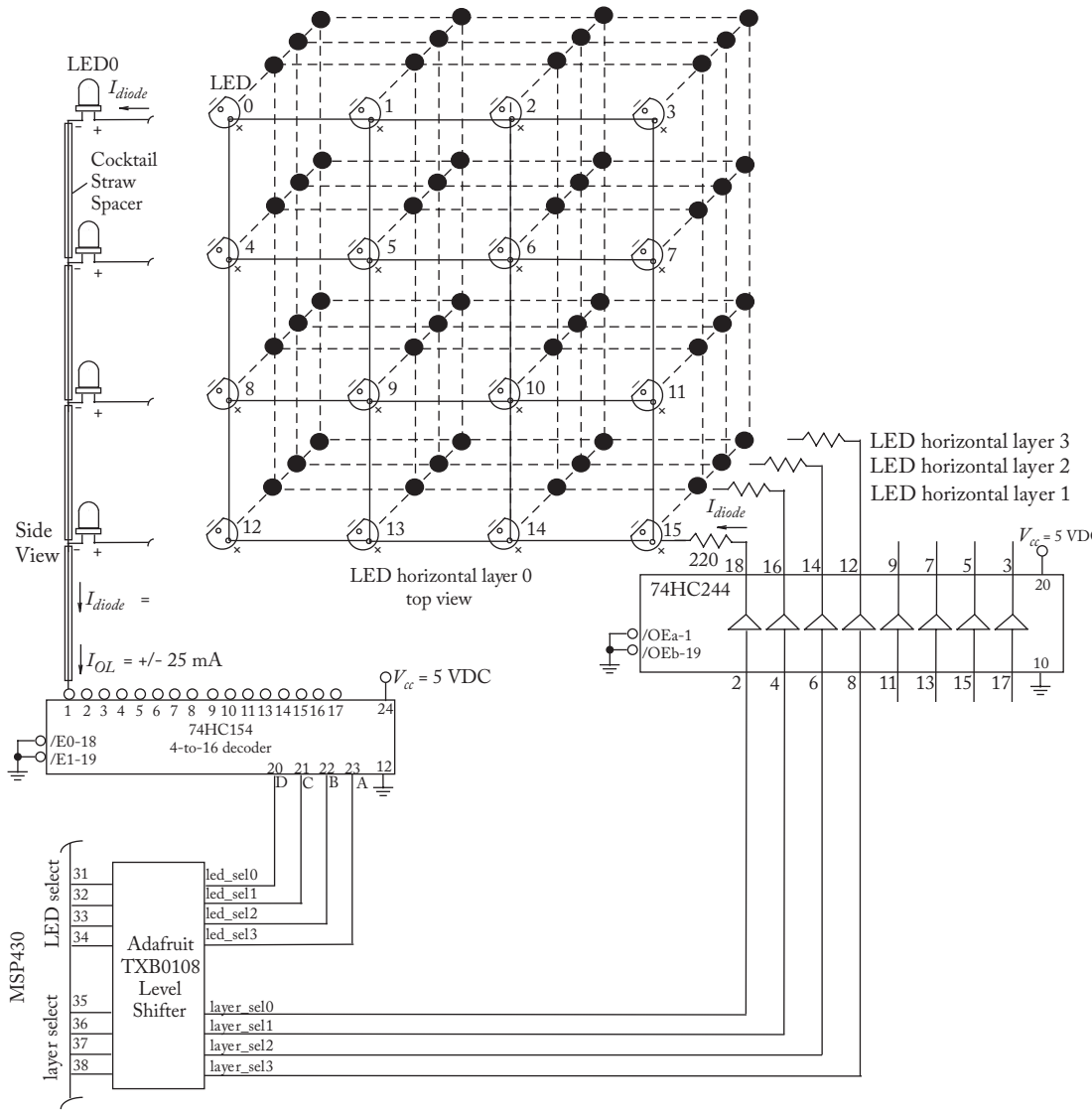
Figure 4.39: Grove starter kit for LaunchPad (www.seeedstudio.com). (Illustrations used with permission of Texas Instruments (www.TI.com).)

1. Interface the 3.3 VDC MSP430 to an LED cube designed for 5 VDC operation via a 3.3–5.0 VDC level shifter.

2. Modify the design of the LED cube to operate at 3.3 VDC.

We explore each design approach in turn.

Approach 1: 5 VDC LED cube. The LED cube consists of 4 layers of LEDs with 16 LEDs per layer. Only a single LED is illuminated at a given time. However, different effects may be achieved by how long a specific LED is left illuminated and the pattern of LED sequence followed. A specific LED layer is asserted using the layer select pins on the microcontroller using a one-hot-code (a single line asserted while the others are de-asserted). The asserted line is fed through a 74HC244 (three state, octal buffer, line driver) which provides an I_{OH}/I_{OL} current of $\pm$ 35 mA, as shown in Figure 4.40. A given output from the 74HC244 is fed to a common anode connection for all 16 LEDs in a layer. All four LEDs in a specific LED position, each



Notes:
1. LED cube consists of 4 layers of 16 LEDs each.
2. Each LED is individually addressed by asserting the appropriate cathode signal (0–15) and asserting a specific LED layer.
3. All LEDs in a given layer share a common anode connection.
4. All LEDs in a given position (0–15) share a common cathode connection.

Figure 4.40: 5 VDC LED special effects cube.

in a different layer, share a common cathode connection. That is, an LED in a specific location within a layer shares a common cathode connection with three other LEDs that share the same position in the other three layers. The common cathode connection from each LED location is fed to a specific output of the 74HC154 4-to-16 decoder. The decoder has a one-cold-code output (one output at logic low while the others are at logic high). To illuminate a specific LED, the appropriate layer select and LED select line are asserted using the layer_sel[3:0] and led_sel[3:0] lines, respectively. This basic design may be easily expanded to a larger LED cube.

To interface the 5 VDC LED cube to the 3.3 VDC MSP430, a 3.3 VDC-to-5 VDC level shifter is required for each of the control signals (layer_sel and led_sel). In this example, a TXB0108 (low voltage octal bidirectional transceiver) is employed to shift the 3.3 VDC signals of the MSP430 to 5 VDC levels. Adafruit provides a breakout board for the level shifter (#TXB0108)(`www.adafruit.com`). Alternatively, a Texas Instruments LSF0101XEVM-001, discussed earlier in the chapter, may be used for level shifting.

Approach 2: 3.3 VDC LED cube: A 3.3 VDC LED cube design is shown in Figure 4.41. The 74HC154 1-of-16 decoder has been replaced by two 3.3 VDC 74LVX138 1-of-8 decoders. The two 74LVX138 decoders form a single 1-of-16 decoder. The led_sel3 is used to select between the first decoder via enable pin /E2 or the second decoder via enable pin E3. Also, the 74HC244 has been replaced by a 3.3 VDC 74LVX244.

4.9.1 CONSTRUCTION HINTS

To limit project costs, low-cost red LEDs (Jameco #333973) are used. This LED has a forward voltage drop (V_f) of approximately 1.8 VDC and a nominal forward current (I_f) of 20 mA. The project requires a total of 64 LEDs (4 layers of 16 LEDs each). An LED template pattern was constructed from a 5″ by 5″ piece of pine wood. A 4-by-4 pattern of holes was drilled into the wood. Holes were spaced 3/4″ apart. The hole diameter was slightly smaller than the diameter of the LEDs to allow for a snug LED fit.

The LED array was constructed a layer at a time using the wood template. Each LED was tested before inclusion in the array. A 5 VDC power supply with a series 220 ohm resistor was used to insure each LED was fully operational. The LED anodes in a given LED row were then soldered together. A fine-tip soldering iron and a small bit of solder were used for each interconnect as shown in Figure 4.42. Cross wires were then used to connect the cathodes of adjacent rows. A 22 gage bare wire was used. Again, a small bit of solder was used for the interconnect points. Four separate LED layers (4-by-4 array of LEDs) were completed.

To assemble the individual layers into a cube, cocktail straw segments were used as spacers between the layers. The straw segments provided spacing between the layers and also offered improved structural stability. The anodes for a given LED position were soldered together. For example, all LEDs in position 0 for all four layers shared a common anode connection.

The completed LED cube was mounted on a perforated printed circuit board (perfboard) to provide a stable base. LED sockets for the 74LS244 and the 74HC154 were also mounted to

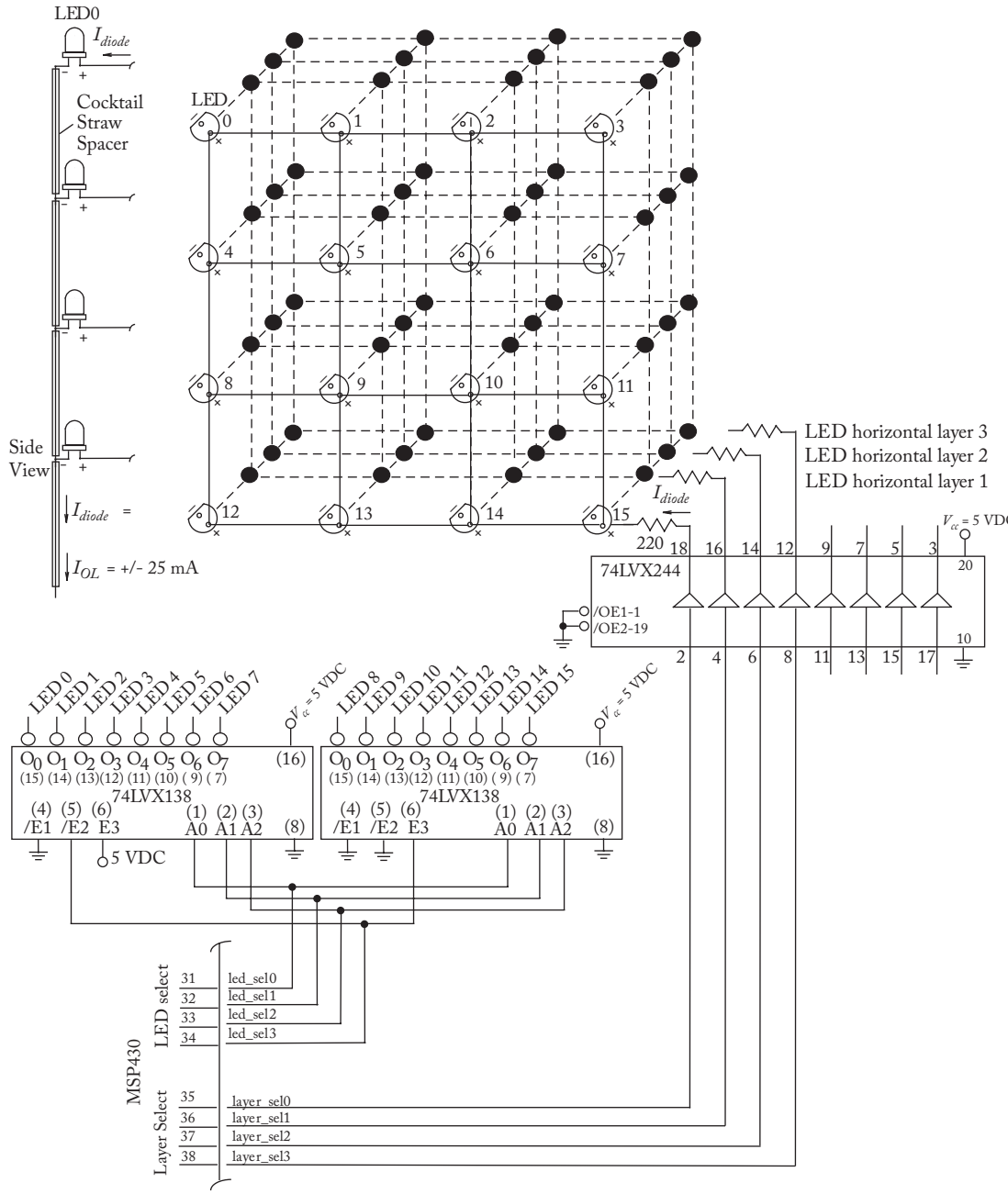
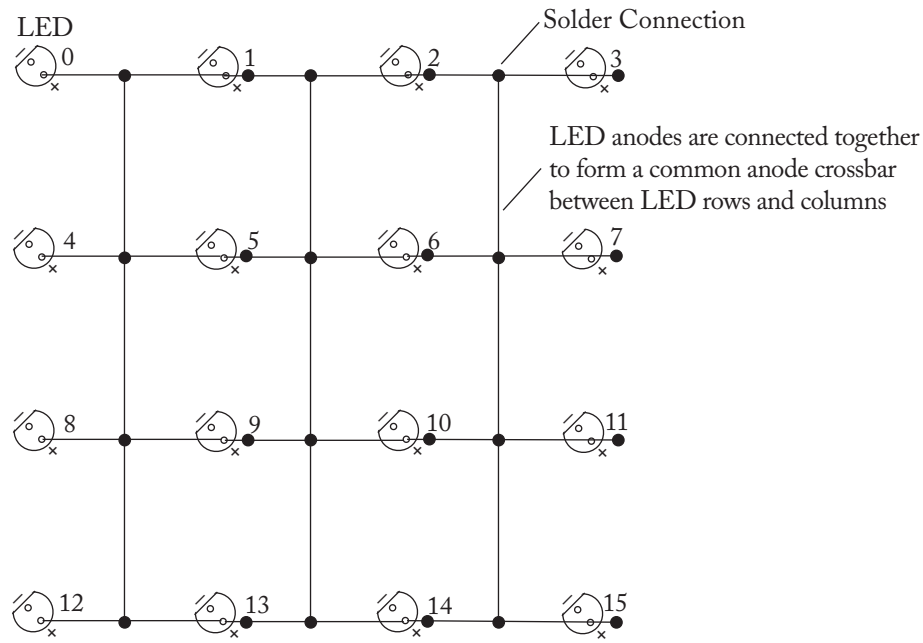
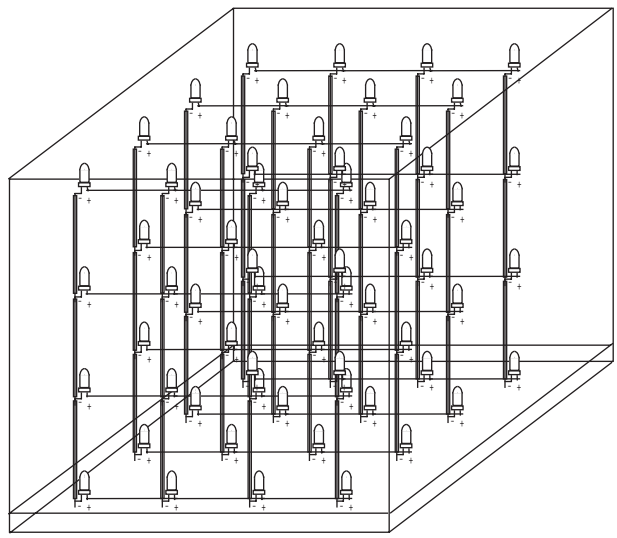


Figure 4.41: LED special effects cube.



(a) LED soldering diagram



(b) 3D LED array mounted within plexiglass cube

Figure 4.42: LED cube construction.

the perfboard. Connections were routed to a 16-pin ribbon cable connector. The other end of the ribbon cable was interfaced to the appropriate pins of the MSP430 via the level shifter. The entire LED cube was mounted within a 4″ plexiglass cube. The cube is available from the Container Store (www.containerstore.com). A construction diagram is provided in Figure 4.42. A picture of the LED cube is shown in Figure 4.43.

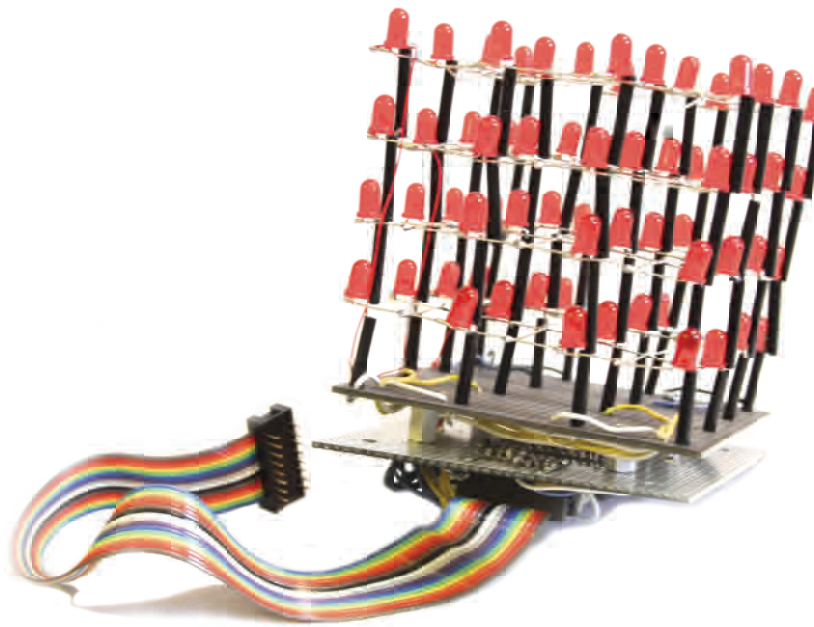


Figure 4.43: LED cube (photo courtesy of J. Barrett).

4.9.2 LED CUBE MSP430 ENERGIA CODE

Provided below is the basic code template to illuminate a single LED (LED 0, layer 0). This basic template may be used to generate a number of special effects (e.g., tornado, black hole, etc.). Pin numbers are provided for the MSP-EXP430FR5994 LaunchPad.

```
//*************************************************
//led_cube
//
//This example code is in the public domain.
//*************************************************

//led select pins
#define led_sel0  31
```

```
#define led_sel1  32
#define led_sel2  33
#define led_sel3  34

//layer select pins
#define layer_sel0  35
#define layer_sel1  36
#define layer_sel2  37
#define layer_sel3  38

void setup()
{
pinMode(led_sel0, OUTPUT);
pinMode(led_sel1, OUTPUT);
pinMode(led_sel2, OUTPUT);
pinMode(led_sel3, OUTPUT);

pinMode(layer_sel0, OUTPUT);
pinMode(layer_sel1, OUTPUT);
pinMode(layer_sel2, OUTPUT);
pinMode(layer_sel3, OUTPUT);
}

void loop()
{
                        //illuminate LED 0, layer 0
                        //led select
digitalWrite(led_sel0, LOW);
digitalWrite(led_sel1, LOW);
digitalWrite(led_sel2, LOW);
digitalWrite(led_sel3, LOW);
                        //layer select
digitalWrite(layer_sel0, HIGH);
digitalWrite(layer_sel1, LOW);
digitalWrite(layer_sel2, LOW);
digitalWrite(layer_sel3, LOW);

delay(500);             //delay specified in ms
}
```

```
//*********************************************
```

In the next example, a function "illuminate_LED" has been added. To illuminate a specific LED, the LED position (0–15), the LED layer (0–3), and the length of time to illuminate the LED in milliseconds are specified. In this short example, LED 0 is sequentially illuminated in each layer. An LED grid map is shown in Figure 4.44. It is useful for planning special effects.

```
//*********************************************
//led_cube2
//
//This example code is in the public domain.
//*********************************************

//led select pins
#define led_sel0  31
#define led_sel1  32
#define led_sel2  33
#define led_sel3  34

//layer select pins
#define layer_sel0  35
#define layer_sel1  36
#define layer_sel2  37
#define layer_sel3  38

void setup()
{
pinMode(led_sel0, OUTPUT);
pinMode(led_sel1, OUTPUT);
pinMode(led_sel2, OUTPUT);
pinMode(led_sel3, OUTPUT);

pinMode(layer_sel0, OUTPUT);
pinMode(layer_sel1, OUTPUT);
pinMode(layer_sel2, OUTPUT);
pinMode(layer_sel3, OUTPUT);
}

void loop()
{
```

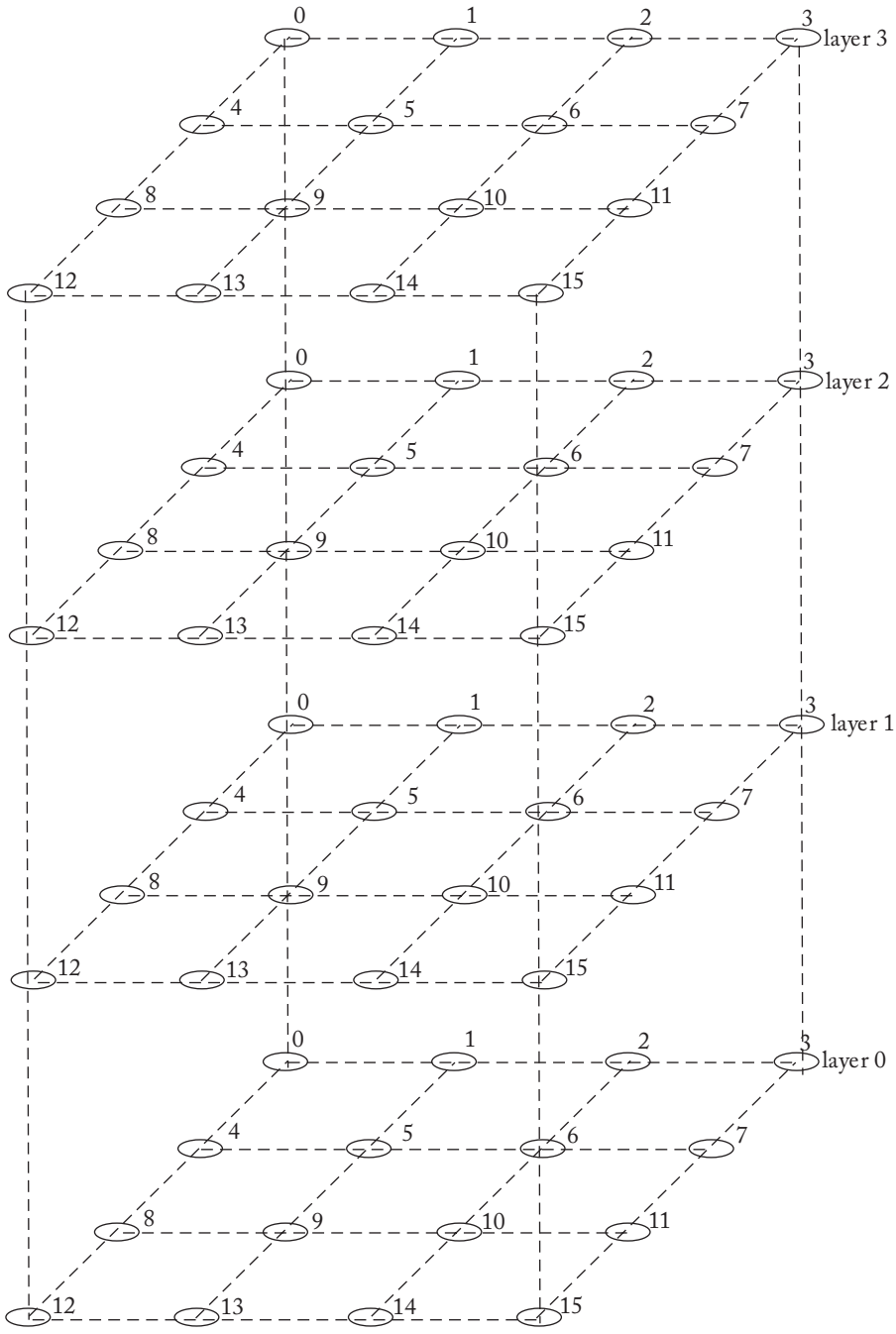


Figure 4.44: LED grid map.

```
illuminate_LED(0, 0, 500);
illuminate_LED(0, 1, 500);
illuminate_LED(0, 2, 500);
illuminate_LED(0, 3, 500);
}

//*******************************************

void illuminate_LED(int led, int layer, int delay_time)
{
if(led==0)
  {
  digitalWrite(led_sel0, LOW);
  digitalWrite(led_sel1, LOW);
  digitalWrite(led_sel2, LOW);
  digitalWrite(led_sel3, LOW);
  }
else if(led==1)
  {
  digitalWrite(led_sel0, HIGH);
  digitalWrite(led_sel1, LOW);
  digitalWrite(led_sel2, LOW);
  digitalWrite(led_sel3, LOW);
  }
else if(led==2)
  {
  digitalWrite(led_sel0, LOW);
  digitalWrite(led_sel1, HIGH);
  digitalWrite(led_sel2, LOW);
  digitalWrite(led_sel3, LOW);
  }
else if(led==3)
  {
  digitalWrite(led_sel0, HIGH);
  digitalWrite(led_sel1, HIGH);
  digitalWrite(led_sel2, LOW);
  digitalWrite(led_sel3, LOW);
  }
else if(led==4)
```

```
  {
  digitalWrite(led_sel0, LOW);
  digitalWrite(led_sel1, LOW);
  digitalWrite(led_sel2, HIGH);
  digitalWrite(led_sel3, LOW);
  }
else if(led==5)
  {
  digitalWrite(led_sel0, HIGH);
  digitalWrite(led_sel1, LOW);
  digitalWrite(led_sel2, HIGH);
  digitalWrite(led_sel3, LOW);
  }
else if(led==6)
  {
  digitalWrite(led_sel0, LOW);
  digitalWrite(led_sel1, HIGH);
  digitalWrite(led_sel2, HIGH);
  digitalWrite(led_sel3, LOW);
  }
else if(led==7)
  {
  digitalWrite(led_sel0, HIGH);
  digitalWrite(led_sel1, HIGH);
  digitalWrite(led_sel2, HIGH);
  digitalWrite(led_sel3, LOW);
  }
if(led==8)
  {
  digitalWrite(led_sel0, LOW);
  digitalWrite(led_sel1, LOW);
  digitalWrite(led_sel2, LOW);
  digitalWrite(led_sel3, HIGH);
  }
else if(led==9)
  {
  digitalWrite(led_sel0, HIGH);
  digitalWrite(led_sel1, LOW);
  digitalWrite(led_sel2, LOW);
```

```
  digitalWrite(led_sel3, HIGH);
  }
else if(led==10)
  {
  digitalWrite(led_sel0, LOW);
  digitalWrite(led_sel1, HIGH);
  digitalWrite(led_sel2, LOW);
  digitalWrite(led_sel3, HIGH);
  }
else if(led==11)
  {
  digitalWrite(led_sel0, HIGH);
  digitalWrite(led_sel1, HIGH);
  digitalWrite(led_sel2, LOW);
  digitalWrite(led_sel3, HIGH);
  }
else if(led==12)
  {
  digitalWrite(led_sel0, LOW);
  digitalWrite(led_sel1, LOW);
  digitalWrite(led_sel2, HIGH);
  digitalWrite(led_sel3, HIGH);
  }
else if(led==13)
  {
  digitalWrite(led_sel0, HIGH);
  digitalWrite(led_sel1, LOW);
  digitalWrite(led_sel2, HIGH);
  digitalWrite(led_sel3, HIGH);
  }
else if(led==14)
  {
  digitalWrite(led_sel0, LOW);
  digitalWrite(led_sel1, HIGH);
  digitalWrite(led_sel2, HIGH);
  digitalWrite(led_sel3, HIGH);
  }
else if(led==15)
  {
```

```
  digitalWrite(led_sel0, HIGH);
  digitalWrite(led_sel1, HIGH);
  digitalWrite(led_sel2, HIGH);
  digitalWrite(led_sel3, HIGH);
  }

if(layer==0)
  {
  digitalWrite(layer_sel0, HIGH);
  digitalWrite(layer_sel1, LOW);
  digitalWrite(layer_sel2, LOW);
  digitalWrite(layer_sel3, LOW);
  }
else if(layer==1)
  {
  digitalWrite(layer_sel0, LOW);
  digitalWrite(layer_sel1, HIGH);
  digitalWrite(layer_sel2, LOW);
  digitalWrite(layer_sel3, LOW);
  }
else if(layer==2)
  {
  digitalWrite(layer_sel0, LOW);
  digitalWrite(layer_sel1, LOW);
  digitalWrite(layer_sel2, HIGH);
  digitalWrite(layer_sel3, LOW);
  }
else if(layer==3)
  {
  digitalWrite(layer_sel0, LOW);
  digitalWrite(layer_sel1, LOW);
  digitalWrite(layer_sel2, LOW);
  digitalWrite(layer_sel3, HIGH);
  }

delay(delay_time);
}

//*******************************************
```

In the next example, a "fireworks" special effect is produced. The firework goes up, splits into four pieces, and then falls back down as shown in Figure 4.45. It is useful for planning special effects.

```
//************************************************
//fireworks
//
//This example code is in the public domain.
//************************************************

//led select pins
#define led_sel0  31
#define led_sel1  32
#define led_sel2  33
#define led_sel3  34

//layer select pins
#define layer_sel0  35
#define layer_sel1  36
#define layer_sel2  37
#define layer_sel3  38
//************************************************

void setup()
{
pinMode(led_sel0, OUTPUT);
pinMode(led_sel1, OUTPUT);
pinMode(led_sel2, OUTPUT);
pinMode(led_sel3, OUTPUT);

pinMode(layer_sel0, OUTPUT);
pinMode(layer_sel1, OUTPUT);
pinMode(layer_sel2, OUTPUT);
pinMode(layer_sel3, OUTPUT);
}

void loop()
{
int i;
```

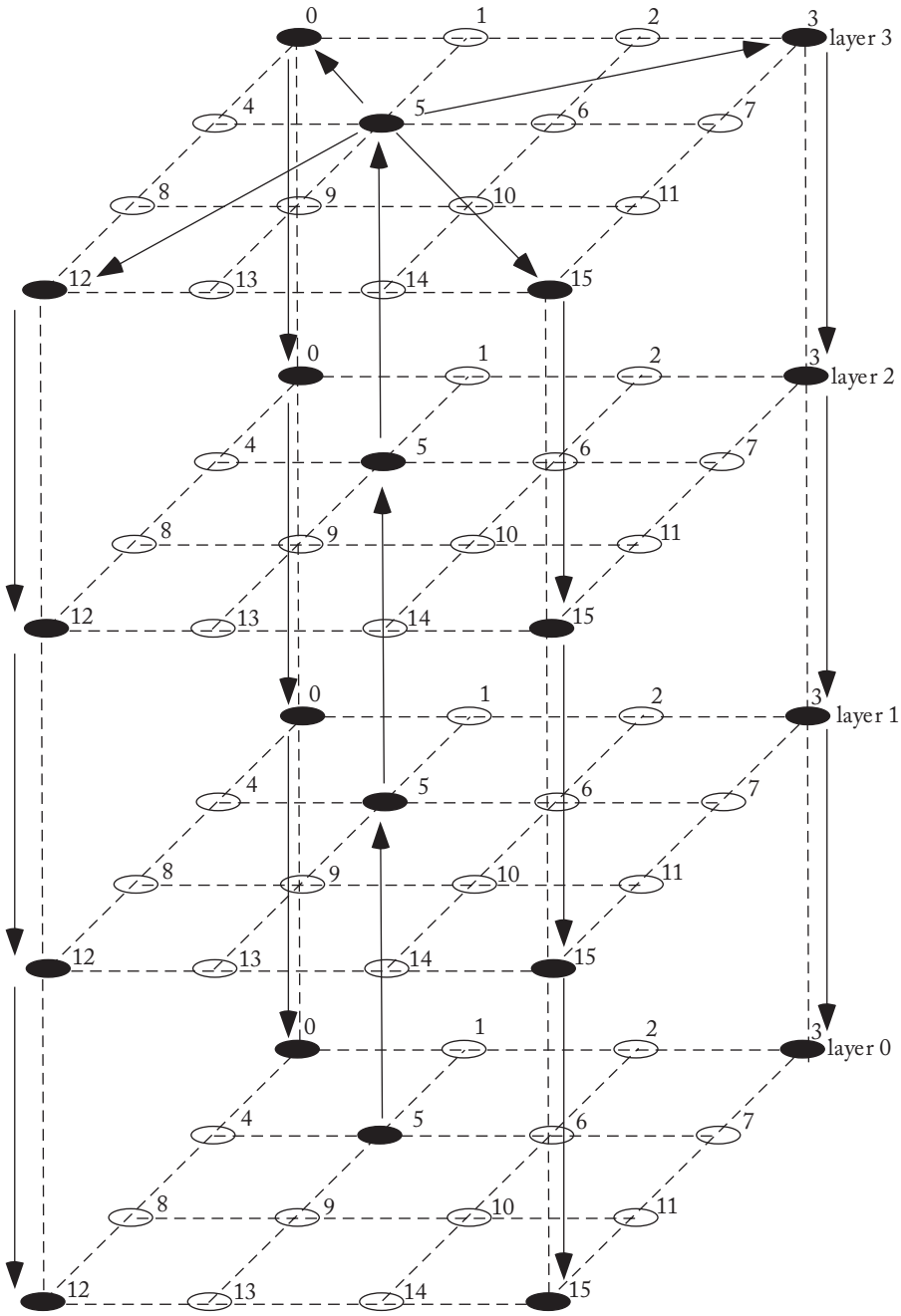


Figure 4.45: LED grid map for a fire work.

```
//firework going up
illuminate_LED(5, 0, 100);
illuminate_LED(5, 1, 100);
illuminate_LED(5, 2, 100);
illuminate_LED(5, 3, 100);

//firework exploding into four pieces
//at each cube corner
for(i=0;i<=10;i++)
  {
  illuminate_LED(0,  3, 10);
  illuminate_LED(3,  3, 10);
  illuminate_LED(12, 3, 10);
  illuminate_LED(15, 3, 10);
  delay(10);
  }

delay(200);

//firework pieces falling to layer 2
for(i=0;i<=10;i++)
  {
  illuminate_LED(0,  2, 10);
  illuminate_LED(3,  2, 10);
  illuminate_LED(12, 2, 10);
  illuminate_LED(15, 2, 10);
  delay(10);
  }

delay(200);

//firework pieces falling to layer 1
for(i=0;i<=10;i++)
  {
  illuminate_LED(0,  1, 10);
  illuminate_LED(3,  1, 10);
  illuminate_LED(12, 1, 10);
  illuminate_LED(15, 1, 10);
  delay(10);
```

```
  }

delay(200);

//firework pieces falling to layer 0
for(i=0;i<=10;i++)
  {
  illuminate_LED(0,  0, 10);
  illuminate_LED(3,  0, 10);
  illuminate_LED(12, 0, 10);
  illuminate_LED(15, 0, 10);
  delay(10);
  }

delay(10);
}

//*********************************************

void illuminate_LED(int led, int layer, int delay_time)
{
if(led==0)
  {
  digitalWrite(led_sel0, LOW);
  digitalWrite(led_sel1, LOW);
  digitalWrite(led_sel2, LOW);
  digitalWrite(led_sel3, LOW);
  }
else if(led==1)
  {
  digitalWrite(led_sel0, HIGH);
  digitalWrite(led_sel1, LOW);
  digitalWrite(led_sel2, LOW);
  digitalWrite(led_sel3, LOW);
  }
else if(led==2)
  {
  digitalWrite(led_sel0, LOW);
  digitalWrite(led_sel1, HIGH);
```

```
  digitalWrite(led_sel2, LOW);
  digitalWrite(led_sel3, LOW);
  }
else if(led==3)
  {
  digitalWrite(led_sel0, HIGH);
  digitalWrite(led_sel1, HIGH);
  digitalWrite(led_sel2, LOW);
  digitalWrite(led_sel3, LOW);
  }
else if(led==4)
  {
  digitalWrite(led_sel0, LOW);
  digitalWrite(led_sel1, LOW);
  digitalWrite(led_sel2, HIGH);
  digitalWrite(led_sel3, LOW);
  }
else if(led==5)
  {
  digitalWrite(led_sel0, HIGH);
  digitalWrite(led_sel1, LOW);
  digitalWrite(led_sel2, HIGH);
  digitalWrite(led_sel3, LOW);
  }
else if(led==6)
  {
  digitalWrite(led_sel0, LOW);
  digitalWrite(led_sel1, HIGH);
  digitalWrite(led_sel2, HIGH);
  digitalWrite(led_sel3, LOW);
  }
else if(led==7)
  {
  digitalWrite(led_sel0, HIGH);
  digitalWrite(led_sel1, HIGH);
  digitalWrite(led_sel2, HIGH);
  digitalWrite(led_sel3, LOW);
  }
if(led==8)
```

```
  {
  digitalWrite(led_sel0, LOW);
  digitalWrite(led_sel1, LOW);
  digitalWrite(led_sel2, LOW);
  digitalWrite(led_sel3, HIGH);
  }
else if(led==9)
  {
  digitalWrite(led_sel0, HIGH);
  digitalWrite(led_sel1, LOW);
  digitalWrite(led_sel2, LOW);
  digitalWrite(led_sel3, HIGH);
  }
else if(led==10)
  {
  digitalWrite(led_sel0, LOW);
  digitalWrite(led_sel1, HIGH);
  digitalWrite(led_sel2, LOW);
  digitalWrite(led_sel3, HIGH);
  }
else if(led==11)
  {
  digitalWrite(led_sel0, HIGH);
  digitalWrite(led_sel1, HIGH);
  digitalWrite(led_sel2, LOW);
  digitalWrite(led_sel3, HIGH);
  }
else if(led==12)
  {
  digitalWrite(led_sel0, LOW);
  digitalWrite(led_sel1, LOW);
  digitalWrite(led_sel2, HIGH);
  digitalWrite(led_sel3, HIGH);
  }
else if(led==13)
  {
  digitalWrite(led_sel0, HIGH);
  digitalWrite(led_sel1, LOW);
  digitalWrite(led_sel2, HIGH);
```

```
  digitalWrite(led_sel3, HIGH);
  }
else if(led==14)
  {
  digitalWrite(led_sel0, LOW);
  digitalWrite(led_sel1, HIGH);
  digitalWrite(led_sel2, HIGH);
  digitalWrite(led_sel3, HIGH);
  }
else if(led==15)
  {
  digitalWrite(led_sel0, HIGH);
  digitalWrite(led_sel1, HIGH);
  digitalWrite(led_sel2, HIGH);
  digitalWrite(led_sel3, HIGH);
  }

if(layer==0)
  {
  digitalWrite(layer_sel0, HIGH);
  digitalWrite(layer_sel1, LOW);
  digitalWrite(layer_sel2, LOW);
  digitalWrite(layer_sel3, LOW);
  }
else if(layer==1)
  {
  digitalWrite(layer_sel0, LOW);
  digitalWrite(layer_sel1, HIGH);
  digitalWrite(layer_sel2, LOW);
  digitalWrite(layer_sel3, LOW);
  }
else if(layer==2)
  {
  digitalWrite(layer_sel0, LOW);
  digitalWrite(layer_sel1, LOW);
  digitalWrite(layer_sel2, HIGH);
  digitalWrite(layer_sel3, LOW);
  }
else if(layer==3)
```

```
  {
  digitalWrite(layer_sel0, LOW);
  digitalWrite(layer_sel1, LOW);
  digitalWrite(layer_sel2, LOW);
  digitalWrite(layer_sel3, HIGH);
  }

delay(delay_time);
}

//*******************************************
```

4.10 LABORATORY EXERCISE: INTRODUCTION TO THE EDUCATIONAL BOOSTER PACK MKII AND THE GROVE STARTER KIT

Introduction. In this laboratory exercise, we get acquainted with the features of the Educational Booster Pack MkII and the Grove Starter Kit.

Procedure 1: Access the MkII support software available with Energia. Execute the software and interact with the peripherals aboard the MkII. Develop a table of features for the MkII. The table should have column headings for:

- MkII feature,
- MSP430 pins used for interface, and
- notes on interesting aspects.

Procedure 2: Access the Grove Starter Kit support software. Execute the software and interact with the peripherals aboard the Grove. Develop a table of features for the Grove. The table should have column headings for:

- grove feature,
- MSP430 pins used for interface, and
- notes on interesting aspects.

4.11 LABORATORY: COLLECTION AND DISPLAY OF WEATHER INFORMATION

Introduction and Background. In the last chapter of the book, we design and implement a multifunction weather station. In preparation for this design project, an interface and support software functions are required for a serial LCD.

Procedure: Earlier in the chapter, we discussed the interface for a serial configured LCD. Research and locate a serial LCD compatible with the MSP430 microcontroller. Design and implement the interface and support software for the LCD.

4.12 SUMMARY

In this chapter, we presented the voltage and current operating parameters for the MSP430 microcontroller. We discussed how this information may be applied to properly design an interface for common input and output circuits. It must be emphasized a carefully and properly designed interface allows the microcontroller to operate properly within its parameter envelope. If due to a poor interface design, a microcontroller is used outside its prescribed operating parameter values, spurious and incorrect logic values will result. We provided interface information for a wide range of input and output devices. We also discussed the concept of interfacing a motor to a microcontroller using PWM techniques coupled with high power MOSFET or SSR switching devices.

4.13 REFERENCES AND FURTHER READING

4N25 Optocoupler, Phototransistor Output, with Base Connection, Doc: 83725, 2010. www.vishay.com

Barrett, S. F. *Arduino Microcontroller Processing for Everyone!*, 3rd ed., Morgan & Claypool, 2013. DOI: 10.2200/s00522ed1v01y201307dcs043. 178, 205

Barrett, S. F. and Pack, D. J. *Microcontrollers Fundamentals for Engineers and Scientists*, Morgan & Claypool Publishers, 2006. DOI: 10.2200/s00025ed1v01y200605dcs001.

Crydom Corporation, San Diego, CA. www.crydom.com 200

Electrical Signals and Systems. Primis Custom Publishing, McGraw-Hill Higher Education, *Department of Electrical Engineering*, United States Air Force Academy, CO.

Faulkenberry, L. *An Introduction to Operational Amplifiers*, John Wiley & Sons, New York, 1977. 164, 166

Faulkenberry, L. *Introduction to Operational Amplifiers with Linear Integrated Circuit Applications*, 1982. 167

Images Company, Staten Island, NY, 10314. 155

Linear Technology, LTC1157 3.3 Dual Micropower High-Side/Low-Side MOSFET Driver. 180

MikroElektronika. www.mikroe.com

Milone Technologies-eTape Liquid Level Sensors. www.milonetech.com

MSP430FR2433 LaunchPad Development Kit (MSP-EXP430FR2433), (SLAU739), Texas Instruments, 2017.

MSP430FR2433 Mixed-Signal Microcontrollers, (SLAB034AD), Texas Instruments, 2017.

MSP430FR2433 Mixed-Signal Microcontroller, (SLASE59C), Texas Instruments, 2018. 141

MSP430FR5994 LaunchPad Development Kit (MSP-EXP430FR5994), (SLAU678A), Texas Instruments, 2016.

MSP430FR599x, MSP430FR596x Mixed-Signal Microcontrollers, (SLASE54C), Texas Instruments, 2018.

Power Switch Tail II. www.powerswitchtail.com

Sedra, A. and Smith, K. *Microelectronic Circuits,* 5th ed., Oxford, Oxford University Press, 2004. 170

Seeed-The IoT Hardware Enabler. www.seeed.com

Shoreline Marine. www.shorelinemarinedevelopment.com

Sick Stegmann Incorporated, Dayton, OH. www.stegmann.com 154, 155

Sparkfun. www.sparkfun.com

Texas Instruments H-Bridge Motor Controller IC, (SLVSA74A), 2010.

Texas Instruments Logic Guide, (SDYU001AB), Texas Instruments, 2017. 142, 143

Texas Instruments LSF010XEVM-001 User's Guide, (SDLU003A), Texas Instruments, 2015. 144

Texas Instruments TPIC6C596 Power Logic 8-Bit Shift Register, (SLIS093D), 2015. 145

4.14 CHAPTER PROBLEMS

Fundamental

1. What will happen if a microcontroller is used outside of its prescribed operating envelope?
2. Discuss the difference between the terms "sink" and "source" as related to current loading of a microcontroller.
3. Can an LED with a series-limiting resistor be directly driven by the MSP430 microcontroller? Explain.
4. In your own words, provide a brief description of each of the microcontroller electrical parameters.
5. What is switch bounce? Describe two techniques to minimize switch bounce.
6. Describe a method of debouncing a keypad.

Advanced

1. What is the difference between an incremental encoder and an absolute encoder? Describe applications for each type.
2. What must be the current rating of the 2N2222 and 2N2907 transistors used in the tri-state LED circuit? Support your answer.

Challenging

1. Draw the circuit for a six-character seven segment display. Fully specify all components. Write a program to display "MSP430."
2. Repeat the question above for a dot matrix display.
3. Repeat the question above for a LCD display.
4. An MSP430 has been connected to a JRP 42BYG016 unipolar, 1.8° per step, 12 VDC at 160 mA stepper motor. A 1-s delay is used between the steps to control motor speed. Pushbutton switches SW1 and SW2 are used to assert CW and CCW stepper motion. Write the code to support this application.

CHAPTER 5

Power Management and Clock Systems

Objectives: After reading this chapter, the reader should be able to:

- describe strategies to minimize power consumption in an MSP430-based microcontroller application;
- balance the power demands of a MSP430 controlled system and the power supply through available power sources;
- describe different methods to enable ultra-low power (ULP) operation;
- define different operating modes for the MSP430 microcontroller and how they contribute to ULP operation;
- illustrate the MSP430 PMM and how it contributes to ULP operation;
- configure the MSP430 supply voltage supervisor (SVS) system and describe how it contributes to ULP operation;
- explain the MSP430 CS and how it contributes to ULP operation;
- define battery capacity and its related parameters;
- describe voltage regulation and different methods of achieving regulation;
- describe Energy Trace Technology features of the MSP430 microcontroller; and
- employ a high-efficiency charge pump circuit to operate an MSP430 microcontroller from a single 1.5 VDC battery.

5.1 OVERVIEW

In a basic circuits course, we learn energy consumption is the product of power and time. We also learn that power is the product of voltage and current. This chapter discusses techniques to minimize the energy consumption in an MSP430 microcontroller-based application. The overriding chapter theme is to strategically use different techniques to minimize the time the microcontroller is operated in high voltage and current modes.

We begin the chapter with a discussion on the balancing act microcontroller-based system designers perform between the energy requirements of a given application and available power sources. We also provide an overview of the ultra-low power strategies and features of the MSP430 microcontroller that support such strategies. We then discuss the LPMs of the MSP430 and how they help to reduce power consumption. We investigate the MSP430 subsystems which contribute to ULP operation, including the Power Management Module, the Supply Voltage Supervisor, and the CS. The MSP430 CS provides for a variety of clock sources and operating frequencies for the MSP430. Since operating frequency is directly related to power consumption, the goal is to choose the lowest operating frequency and still accomplish the microcontroller's task. We then consider the battery supply. We begin with a discussion of battery capacity and its key parameters. We also describe the important concept of voltage regulation and different methods of achieving a stable voltage source within a circuit. The chapter concludes with a laboratory exercise to investigate current drain in different MSP430 operating modes and methods to operate an MSP430 using various power sources including a single 1.5 VDC battery by employing a high-efficiency charge pump integrated circuit.

5.2 BACKGROUND THEORY

The MSP430 is the lowest-power consuming microcontroller available on the market [Day, 2009]. It has been designed with a wide variety of ultra-low power features. Although it can be used in a wide variety of applications, the controller is intended for battery-operated applications where frequent battery replacement is undesirable or impractical. Application examples include battery-operated toys, portable measurement instruments, home automation products, medical instruments, metering applications, and portable smart card readers [SLVS362A, 2001].

Using its power saving capabilities, the MSP430 microcontroller is typically employed in applications where operation on a battery supply is required over an extended period. To meet this operational requirement, the power demands of the MSP430 must be balanced with the capacity of the battery source. The overall current demand of the MSP430, although the lowest in the industry, increases with [Day, 2009]:

- supply voltage level,
- CPU clock speed,
- operating temperature,
- peripheral device selection,
- I/O use, and
- memory type and size.

The MSP430 microcontroller was designed as an ultra-low power processor. Taking advantage of the ULP features is more involved than simply minimizing the effects listed above. The general approach is to minimize instantaneous current draw while maximizing the time spent in LPMs. To do this, the designer must be well acquainted with the MSP430 operating modes, the Power Management Module, and the CS of the MSP430.

As discussed in Chapter 4, the designer must also ensure that peripheral devices are properly interfaced to the microcontroller. Input peripherals must provide a logic high at the supply voltage level and a logic low at ground level. This avoids intermediate input voltages that may wreak havoc upon the microcontroller. Also, a proper interface must be designed for all output devices. Furthermore, any unused microcontroller pins should be set as outputs.

To become better acquainted with MSP430 ULP features, we review operating modes, the PMM, the SVS, and the CS in the next several sections.

5.3 OPERATING MODES

The basic premise behind the lower power operating modes is to turn off system clocks (e.g., ACLK: auxiliary clock, MCLK: master clock) that are not currently in use. Since a CMOS circuit consumes power when switching logic states, turning off clocks not in use conserves power. The operating mode of the controller is determined by the settings of four bits within the Status Register (R2): CPUOFF, OSCOFF, SCG0 (System Clock Generator 0), and SCG1 (System Clock Generator 1). By configuring these four bits, a designer can select the operating mode for the controller based on the system application and at different times within an application. The different processor operating modes are summarized in Figure 5.1. It is interesting to note that current consumption is lower, the higher the LPM number selected. In the laboratory exercise, we investigate the current requirements of several different low power operating modes.

One of the advantages of designating the low power operating mode with Status Register bits is that when an interrupt occurs, the operating mode configuration is automatically saved onto the stack. The same operating mode configuration is then retrieved and restored at the completion of the interrupt service routine, automatically switching between two power modes.

Example: In this example, the MSP-EXP430FR5994 LaunchPad is placed in LPM 4.5 until the switch on P1.3 is asserted. The current is measured in the different power modes.

We program in C using the Energia integrated development environment. To compile, upload, and execute a C program in Energia; perform the following steps.

- Connect the MSP-EXP430FR5994 LaunchPad to the Energia host computer.
- Select the board type using Tools- >Board.
- Start a new sketch using File- >New.
- Click on the small down arrow icon in the top right corner of the Energia screen and then select “New Tab.”

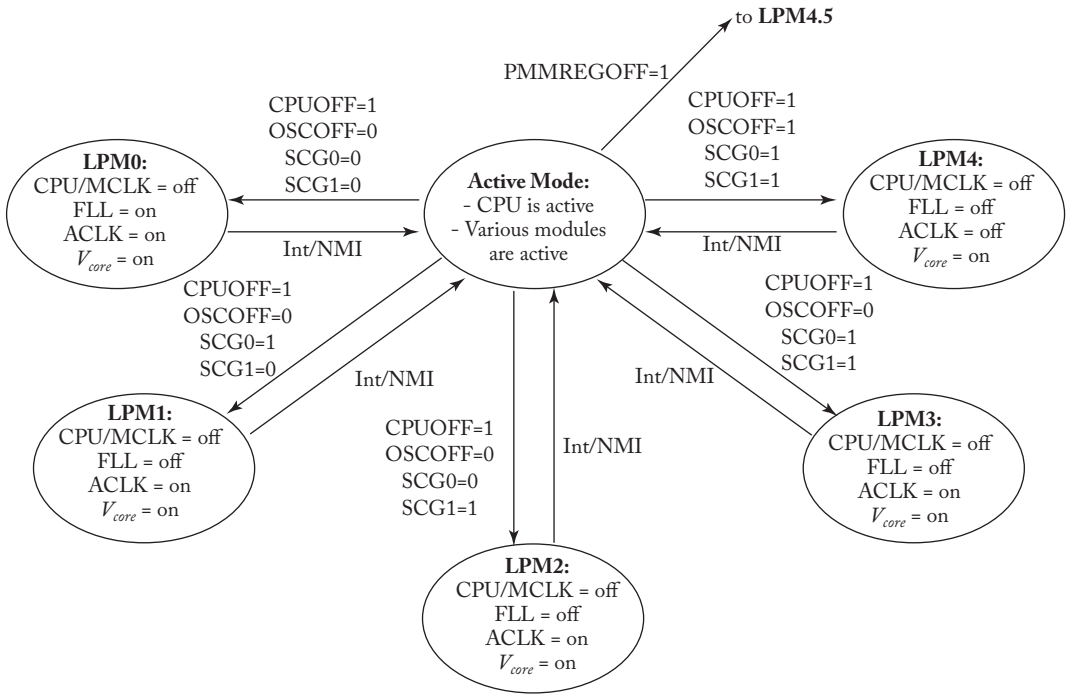


Figure 5.1: MSP430 operating modes. Adapted from Texas Instruments SLAU208Q [2018]. (Illustration used with permission of Texas Instruments (www.ti.com).)

- Insert filename.c into the white box at the bottom right corner of the Energia screen and click "OK." A new tab will now open with the designation filename.c.
- The C program may now be written in the new tab.
- Compile and load the program to the LaunchPad using the check and right arrow icons.

Note: The MSP-EXP430FR2433 LaunchPad may also be used for this example; however, the 32 kHz crystal is not connected. Procedures for connecting the crystal is provided in the *MSP430FR2433 LaunchPad Development Kit User's Guide* [SLAU739, 2017].

Use the following procedures to measure the current used by the LaunchPad [SLAU678A, 2016]:

- Remove the 3V3 jumper that links the eZ-FET Debug Probe from the MSP430FR5994 portion of the MSP-EXP430FR5994 LaunchPad, as shown in Figure 5.2.
- Connect an ammeter in place of the jumper. Insure the ammeter is capable of measuring microamps.

- Disconnect the UART channel back to the host computer by removing the TXD jumper.
- Declare all unused pins as output.
- Once the current measurement is complete, restore the 3V3 and TXD jumpers.

```
//******************************************************************
// --COPYRIGHT--,BSD_EX
// Copyright (c) 2014, Texas Instruments Incorporated
// All rights reserved.
//
//                  MSP430 CODE EXAMPLE DISCLAIMER
//
//******************************************************************
//MSP430FR5x9x Demo - Entering and waking up from LPM4.5 via P1.3
//interrupt with SVS disabled.
//
//Description:
//- Download and run the program.
//- When LPM4.5 entered, no LEDs should be on.
//- Use a multimeter to measure current on JP1 and compare to the
//  datasheet.
//- When a positive voltage is applied to P1.3 the device should
//  wake up from LPM4.5. This will enable the LFXT oscillator
//  and blink the LED (on P1.0).
//
//           MSP430FR5994
//         ---------------
//     /|\|            XIN|-
//      | |               | 32KHz Crystal
//      --|RST        XOUT|-                /|\
//        |               |                  |
//        |           P1.0|---> LED          / switch
//        |               |                  |
//        |           P1.3|<------------------
//                                  Pulled-down internally.
//                                  Apply positive voltage.
//
//William Goh
//Texas Instruments Inc.
//October 2015
```

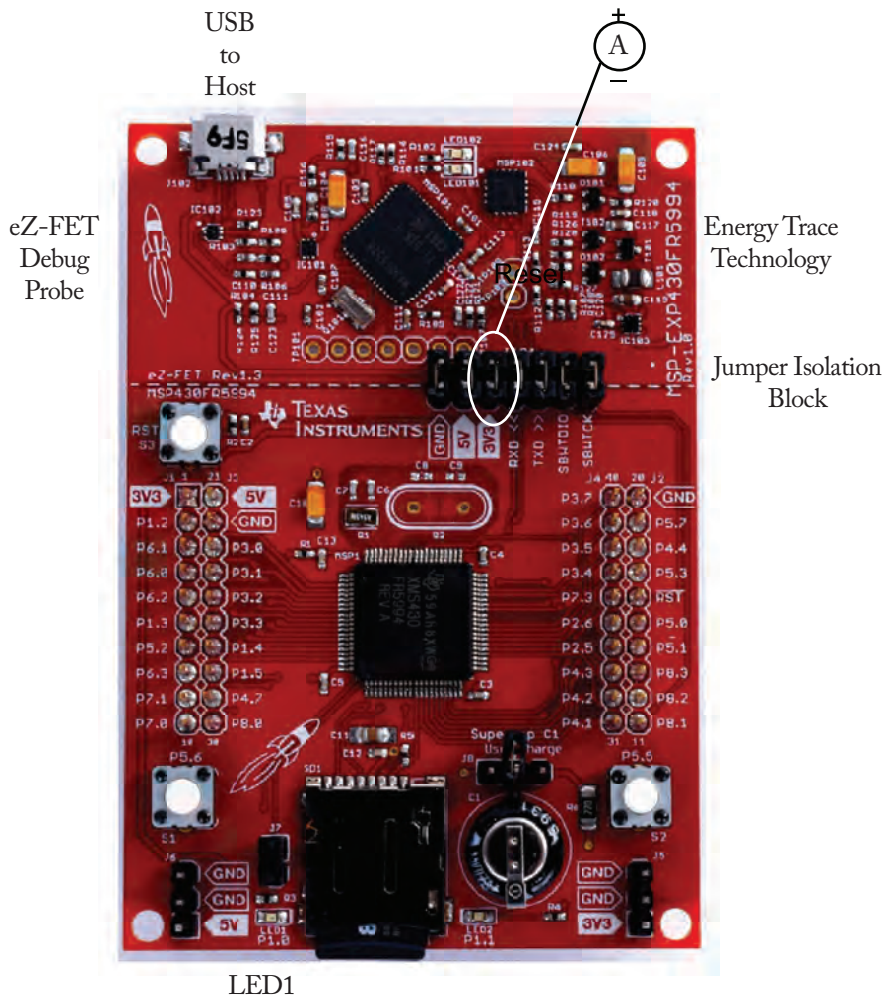


Figure 5.2: Configuring the FR5994 for current measurements. (Illustration used with permission of Texas Instruments (www.ti.com).)

```
//Built with IAR Embedded Workbench V6.30 & Code Composer Studio V6.1
//******************************************************************

#include <msp430.h>

//Function prototypes
void configure_GPIO(void);

int main(void)
{
WDTCTL = WDTPW | WDTHOLD;          //stop watchdog timer
configure_GPIO();                  //configure I/O

//Determine if MSP430 is coming out of an LPMx.5 or a regular RESET.
//SYSRSTIV used to determine source of interrupt or reset
if(SYSRSTIV == SYSRSTIV__LPM5WU)
  {
  PJSEL0 = BIT4 | BIT5;            //for XT1

  //Clock System Setup
  CSCTL0_H = CSKEY_H;              //Unlock CS registers
  CSCTL1 = DCOFSEL_0;              //Set DCO to 1MHz
  CSCTL2 = SELA__LFXTCLK | SELS__DCOCLK | SELM__DCOCLK;
  CSCTL3 = DIVA__1 | DIVS__1 | DIVM__1; //Set all dividers
  CSCTL4 &= ~LFXTOFF;

  //Configure LED pin for output
  P1DIR |= BIT0;

  //After wakeup from LPM4.5 state of I/Os are locked until the LOCKLPM5
  //bit in the PM5CTL0 register is cleared.
  //Disable the GPIO power-on default high-impedance mode to activate
  //previously configured port settings. The oscillator should now
  //start...
  PM5CTL0 &= ~LOCKLPM5;

  do{
    CSCTL5 &= ~LFXTOFFG;            //Clear XT1 fault flag
    SFRIFG1 &= ~OFIFG;
```

```
    }while (SFRIFG1 & OFIFG);       //Test oscillator fault flag
  }//end if
else
  {
  //Configure P1.3 Interrupt
  P1OUT &= ~BIT3;                   //Pull-down resistor on P1.3
  P1REN |= BIT3;                    //Select pull-down mode for P1.3
  P1DIR = 0xFF ^ BIT3;              //Set all but P1.1 to output
                                    //direction
  P1IES &= ~BIT3;                   //P1.3 Lo/Hi edge
  P1IFG = 0;                        //Clear all P1 interrupt flags
  P1IE |= BIT3;                     //P1.3 interrupt enabled

  //Disable the GPIO power-on default high-impedance mode to activate
  //previously configured port settings
  PM5CTL0 &= ~LOCKLPM5;
  PMMCTL0_H = PMMPW_H;              //Open PMM Registers for write
  PMMCTL0_L &= ~(SVSHE);            //Disable high-side SVS
  PMMCTL0_L |= PMMREGOFF;           //and set PMMREGOFF
  PMMCTL0_H = 0;                    //Lock PMM Registers

  //Enter LPM4 Note that this operation does not return. The LPM4.5
  //will exit through a RESET event, resulting in a re-start
  //of the code.
  __bis_SR_register(LPM4_bits);     //SCG1+SCG0+OSCOFF+CPUOFF

  // Should never get here...
  while (1);
  }

//Now blink the LED in an endless loop.
while(1)
  {
  P1OUT ^= BIT0;                      //P1.0 = toggle
  __delay_cycles(100000);
  }
}

//*****************************************************************
```

```
void configure_GPIO(void)
{
P1OUT = 0; P1DIR = 0xFF;  P2OUT = 0; P2DIR = 0xFF;
P3OUT = 0; P3DIR = 0xFF;  P4OUT = 0; P4DIR = 0xFF;
P5OUT = 0; P5DIR = 0xFF;  P6OUT = 0; P6DIR = 0xFF;
P7OUT = 0; P7DIR = 0xFF;  P8OUT = 0; P8DIR = 0xFF;

PJOUT = 0; PJSEL0 = BIT4 | BIT5;     // For XT1
PJDIR = 0xFFFF;
}

//******************************************************************
```

5.4 THE POWER MANAGEMENT MODULE (PMM) AND SUPPLY VOLTAGE SUPERVISOR (SVS)

The PMM is responsible for providing the main core voltage for the MSP430 microcontroller, shown in Figure 5.3. The core voltage is designated as V_{CORE} and is derived from the supply voltage provided to the microcontroller. The externally provided supply voltage is designated as the device V_{CC} or DV_{CC}. The V_{CORE} voltage, as its name implies, provides a voltage source for the core features of the microcontroller: the central processing unit (CPU), memories, and digital modules. The supply voltage DV_{CC} is primarily used to provide a voltage source for the I/O, the oscillators, and the analog modules [SLAU208Q, 2018].

As part of the PMM, the MSP430 has an onboard, integrated low-dropout voltage regulator designated as the LDO. The LDO steps the high side supply voltage DV_{CC} down to the low side core voltage V_{CORE}. The V_{CORE} voltage may be programmed in four distinct steps. The specific voltage is determined based on the anticipated system operating frequency, that is, the MCLK rate. As shown in Figure 5.4, higher system operating frequencies require higher supply voltages. The settings for the PMMCOREVx control bits for the LDO regulator (see Figure 5.3) are shown within the field intersections in Figure 5.4. Figure 5.4 is a generic figure for microcontrollers within the MSP430 family. Each family member has a specific system frequency vs. supply voltage profile [SLAU208Q, 2018].

5.4.1 SUPPLY VOLTAGE SUPERVISOR

In addition to providing the V_{CORE} voltage, the PMM has a wide variety of voltage supervisory options for both DV_{CC} and V_{CORE} via the SVS modules, shown in Figure 5.3. The SVS provides two different levels of voltage level sensing: supervision and monitoring. If supervision options are selected, the microcontroller will perform a power on reset (POR) when the sensed voltage

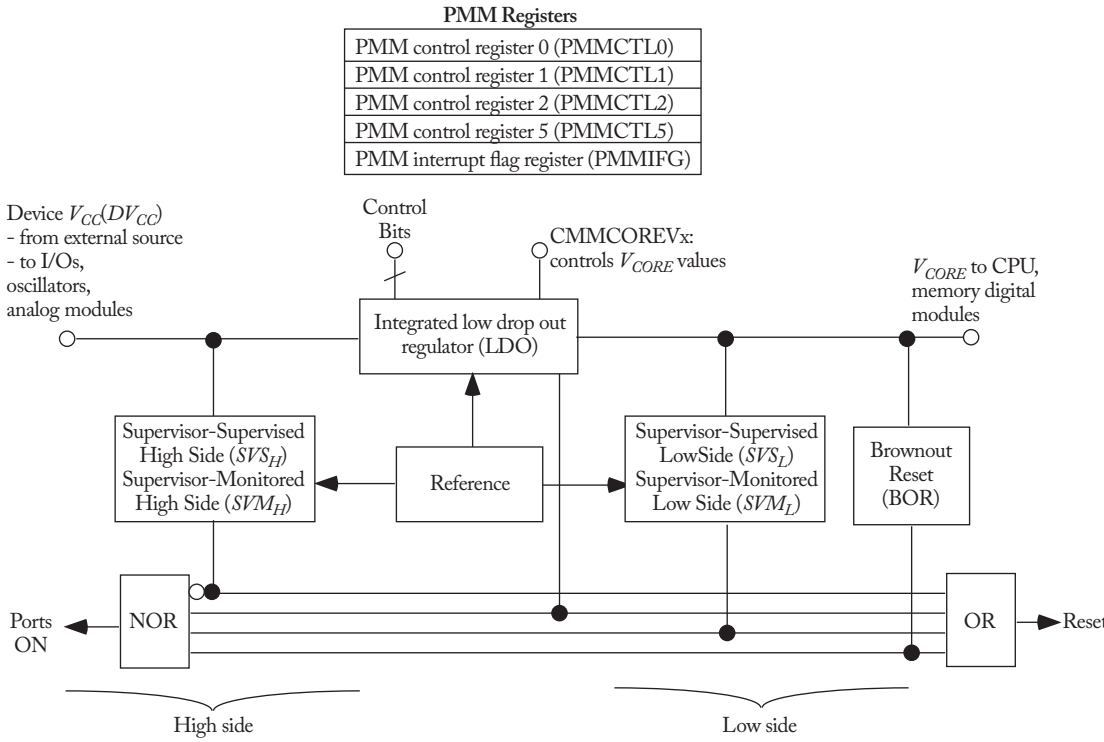


Figure 5.3: Power management module and supply voltage supervisor. (Adapted from SLAU208Q [2018].)

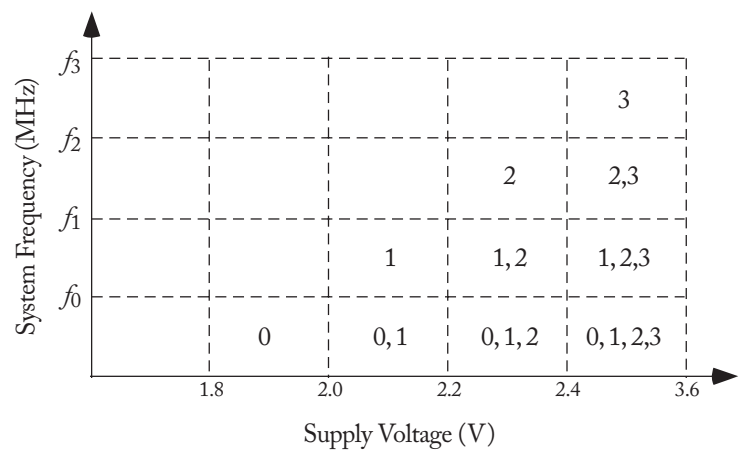


Figure 5.4: Operating frequency vs. supply voltage profile for the MSP430 microcontroller. (Adapted from SLAU208Q [2018].)

falls below a prescribed level. If monitoring options are selected, an interrupt is generated to indicate which sensed voltage has fallen below prescribed levels [SLAU208Q, 2018].

The SVS also has brownout reset (BOR) circuitry which monitors the V_{CORE} voltage. During startup, the BOR keeps the microcontroller in the reset state until voltages have reached minimum levels and stabilized. During normal operation, the BOR will generate a reset if V_{CORE} falls below prescribed levels [SLAU208Q, 2018].

5.4.2 PMM REGISTERS

All activities of the PMM and SVS are controlled by the user-defined settings of the PMM and SVS-related registers, listed in Figure 5.3. Space does not permit a detailed description of these registers. Refer to the specific documentation for a given MSP430 microcontroller for details on the register settings.

5.5 CLOCK SYSTEM

The MSP430FR2433 and the MSP430FR5994 share a similar CS, as shown in Figure 5.5. The CS allows the MSP430 to operate from many different clock sources: two external (XT1CLK and XT2CLK) and four internal (VLOCLK, REFOCLK, DCOCLK, and the MODOSC). The FR2433 is not equipped with the XT2CLK external clock.

The clock sources are routed through frequency dividers to the three main clocks for the MSP430:

- Auxiliary Clock (ACLK)
- Master Clock (MCLK)
- Subsystem Master Clock (SMCLK)

The frequency dividers allow the clock sources to be divided down by a factor of 1, 2, 4, 8, 16, or 32. This allows the frequency of a specific clock to be set optimally for a given application. A variety of clock sources provides flexibility for use in a specific application. They are the following.

- XT1CLK, or external clock 1, provides a connection for an external clock source. The external clock source is typically a low-frequency 32,768 Hz source from a crystal or a resonator.[1] This time base may be processed and conveniently used to provide a 1-s time base for RTC applications. In the default configuration for the MSP-EXP430FR2433 LaunchPad, the timebase is populated but not connected.
- VLOCLK is an internal, low-frequency, low-power consumption oscillator with a frequency typically of 10 kHz.

[1] A crystal time base is typically more stable than a resonator type time base.

Clock System Control Registers
Clock System Control 0 Register (CSCTL0)
Clock System Control 1 Register (CSCTL1)
Clock System Control 2 Register (CSCTL2)
Clock System Control 3 Register (CSCTL3)
Clock System Control 4 Register (CSCTL4)
Clock System Control 5 Register (CSCTL5)
Clock System Control 6 Register (CSCTL6)
Clock System Control 7 Register (CSCTL7)
Clock System Control 8 Register (CSCTL8)

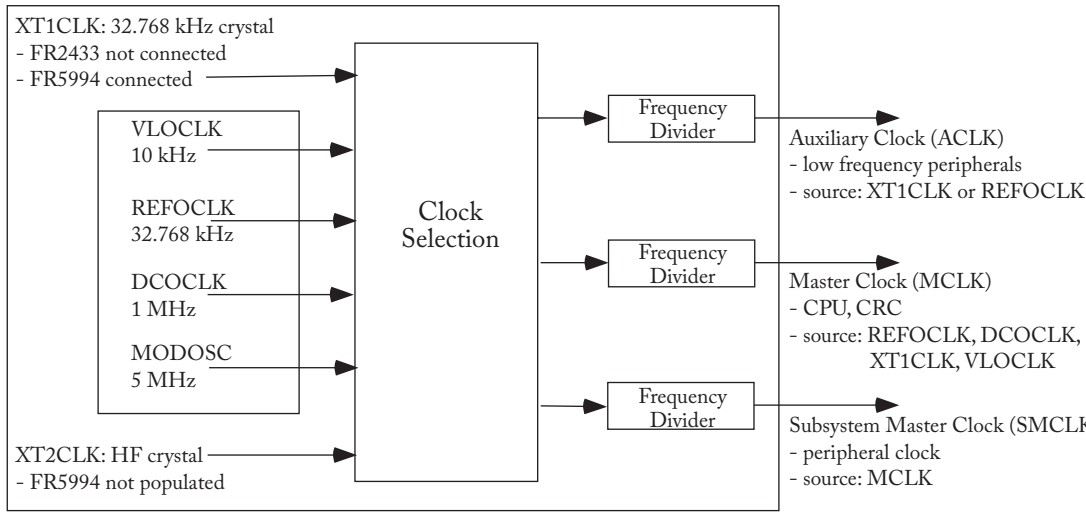


Figure 5.5: Clock system (CS) overview.

- REFOCLK is an internal low-frequency reference oscillator with a frequency of 32768 Hz.
- DCOCLK is an internal digitally controlled oscillator set for 1 MHz.
- MODOSC is an internal digitally controlled oscillator set for 5 MHz.
- XT2CLK provides a connection for an external, high-frequency clock source. This source is not available in the MSP430FR2433. In the default configuration for the MSP-EXP430FR5994 LaunchPad, the timebase is not populated on the printed circuit board.

The operation and configuration of the clock system is controlled by the CS control registers and their specific control bit settings. The detailed CS block diagrams for the MSP430FR2433 and the MSP430FR5994 are shown in Figures 5.6 and 5.7. Both diagrams provide clock sources on the left, clock divider features in the center, and clock outputs on

the right. We examine the flexibility of the CS with several examples [SLAU445G, 2016, SLAU367O, 2017].

Example: In this example, the MSP-EXP430FR2433 LaunchPad is used to configure the MCLK for 16 MHz operation. The MCLK, SMCLK, and the ACLK are routed to external pins for observation on an oscilloscope or logic analyzer.

```
//******************************************************************
// --COPYRIGHT--,BSD_EX
// Copyright (c) 2014, Texas Instruments Incorporated
// All rights reserved.
//
//                  MSP430 CODE EXAMPLE DISCLAIMER
//
//******************************************************************
//MSP430FR243x Demo - Configure MCLK for 16MHz operation,
//                    and REFO sourcing FLLREF and ACLK.
//
//Description:
//- Configure MCLK for 16MHz. FLL reference clock is REFO.  At this
//  speed, the FRAM requires wait states.
//- ACLK = default REFO ~32768Hz, SMCLK = MCLK = 16MHz.
//- Toggle LED to indicate that the program is running.
//
//           MSP430FR2433
//         ---------------
//     /|\|               |
//      | |               |
//      --|RST            |
//        |          P1.0 |---> LED
//        |          P1.3 |---> MCLK  = 16MHz
//        |          P1.7 |---> SMCLK = 16MHz
//        |          P2.2 |---> ACLK  = 32768Hz
//
//
//Ling Zhu, Texas Instruments Inc., Feb 2015
//Built with IAR Embedded Workbench v6.20 & Code Composer Studio v6.0.1
//******************************************************************

#include <msp430.h>
```

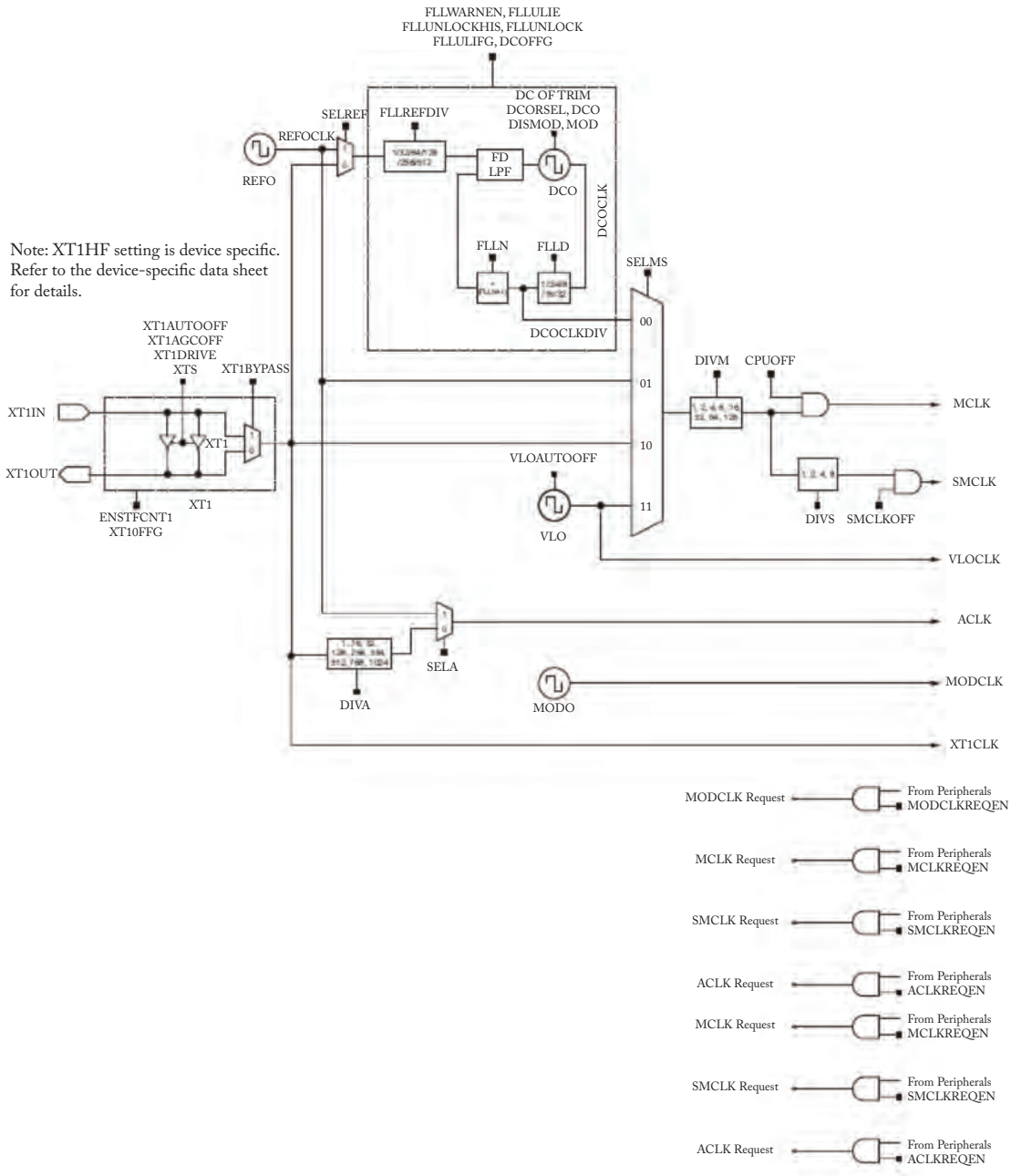


Figure 5.6: MSP430FR2433 clock system (CS) overview [SLAU445G, 2016]. (Illustration used with permission of Texas Instruments (www.ti.com).)

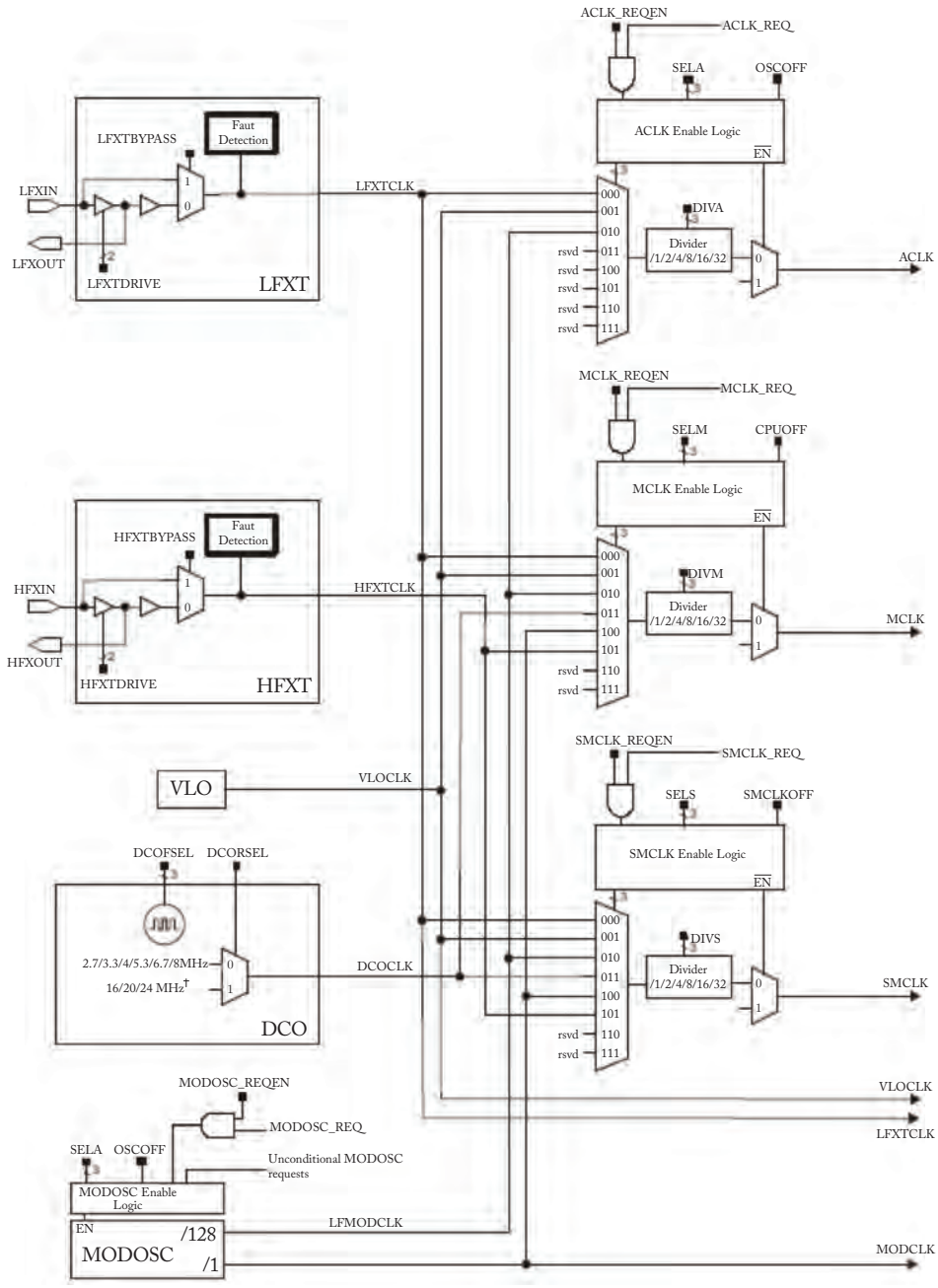


Figure 5.7: MSP430FR5994 clock system (CS) overview [SLAU367O, 2017]. (Illustration used with permission of Texas Instruments (www.ti.com).)

```
int main(void)
{
WDTCTL = WDTPW | WDTHOLD;          //Stop watchdog timer

//Configure one FRAM wait state as required by the device datasheet
//for MCLK operation beyond 8MHz before configuring the clock system.
FRCTL0 = FRCTLPW | NWAITS_1;

__bis_SR_register(SCG0);           //disable FLL
CSCTL3 |= SELREF__REFOCLK;         //Set REFO as FLL reference
CSCTL0 = 0;                        //clear DCO and MOD registers
CSCTL1 &= ~(DCORSEL_7);            //clear DCO freq select bits first
CSCTL1 |= DCORSEL_5;               //Set DCO = 16MHz
CSCTL2 = FLLD_0 + 487;             //DCOCLKDIV = 16MHz
__delay_cycles(3);
__bic_SR_register(SCG0);           //enable FLL
while(CSCTL7 & (FLLUNLOCK0 | FLLUNLOCK1)); //FLL locked
CSCTL4 = SELMS__DCOCLKDIV | SELA__REFOCLK;

//set default REFO(~32768Hz) as ACLK source, ACLK = 32768Hz
//default DCOCLKDIV as MCLK and SMCLK source

P1DIR |= BIT0 | BIT3 | BIT7;       //set MCLK SMCLK and LED pin as output
P1SEL1 |= BIT3 | BIT7;             //set MCLK, SMCLK pin as second func
P2DIR |= BIT2;                     //set ACLK pin as output
P2SEL1 |= BIT2;                    //set ACLK pin as second function
PM5CTL0 &= ~LOCKLPM5;              //disable the GPIO power-on default
                                   //high-impedance mode
                                   //to activate previously configured
                                   //port settings
while(1)
  {
  P1OUT ^= BIT0;                   //Toggle P1.0 using exclusive-OR
  __delay_cycles(8000000);         //delay for 8000000*(1/MCLK)=0.5s
  }
}

//*******************************************************************
```

Example: In this example the MSP-EXP430FR5994 LaunchPad is used to configure the MCLK and SMCLK for 8 MHz operation. The ACLK is sourced by the VLOCLK and divided for a 9.4 kHZ output. The clock signals are routed to external pins for observation on an oscilloscope or logic analyzer.

```
//******************************************************************************
// --COPYRIGHT--,BSD_EX
// Copyright (c) 2014, Texas Instruments Incorporated
// All rights reserved.
//
//                  MSP430 CODE EXAMPLE DISCLAIMER
//
//******************************************************************************
//MSP430FR5x9x Demo - Configure MCLK for 8MHz operation
//
//Description: Configure SMCLK = MCLK = 8MHz, ACLK = VLOCLK.
//
//           MSP430FR5994
//         ---------------
//     /|\|               |
//      | |               |
//      --|RST            |
//        |               |
//        |           P1.0|---> LED
//        |           P2.0|---> ACLK = ~9.4kHz
//        |           P3.4|---> SMCLK = MCLK = 8MHz
//
//William Goh, Texas Instruments Inc., October 2015
//Built with IAR Embedded Workbench V6.30 & Code Composer Studio V6.1
//******************************************************************************

#include <msp430.h>

int main(void)
{
WDTCTL = WDTPW | WDTHOLD;          //Stop WDT
                                   //Configure GPIO
P1OUT &= ~BIT0;

//Clear P1.0 output latch for a defined power-on state
```

```
P1DIR |= BIT0;                      //Set P1.0 to output direction

P2DIR |= BIT0;
P2SEL0 |= BIT0;                     //Output ACLK
P2SEL1 |= BIT0;

P3DIR |= BIT4;
P3SEL1 |= BIT4;                     //Output SMCLK
P3SEL0 |= BIT4;

//Disable the GPIO power-on default high-impedance mode to activate
//previously configured port settings
PM5CTL0 &= ~LOCKLPM5;

//Clock System Setup
CSCTL0_H = CSKEY_H;                 //Unlock CS registers
CSCTL1 = DCOFSEL_0;                 //Set DCO to 1MHz

//Set SMCLK = MCLK = DCO, ACLK = VLOCLK
CSCTL2 = SELA__VLOCLK | SELS__DCOCLK | SELM__DCOCLK;

//Per Device Errata set divider to 4 before changing frequency to
//prevent out of spec operation from overshoot transient
CSCTL3=DIVA__4 | DIVS__4 | DIVM__4;//Set all corresponding clk sources
                                    //to divide by 4 for errata
CSCTL1 = DCOFSEL_6;                 //Set DCO to 8MHz

//Delay by ~10us to let DCO settle.
//60 cycles = 20 cycles buffer + (10us / (1/4MHz))
__delay_cycles(60);
CSCTL3=DIVA__1 | DIVS__1 | DIVM__1;//Set all dividers to 1 for 8MHz
                                    //operation
CSCTL0_H = 0;                       //Lock CS Registers

while(1)
  {
  P1OUT ^= BIT0;                    //Toggle LED
  __delay_cycles(8000000);          //Wait 8,000,000 CPU Cycles
  }
```

```
}
//*********************************************************************
```

5.6 BATTERY OPERATION

Many embedded applications involve remote, portable systems, operating from a battery supply. To properly design a battery source for an embedded system, the operating characteristics of the embedded system must be matched to the characteristics of the battery supply. To properly match the battery supply to the embedded system, the following questions must be addressed.

- What are the voltage and current required by the embedded system?
- How long must the embedded system operate before battery replacement or recharge?
- Will the embedded system be powered from primary, non-rechargeable batteries or secondary, rechargeable batteries?
- Are there weight or size limitations to be considered in selecting a battery?

Once these questions have been answered, a battery may be chosen for a specific application. To choose an appropriate battery, the following items must be specified:

- battery voltage,
- battery capacity,
- battery size and weight, and
- primary or secondary battery.

Battery capacity is typically specified as a mAH rating. The capacity is the product of the current drain and the battery operational life at that current level. It provides an approximate estimate of how long a battery will last under a given current drain. The capacity is reduced at higher discharge rates. It is important to note that a battery's voltage declines as the battery discharges. Provided in Figure 5.8 are approximate capacity ratings for common battery sizes and technologies (`www.duracell.com`).

Primary and secondary batteries are manufactured using a wide variety of processes. In general, primary (non-rechargeable) batteries have a higher capacity than their secondary (rechargeable) counterparts. Also, batteries with higher capacity are more expensive than those using a lower-capacity technology. A thorough review of the manufacturers' literature is recommended to select a battery for a specific application.

	Primary (non-rechargeable) Alkaline Battery	Secondary (rechargeable) Nickel-Cadmium (Ni-Cad)	Other Battery Technologies
Battery Designator	Rated Voltage Capacity	Rated Voltage Capacity	Rated Voltage Capacity
D	1.5 V 15,000 mAh	1.2 V 1,200 mAh	---
C	1.5 V 7,000 mAh	1.2 V 1,200 mAh	---
AA	1.5 V 2,250 mAh	1.2 V 500 mAh	---
AAA	1.5 V 1,000 mAh	1.2 V 180 mAh	---
3.7 V (18650)	---	---	3.7 V 6,000 mAh (Li-ion)
9 V	9.0 V 550 mAh	---	9.0 V 250 mAh (NI-MH)
12 V	---	---	12.0 V 8.5 Ah (sealed Lead Acid)

Figure 5.8: Approximate battery capacities.

5.7 VOLTAGE REGULATION

It is essential to provide a stable supply voltage to the LaunchPad. As discussed earlier in the chapter, a voltage regulator is required to stabilize an input voltage source. Since the MSP430 is typically used in a remote application, a battery source coupled with a regulator is used to provide the stable input voltage. Figure 5.9 provides a sample circuit to provide a +3.3 VDC source. The LM1117-3.3 is a 3.3 VDC, 800 mA low dropout regulator. The maximum input voltage to the regulator is 7 VDC [SNOS412N, 2016].

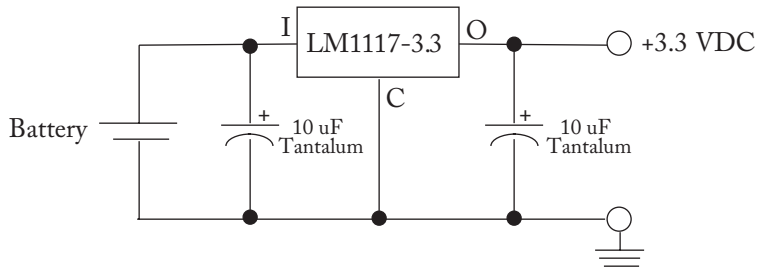


Figure 5.9: Battery supply circuits employing a 3.3 VDC regulators [SNOS412N, 2016].

5.8 HIGH-EFFICIENCY CHARGE PUMP CIRCUITS

An alternative to using a battery and regulator to power a portable application is to employ a battery and a high-efficiency charge pump circuit. For example, the Texas Instruments TPS60310 charge pump requires a 0.9–1.8 VDC input (e.g., AAA, AA battery) to produce a 3.3 VDC output suitable for powering the MSP430 microcontroller. The TPS60310 provides a maxi-

mum output current of 20 mA. In addition to the TPS60310, five additional capacitors are required, as shown in Figure 5.10 [SLVS362A, 2001].

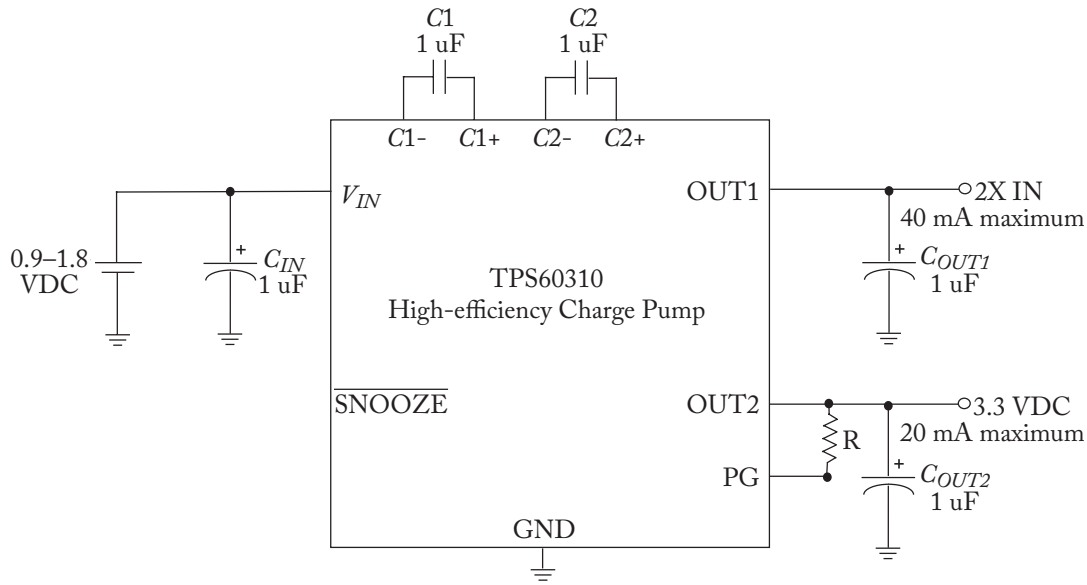


Figure 5.10: High-efficiency charge pump circuit [SLVS362A, 2001].

5.9 LABORATORY EXERCISE: MSP430 POWER SYSTEMS AND LOW-POWER MODE OPERATION

Throughout this chapter, we have investigated concepts related to providing a power supply to the MSP430 microcontroller and how to best manage this power source for extended operation. In this laboratory exercise, we investigate related aspects. The laboratory is divided into three sections. In Section 5.9.1, the current requirement of the MSP430FR5994 is investigated for several different modes of operation using the MSP-EXP430FR5994 LaunchPad. In Sections 5.9.2 and 5.9.3, a battery supply is developed for the MSP430 microcontroller, and its characteristics are studied in detail. Specifically, each supply is characterized for use in a remote, battery-operated application.

5.9.1 CURRENT MEASUREMENTS IN DIFFERENT OPERATING MODES

In this portion of the laboratory exercise, we investigate the current consumption of the MSP430FR5994 using the MSP-EXP430FR5994 LaunchPad under various operating conditions. Earlier in the chapter we provided code examples to place the MSP430 in different

LPMs and to change the processor operating frequency. Adapt these examples to measure the current requirement of the MSP430 operating under a variety of LPMs and frequencies. Develop a summary chart of your findings.

5.9.2 OPERATING AN MSP430 FROM A SINGLE REGULATED BATTERY SOURCE

Earlier in the chapter, we discussed the design of a regulated power supply from a battery source. Design and construct a regulated battery source. Based on measurements taken in the first section of the laboratory, predict the operational life of the MSP430FR5994, when operated from this supply, when operated in the active mode at the maximum operating frequency. Connect a resistor to the power supply to simulate the load of the MSP430FR5994 when operating under these conditions. Plot the voltage degradation as a function of time.

5.9.3 OPERATING AN MSP430 FROM A SINGLE 1.5 VDC BATTERY

Recall the design of a power supply using a single battery and a high-efficiency charge pump. Design and construct a source of this type. Based on measurements taken in the first section of the laboratory, predict the operational life of the MSP430FR5994 using this supply when operated in the active mode at the maximum operating frequency. Connect a resistor to the power supply to simulate the load of the MSP430FR5994 when operating under these conditions. Plot the voltage degradation as a function of time.

5.10 SUMMARY

In this chapter, we presented the ultra-low power features of the MSP430 microcontroller. We then reviewed the low-power operating modes to reduce power consumption. We then investigated the MSP430 subsystems which contribute to ULP operation, including the power management system, the supply voltage supervisor, and the CS. By evaluating the required battery capacity and the operating modes of the MSP430 controllers, one can choose appropriate batteries to satisfy system requirements.

We then examined the other side of the coin, the battery supply. We began with a discussion of battery capacity and its key parameters. We also described the important concept of voltage regulation and different methods of achieving a stable input voltage for the MSP430. The chapter concluded with a laboratory exercise to investigate current drain in different MSP430 operating modes and how to operate an MSP430 using a single 1.5 VDC battery, by employing a high-efficiency charge pump integrated circuit.

5.11 REFERENCES AND FURTHER READING

Day, M. Using power solutions to extend battery life in MSP430 applications. *Analog Applications Journal*, Texas Instruments Incorporated, 10–12, Fourth Quarter 2009. 232

LM1117 800-mA Low-Dropout Linear Regulator, (SNOS412N), Texas Instruments, 2016. 250

MSP430FR2433 LaunchPad Development Kit (MSP-EXP430FR2433), (SLAU739), Texas Instruments, 2017. 234

MSP430FR2433 Mixed-Signal Microcontrollers, (SLAB034AD), Texas Instruments, 2017.

MSP430FR2433 Mixed-Signal Microcontroller, (SLASE59C), Texas Instruments, 2018.

MSP430FR4xx and MSP430FR2xx Family User's Guide, (SLAU445G), Texas Instruments, 2016. 243, 244

MSP430FR58xx, MSP430FR59xx, and MSP430FR6xx Family User's Guide, (SLAU367O), Texas Instruments, 2017. 243, 245

MSP430FR5994 LaunchPad Development Kit (MSP-EXP430FR5994), (SLAU678A), Texas Instruments, 2016. 234

MSP430FR599x, MSP430FR596x Mixed-Signal Microcontrollers, (SLASE54C), Texas Instruments, 2018.

Texas Instruments MSP430x5xx/MSP430x6xx Family User's Guide, (SLAU208Q), Texas Instruments, 2018. 234, 239, 240, 241

TPS60310, TPS60311, TPS60312, TPS60313 Single-Cell to 3-V/3.3-V, 20 mA Dual Output High-Efficiency Charge Pump with Snooze Mode, (SLVS362A), Texas Instruments, Dallas, TX, 2001. 232, 251

5.12 CHAPTER PROBLEMS

Fundamental

1. Draw a block diagram and describe the operation of the power management system (PMM).
2. Draw a block diagram and describe the operation of the supply supervisor system (SVS).
3. Draw a block diagram and describe the operation of the clock system (CS).
4. Describe the differences of the three clocks on the MSP430: the MCLK, ACLK, and SMCLK.
5. What is the difference between supply voltage supervision (SVS) and monitoring?
6. What is the difference between a primary and a secondary voltage sources?

Advanced

1. Design a 3.3 VDC source for an MSP430 using a single AA battery. Fully specify all components.
2. Design a 3.3 VDC source for an MSP430 using a battery and a regulator. Fully specify all components.
3. Write a single-page point paper summarizing best practices for low-power operation.
4. Write a single-page point paper on the concept of battery capacity.
5. Construct a table summarizing low-power modes (LPMs). The table should include bit settings to enter the specific LPM and features available in the mode.
6. Construct an experiment to monitor battery voltage degradation during use. Plot results for several different battery technologies.
7. Construct a table summarizing available primary and secondary battery sources. At a minimum, the table should include common battery sizes (AA, AAA, C, D, and 9 VDC) and their capacity.

Challenging

1. Write a function in C to place the MSP430 in a specified low-power mode (LPM). The desired LPM is passed into the function as a variable.
2. Compile a list of best practices to operate the MSP430 microcontroller in the most efficient manner.

CHAPTER 6

MSP430 Memory System

Objectives: After reading this chapter, the reader should be able to:

- describe the importance of different memory components in a microcontroller-based system;
- employ the binary and hexadecimal numbering systems to describe the contents or address of a specific memory location;
- specify the length and width of a memory component;
- describe the function of the address, data, and control buses of a memory component;
- list the steps required for a memory component to be read from or written to;
- describe the difference between a Harvard and a von Neumann microcontroller architecture;
- provide the distinguishing features of RAM, ROM, and EEPROM type memory components;
- sketch the memory map for the MSP430FR2433 and the MSP430FR5994 microcontrollers;
- list the advantages of employing DMA techniques to transfer data using the MSP430FR5994;
- program the MSP430 DMA controller to transfer data from different portions of memory; and
- explain the key attributes of FRAM.

6.1 OVERVIEW

In general, there are two purposes for a memory system in a microcontroller: the storage and retrieval space for computer instructions and the storage and working space for data. For a memory-mapped I/O system, memory locations are used to capture inputs, store outputs, and

program subsystems of microcontrollers. As its name implies, the memory system allows a microcontroller to retain the program it is supposed to execute. Various memory components also allow data to be stored and modified during program execution.

Memory is essential in all microcontroller-based applications. In a smart home application, memory is used to store the algorithm to control the environmental factors serving the entire home. Other memory components are used to store and allow the update of combinations to secure areas of the home, control temperature, and to activate lawn irrigation systems. Furthermore, external memory components can be used for data logging. For example, key smart home parameters such as temperature and humidity may be measured and stored on a multi media card/secure digital (MMC/SD) card. The data may be logged over a long period of time. The MMC/SD card may be removed from the microcontroller-based system and read by a personal computer (PC) to examine and analyze the data. From this analysis, trends may be established to optimize the operation of the smart home control algorithm.

The intent of this chapter is to allow the reader to become acquainted with memory concepts, types of memory, and its operation. The chapter begins with a review of key memory concepts and terminology. This is followed by detailed maps of the MSP430FR2433 and the MSP430FR5994 microcontroller memory components. The different memory systems onboard the MSP430 are then discussed. These include the FRAM, RAM, and the DMA memories. We also provide examples on how to equip a LaunchPad with an MMC/SD card. The chapter concludes with a laboratory exercise detailing the operation of the DMA memory controller.

6.2 BASIC MEMORY CONCEPTS

In this section, we review terminology and concepts associated with microcontroller memory. Figure 6.1a provides a general model for a memory system. A microcontroller's memory system consists of a variety of memory technologies, including RAM, ROM, and EEPROM. Each technology has a specific function within the microcontroller.

Memory may be viewed as a two-dimensional (2D) array of storage elements called bits. A memory bit can store a single piece of digital information: a logic high or a logic low. Rather than access a single bit for reading or writing, memory is typically configured such that a collection of bits is read or written in parallel. The width, or how many bits are simultaneously accessed, is a function of the specific microcontroller. Memory widths of a byte (8 bits) or a double byte (16 bits) are common. The term "word" is often used to describe the width of the memory system.

A memory system may be viewed as a series of different memory locations each with a separate and unique address, as shown in Figure 6.1b. At each address is the capacity to store a memory word of n data bits.

6.2.1 MEMORY BUSES

Memory system activities are controlled by several different buses including the address bus, the data bus, and the control bus. A bus is a collection of conductors with a common function. The

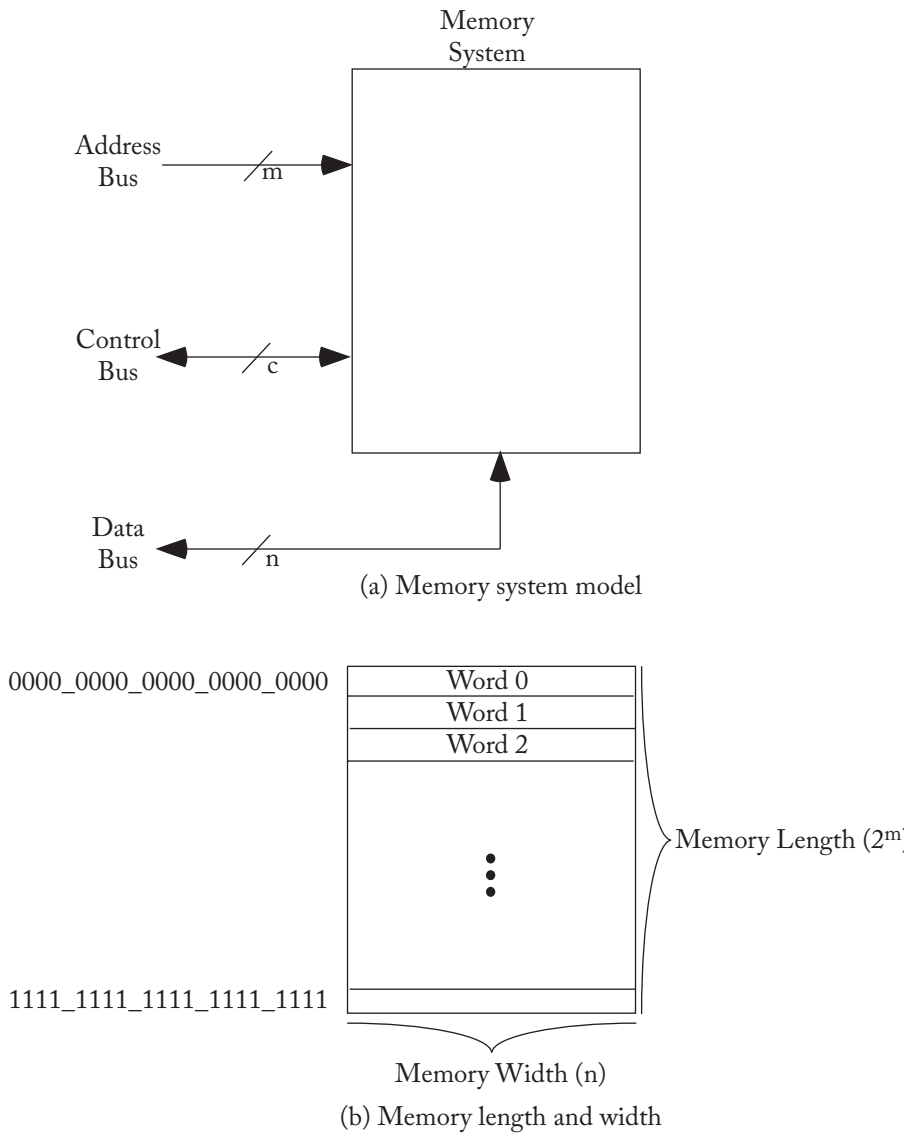


Figure 6.1: (a) General model for a memory system and (b) memory length and width.

address bus contains a number (m) of separate address lines. Using linear addressing techniques, the expression $2^{address\ lines}$ shows the number of uniquely addressable memory locations. For example, some variants of the MSP430 microcontroller have 20 address lines and, therefore, may separately address 2^{20} or 1,048,576 different memory locations.

The data bus width (n) usually matches the width of memory or the number of bits stored at each memory location. This allows the contents of a specific memory location to be read from or written to simultaneously. The MSP430 microcontroller has a 16-bit data path. The width of the data path also determines the maximum size of mathematical arguments that can be easily processed by the microcontroller. For example, with a 16-bit data width, the maximum unsigned integer that may be processed without overflow is 2^{16} or 65,535.

The control bus consists of the signal lines required to perform memory operations such as read and write. There are typically control signals to specify the memory operation (read or write), a clock input, and an enable output for the memory system.

6.2.2 MEMORY OPERATIONS

Operations that are typically performed on a memory system include read from and write to the memory system. The memory read operation consists of the following activities.

- The address of the memory location to be read is provided by the microcontroller on its address bus to the memory system.
- The control signal to read the specified memory location is asserted by the microcontroller.
- The data at the specified memory location is fetched from memory and placed on the memory system data lines.
- The control signal to enable the memory system output is asserted. This allows the fetched memory data access to the data bus.

The memory write operation consists of the following activities:

- The address of the memory location to be written to is provided by the microcontroller on its address bus to the memory system.
- The data to be written to the specified memory location is provided by the microcontroller on its data bus.
- The control signal to write to the specified memory location is asserted.
- The data is written to the specified memory location.

6.2.3 BINARY AND HEXADECIMAL NUMBERING SYSTEMS

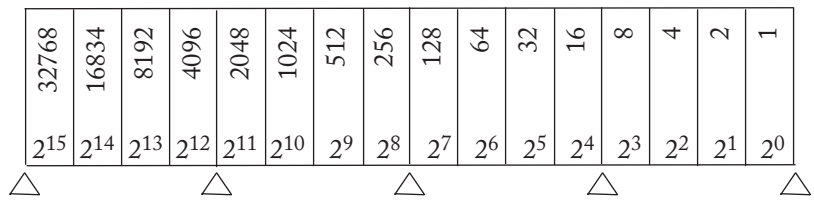
The binary, or base 2, numbering system is used to specify addresses and data within microcontroller systems. Examples of binary numbers and their equivalent base ten numbers are provided in Figure 6.2a. The largest unsigned integer that can be specified with a 16-bit binary number is 1111_1111_1111_1111 or 65, 535. Large binary numbers are difficult to read. To help in the readability of lengthy binary numbers, underscores maybe inserted between every four bits.

32768	16834	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}

(a) Binary number system

Binary	Hex	Binary	Hex	Binary	Hex	Binary	Hex
0000	0	0100	4	1000	8	1100	c
0001	1	0101	5	1001	9	1101	d
0010	2	0110	6	1010	a	1110	e
0011	3	0111	7	1011	b	1111	f

(b) Hexadecimal number system



(c) Binary to hexadecimal conversion

Figure 6.2: (a) The binary numbering system, (b) the hexadecimal numbering system, and (c) conversion from binary to hexadecimal.

As previously mentioned, the address bus of some variants of the MSP430 is 20 bits wide and the data bus is 16 bits. Rather than specify large binary numbers, the contents of the address and data bus are often expressed in the hexadecimal numbering system to enhance readability. A set of equivalent numbers of the binary system and the hexadecimal system is shown in Figure 6.2b.

To convert from the binary numbering system to the hexadecimal system, binary bits are grouped in fours from either side of the radix point. Each group of four binary bits is represented by its hexadecimal equivalent. Various methods are used to indicate that a specific number is represented in the hexadecimal numbering system. In assembly language, the number is followed by an "h". In the C programming language the number is preceded by a 0x. Various documen-

tation sources will place a dollar sign ($) before the numerical value or a subscripted 16 by the number indicating the hexadecimal content.

Microcontroller memory capacity is commonly specified in kilobytes (kB). The prefix kilo specifies (10^3) or for a kilobyte, 1,000 bytes. However, in common computer usage a kilobyte is used to specify 1,024 bytes or 2^{10} bytes.

Examples:

1. Some variants of the MSP430 has 16 address lines, giving it the capability to separately address 65,536 different memory locations. What is the address of the first and last memory location expressed in hexadecimal?

   Answer: The first and last addressable memory locations expressed in hexadecimal notation are $(0000)_{16}$ and $(FFFF)_{16}$.

2. Express the hexadecimal number $(CF)_{16}$ in binary.

   Answer: Each hexadecimal value is converted into its four bit binary equivalent resulting in the value of $(1100\_1111)_2$.

3. The MSP430 has the value of $(0001\_1010\_1111\_1100)_2$ present on the data base. Express the value in hexadecimal.

   Answer: Each group of four binary bits is expressed with its corresponding hexadecimal equivalent resulting in $(1AFC)_{16}$.

6.2.4 MEMORY ARCHITECTURES

There are two basic types of computer architectures based on the memory organization: the von Neumann architecture and the Harvard architecture. The von Neumann architecture has computer instructions and data resident within the same memory system, whereas the Harvard architecture provides separate paths to obtain instructions and data. The MSP430 microcontroller employs the von Neumann-type architecture since data and instructions are both retained within the same memory component.

6.2.5 MEMORY TYPES

Memory systems typically have several different types of memory technology available for use. Each technology type has its inherent advantages and disadvantages. We briefly describe each type.

RAM

RAM is volatile. That is, it only retains its memory contents while power is present. Within a microcontroller system, RAM memory is used for storing global variables, local variables,

stack implementation, and the dynamic allocation of user-defined data types during program execution.

ROM

ROM is non-volatile. That is, a ROM memory retains its contents even when power is lost. A ROM variant, EEPROM, is often referred to as flash memory. Flash memory is used to store programs and system constants that must be retained when system power is lost (non-volatile memory).

FRAM

Some variants of the MSP430, including the MSP430FR2433 and the MSP430FR5994, have been equipped with FRAM memory. FRAM is a nonvolatile, ULP with access speeds similar to RAM. It has been termed a universal memory because it can be used for storing program code, variables, constants, and for stack operations. Note these functions are typically performed by nonvolatile ROM and volatile RAM. FRAM also has a high level of write endurance on the order of 10^{15} cycles [SLAA526A, 2014, SLAA628, 2014].

External Memory Components

A microcontroller's memory system may be enhanced or extended using external memory components. For example, bulk storage capability may be added to a microcontroller-based system by interfacing a MMC/SD card. The MMC/SD card is equipped with a large complement of flash memory. The MMC/SD card is interfaced to the microcontroller via a serial communication link. The MMC/SD card is typically housed in a socket for easy removal from the host microcontroller-based system [SanDisk, 2000]. With an MMC/SD card, data may be logged over a long period of time. The MMC/SD card may then be removed from the microcontroller--based system and read by a personal computer (PC) to examine and analyze the data. The MSP-EXP430FR5994 LaunchPad is equipped with an onboard SD card.

Examples: A microcontroller-based application is being developed to log wind data at various remote locations over long periods of time to determine the efficacy of a wind energy farm at a specific site. Answer the following questions based on this scenario.

1. The algorithm to store the data is fairly complex and will require much storage space. What memory component must you insure is adequate to hold the algorithm?

   Answer: The coded algorithm to control the data collection system is stored in flash or FRAM memory. An MSP430 variant must be chosen that has sufficient memory capacity to hold the algorithm.

2. A good design technique is to compartmentalize specific algorithm operations into subroutines or functions. What memory component is required to support the call to subroutines or functions?

Answer: When a subroutine or function is called, local variables are placed on the stack. The stack is typically implemented as a portion of RAM memory.

3. The data logging system will be dispersed at a number of locations on existing farms and ranches. The plan is to collect the data over a six-month period and then have the property owner transfer the data to a central facility for processing. What is the appropriate memory technology to use in this situation?

   Answer: A microcontroller-based data collection system equipped with a removable MMC/SD card would be a good choice in this situation. The data could be collected for a long period of time, and the MMC/SD card could then be removed and sent to the central facility.

4. How do you determine the required capacity for the MMC/SD card to log data over a six-month period?

   Answer: To determine the required memory capacity the following parameters must be considered.

   - How many data variables are collected (e.g., date, time, temperature, wind speed, altitude) at a time?
   - In what format will the data be stored (e.g., integers, floating point numbers, custom abstract data type such as a record)?
   - How often will data be collected (e.g., every 15 min, hourly, every 6 h, daily)?
   - Over what time period will data be collected?

6.2.6 MEMORY MAP

The memory map is a visualization tool used to map the memory system onboard the microcontroller. As previously mentioned, some variants of the MSP430 has a 20 bit memory address. This allows the microcontroller to span the address memory space from $(00000)_{16}$ to $(fffff)_{16}$. Although the microcontroller may span this space, it does not necessarily mean there are memory components installed at each location. A memory map shows which addresses are occupied by a specific memory component and what locations are currently available for connection to other devices. The memory map for the MSP430FR2433 microcontroller is shown in Figure 6.3 and the one for the MSP430FR5994 microcontroller in Figure 6.4.

There are a variety of memory technologies within the MSP430 memory map. For each memory component, the start and stop address is provided as well as the span on the memory component. The span is provided as the number of locations in hexadecimal, decimal, and rounded off to the nearest byte. Most of the memory technologies provided in the memory map have already been discussed. Some require additional comment, which follows.

	Memory Contents	Start Address	Stop Address	Span
$(FFFF)_{16}$	Interrupt Vectors (FRAM)	$(FF80)_{16}$	$(FFFF)_{16}$	$(80)_{16}$ $(128)_{10}$ ~ 128 bytes
$(FF80)_{16}$	Code Memory (FRAM)	$(C400)_{16}$	$(FFFF)_{16}$	$(3C00)_{16}$ $(15360)_{10}$ ~ 15K bytes
$(C400)_{16}$				
$(3000)_{16}$	Random Access Memory (FRAM)	$(2000)_{16}$	$(2FFF)_{16}$	$(01000)_{16}$ $(4096)_{10}$ ~ 4K bytes
$(2000)_{16}$	Information Memory (FRAM)	$(1800)_{16}$	$(19FF)_{16}$	$(00200)_{16}$ $(512)_{10}$ bytes
$(1800)_{16}$	Bootstrap Loader Segment 1	$(1000)_{16}$	$(17FF)_{16}$	$(0800)_{16}$ $(2048)_{10}$ ~ 2K bytes
$(1000)_{16}$	Peripherals	$(0000)_{16}$	$(0FFF)_{16}$	$(1000)_{16}$ $(4096)_{10}$ ~ 4K bytes
$(0000)_{16}$				

Figure 6.3: The MSP430FR2433 memory map.

Bootstrap Loader

The bootstrap loader (BSL) portion of flash memory allows the user to interact with the flash memory and RAM onboard the MSP430 microcontroller. Specifically, the user can interact with the MSP430 via a host PC during prototype development. Data is exchanged between the host and the microcontroller via a serial link [slau319x].

Interrupt Vectors

Interrupts provide the microcontroller the capability to break out of routine processing and temporarily respond to a higher priority event. When an interrupt occurs, the microcontroller will temporarily suspend normal program execution and, instead, execute an interrupt service routine for the specific interrupt. The interrupt vectors between memory locations $(0FF80)_{16}$ and $(0FFFF)_{16}$ show the starting locations for each of the interrupt service routines.

	Memory Contents	Start Address	Stop Address	Span
$(FFFF)_{16}$	Interrupt Vectors (FRAM)	$(0FF80)_{16}$	$(0FFFF)_{16}$	$(80)_{16}$ $(128)_{10}$ ~ 128 bytes
$(0FF80)_{16}$	Code Memory (FRAM)	$(04000)_{16}$	$(043FFF)_{16}$	$(40000)_{16}$ $(262,144)_{10}$ ~ 256K bytes
$(03C00)_{16}$	Random Access Memory (RAM)	$(01C00)_{16}$	$(03BFF)_{16}$	$(2000)_{16}$ $(8192)_{10}$ ~ 8K bytes
$(01C00)_{16}$				
	Device Descriptor (TLV) (FRAM)	$(01A00)_{16}$	$(01AFF)_{16}$	$(00100)_{16}$ $(256)_{10}$ bytes
$(01A00)_{16}$	Information Memory A-D (FRAM)	$(01800)_{16}$	$(019FF)_{16}$	$(00200)_{16}$ $(512)_{10}$ bytes
$(01800)_{16}$	Bootstrap Loader Segment 0-3	$(01000)_{16}$	$(017FF)_{16}$	$(800)_{16}$ $(2048)_{10}$ ~ 2K bytes
$(01000)_{16}$	Peripherals	$(00020)_{16}$	$(00FFF)_{16}$	$(\sim 1000)_{16}$ $(4096)_{10}$ ~ 4K bytes
$(00000)_{16}$				

Figure 6.4: The MSP430FR5994 memory map.

6.2.7 DIRECT MEMORY ACCESS (DMA)

DMA provides the capability to move data from memory location to memory location without involving the CPU. This is especially useful for low-power operation when moving data from peripherals to specific memory locations [SLAU367O, 2017].

6.3 ASIDE: MEMORY OPERATIONS IN C USING POINTERS

Before delving into a detailed look at the MSP430 microcontroller's memory system, we need to discuss the concept of pointers in the C language. Pointer syntax allows one to easily refer to a memory location's address and the data contained at the address.[1]

[1] The information on pointers and examples provided were adapted from Dr. Jerry Cupal's, EE4390 Microprocessors class notes.

A pointer is another name for an address. To declare a pointer, an asterisk (∗) is placed in front of the variable name. The compiler will designate the variable as an address to the variable type specified.

Shown below is the syntax to declare an integer and a pointer (address of) for an integer. It is helpful to choose a variable name that helps you remember that a pointer variable has been declared. A pointer may be declared as a global or local variable.

```
int      x;
int      *ptr_x;
```

Once a pointer has been declared, there is other syntax that may be used in the body of a program to help manipulate memory addresses and data. The ampersand (&) is used to reference the address of a variable. Whereas, the asterisk (∗) is used as a dereference operator to refer to the contents of a memory location that a pointer is referencing.

Example: Given the code snapshot below, what is the final value in variable n?

```
int     m,n;      //declare integers m and n
int     *ptr_m;   //declare pointer to integer type

m = 10;           //set integer m equal to 10
ptr_m = &m        //set integer pointer to address of integer m
n = *ptr_m;       //Note use of the deference operator
```

Answer: The final value of n will be 10. The dereference operator in the last line of code refers to the contents of the memory location referenced.

Example: In this example, a technique is provided to point to a location in memory space.

```
char   *ptr_mem;            //configure a pointer to 8 bit locations

   :
   :

ptr_mem = (char*) 0x4000;  //cast the number 4000h into a pointer
                           //that points to character locations
```

The next example shows how a pointer may be used to move about memory locations by using some basic mathematical operations.

```
//*******************************************************************
// This function fills a buffer located between memory locations 2000h
// 2FFFh in memory space with incrementing 16 bit numbers.  The values
// loaded into memory start with 0000h and increments up to 0FFFh.
//*******************************************************************

int      x;                 //integer variable x
int      *ptr_buffer;       //pointer to buffer

void main(  )
{
ptr_buffer = (int*) 0x2000; //cast a pointer equal to 2000
for(x=0x0000; x<=0x0fff; x++)
  {
  *ptr_buffer = x;          //move the variable x into buffer
   ptr_buffer++;            //increment the pointer
  }
}
//*******************************************************************
```

6.4 DIRECT MEMORY ACCESS (DMA) CONTROLLER

If you have ever traveled in a big, busy city, you know how frustrating it can be to get bogged down in heavy traffic. It is particularly stressing if you are trying to get from one end of the city to the other and you are short on time. It would be nice to have a bypass around the city to avoid the downtown congestion and arrive safely and quickly to your destination. That is exactly what the DMA controller provides for the MSP430 microcontroller. The DMA feature allows for the fast and efficient transfer of data from one memory location to another without involving the CPU. In fact, the CPU may be left in a low power mode during DMA transfers. The MSP430FR5994 is equipped with six independent DMA channels. In this section, we discuss the overall DMA system, DMA addressing modes, transfer modes, triggering operations, the DMA register set and provide a representative DMA transfer example [SLAU367O, 2017].

6.4.1 DMA SYSTEM

The DMA system block diagram is shown in Figure 6.5. The figure may appear overwhelming at first but realize the DMA system in the MSP430F5438 has three independent channels. We investigate a single channel in Figure 6.6 [SLAU367O, 2017].

The DMA system allows the efficient transfer of data from a source location(s) in memory to a destination location(s). The source and destination addresses must be provided to the specific

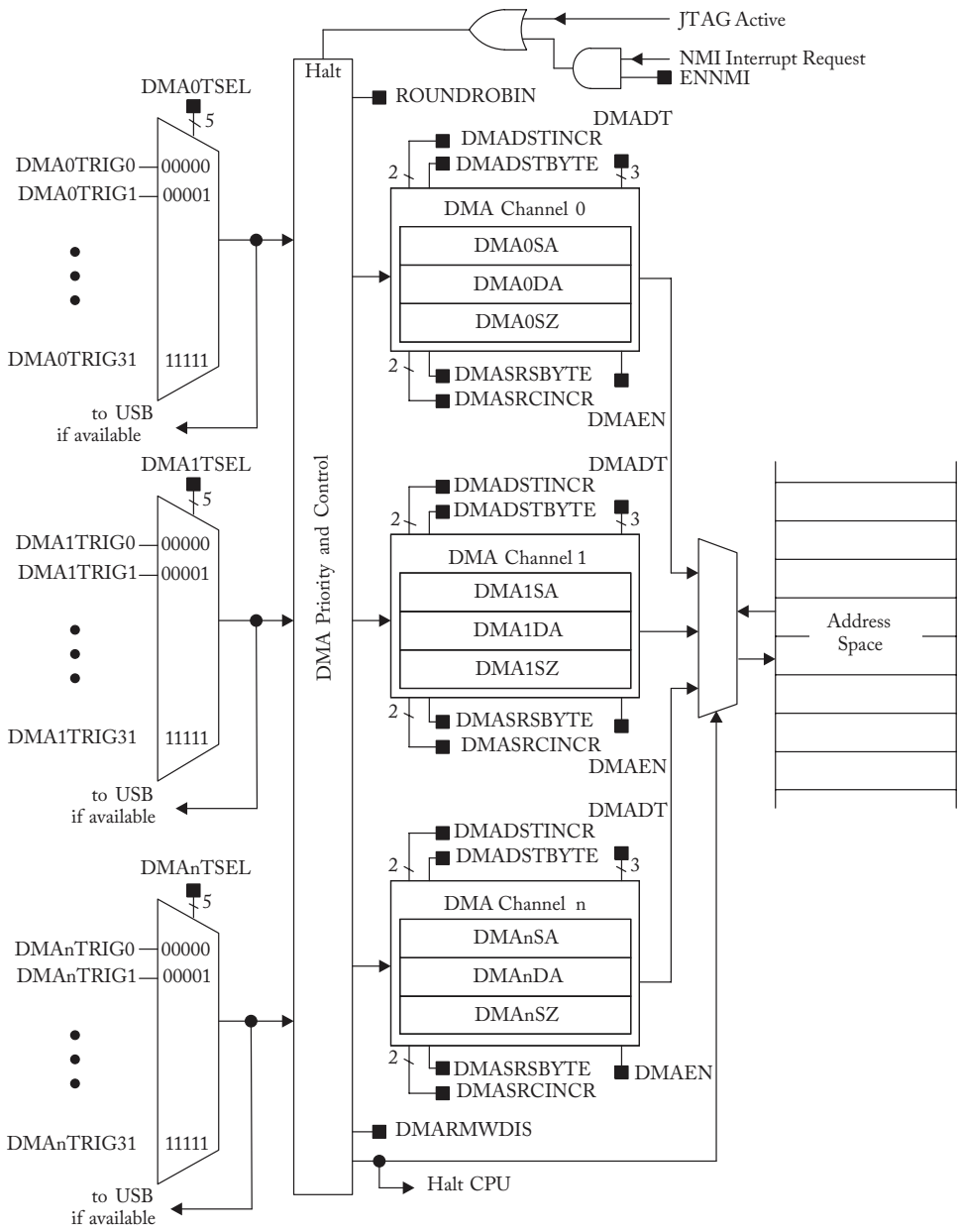
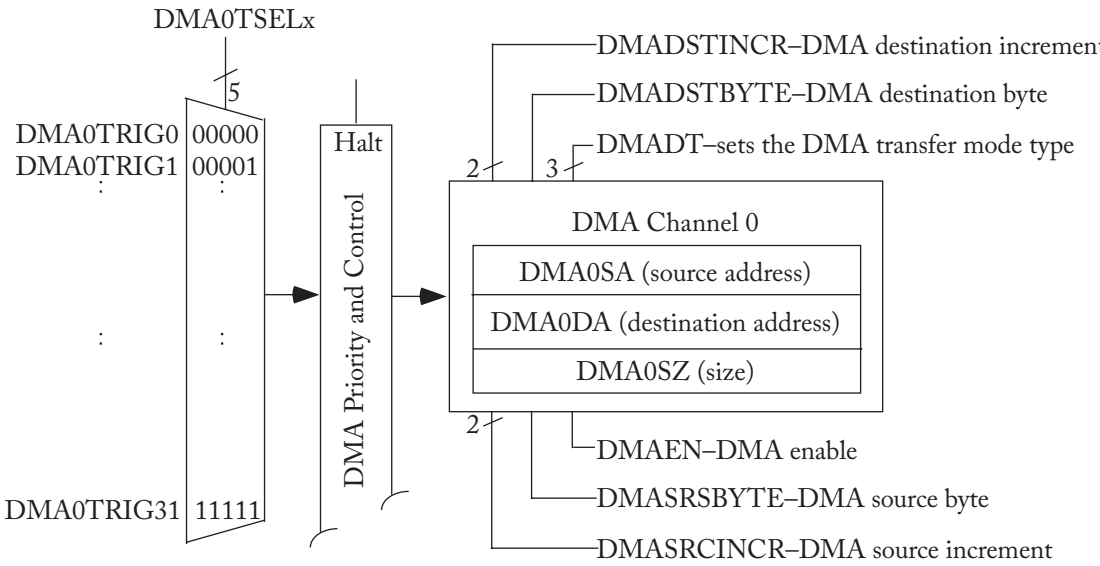


Figure 6.5: DMA block diagram [SLAU367O, 2017]. (Illustration used with permission of Texas Instruments (www.ti.com).)



DMA0TRIGx Sources

0 DMAREQ_	16 UCA1RXIFG
1 TA0CCR0 CCIFG	17 UCA1TXIFG
2 TA0CCR2 CCIFG T	18 UCB0RXIFG (SPI)
3 TA1CCR0 CCIFG	UCB0RXIFG0 (I2C)
4 TA1CCR2 CCIFG T	19 UCB0TXIFG (SPI)
5 TA2CCR0 CCIFG	UCB0TXIFG0 (I2C)
6 TA3CCR0 CCIFG T	20 UCB0RXIFG1 (I2C)
7 TB0CCR0 CCIFG	21 UCB0TXIFG1 (I2C)
8 TB0CCR2 CCIFG T	22 UCB0RXIFG2 (I2C)
9 TA4CCR0 CCIFG	23 UCB0TXIFG2 (I2C)
10 Reserved	24 UCB0RXIFG3 (I2C)
11 AES Trigger 0	25 UCB0TXIFG3 (I2C)
12 AES Trigger 1	26 ADC12 end of conversion
13 AES Trigger 2	27 LEA ready(2)
14 UCA0RXIFG	28 Reserved
15 UCA0TXIFG	29 MPY ready
	30 DMA0IFG
	31 DMAE0

Figure 6.6: DMA channel. (Illustration used with permission of Texas Instruments (www.ti.com).)

DMA channel as shown in Figure 6.6. This information is provided to the DMA channel via the DMA Source Address Register (DMAxSA) and the DMA Destination Register Address Register (DMAxDA). The "x" designates the DMA channel number. The DMAxSA, a 32-bit register, specifies the source address for a single transfer or the first source address for a block transfer. Similarly, the DMAxDA register specifies the destination address for single transfers or the first destination address for block transfers [SLAU367O, 2017].

The number of byte/word transfers must also be specified using the DMA Size Address Register (DMAxSZ). Since the DMAxSZ register is 16 bits, the maximum transfer size that may be specified is 65,535 bytes or words. During the DMA transfer event, the value within the DMAxSZ decrements with each byte/word transfer. Generally, two MCLK clock cycles are required to complete each transfer [SLAU367O, 2017].

Other parameters to tailor the DMA transfer for a specific application are provided within the DMA Channel x Control Register (DMAxCTL) by the following bits.

- DMASRCINCR specifies the DMA source increment or decrement.
- DMASRCBYTE specifies the DMA source as either a byte (1) or a word (0).
- DMAEN is the DMA enable bit.
- DMADT specifies the type of DMA transfer mode. There are six different transfer modes that may be specified. In general, the transfer mode dictates the type of transfer (single, burst, burst-block) and the type of triggered required.
- DMADSTBYTE selects the destination as either a byte (1) or a word (0).
- DMADSTINCR specifies the DMA destination increment or decrement.

The DMA transfer event may be initiated via a variety of different trigger sources as specified by the DMAxTSELx bits within DMA Control Register 0 (DMACTL0). We now take a more detailed look at each of these features.

DMA Adressing Modes

The different addressing modes available with the DMA system are illustrated in Figure 6.7 [SLAU367O, 2017]. Transfers may be specified from:

- a fixed source address to a fixed destination address,
- a fixed source address to a block of destination addresses,
- a block of source addresses to a fixed destination address, and
- a block of source addresses to a block of destination addresses.

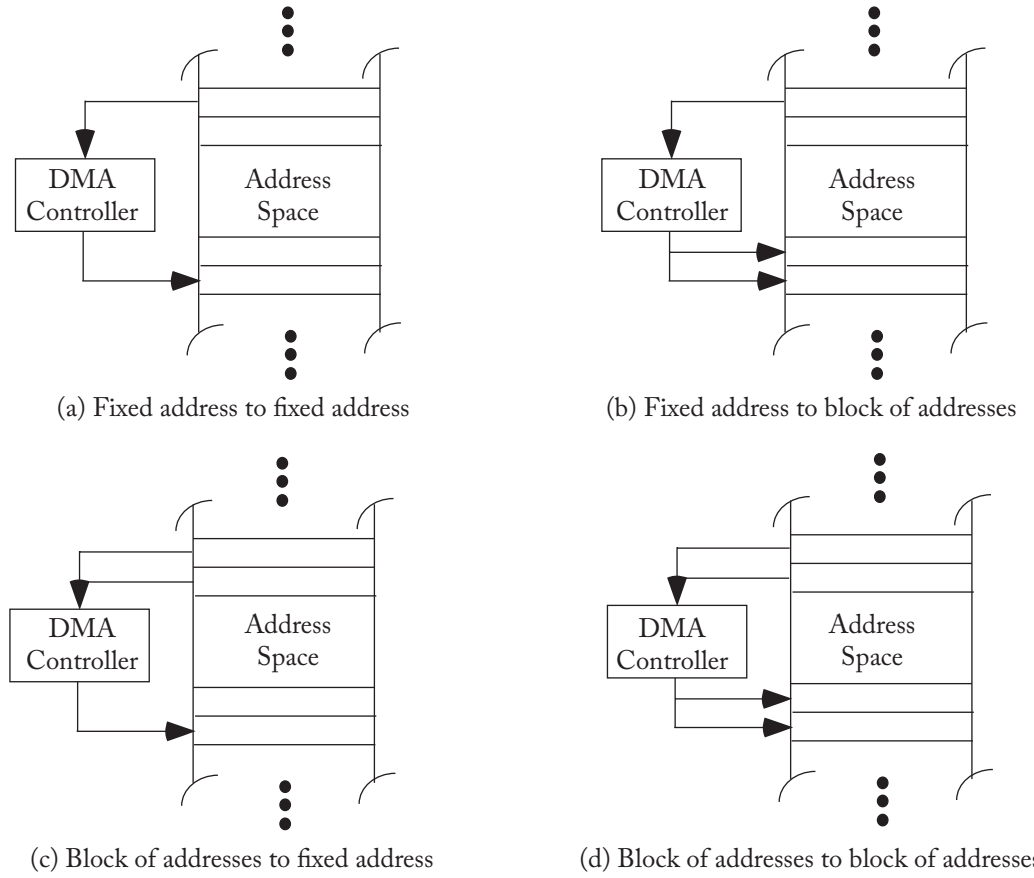


Figure 6.7: DMA addressing modes [SLAU367O, 2017].

DMA Transfer Modes

There are six different transfer modes that may be specified. In general, the transfer mode dictates the type of transfer (single, burst, burst-block) and the type of triggered required. The different DMA transfer modes are illustrated in Figure 6.8 [SLAU367O, 2017].

DMA Triggering

Figure 6.8 summarizes the different triggering requirements for a specific DMA transfer mode. The actual source of the triggering signal must also be specified. Figure 6.6 provided the various triggering sources available to a given DMA channel. The specific trigger signal is selected using the DMA0TSELx bits.

DMADTx	Transfer Mode	Trigger Description	DMAEN After Transfer
000	Single transfer	Each transfer requires a trigger	0
001	Block transfer	A complete block is transferred with one trigger	0
010, 011	Burst-block transfer	CPU activity is interleaved with a block transfer	0
100	Repeated single transfer	Each transfer requires a trigger	1
101	Repeated block transfer	A complete block is transferred with one trigger	1
110, 111	Repeated burst-block transfer	CPU activity is interleaved with a block transfer	1

Figure 6.8: DMA transfer modes [SLAU367O, 2017].

DMA Register Set

In the last several sections, we have seen the efficiency and flexibility of the DMA system in moving data from one location to another without involving the central processing unit. All DMA operations are configured using the DMA-related registers illustrated in Figures 6.9 and 6.10. We have already discussed many of the register features already. In this section, we provide a concise review of DMA associated registers.

The MSP430FR5994 uses DMA Control Register 0 through 2 (DMACTL0 to DMACTL2) to select the trigger source for DMA channels 0–5. The DMA trigger source is specified as a five-bit binary value as described in Figure 6.6 [SLAU367O, 2017].

The DMA systems is also equipped with interrupts. The MSP430 interrupt system will be discussed in an upcoming chapter.

The DMAxSA provides the source address for the DMA transfer; whereas, the DMA Destination Address Register (DMAxDA) provides the destination address for the transfer. The DMAxSZ specifies the number of byte/word data per block transfer [SLAU367O, 2017].

The main control register for the DMA system is the DMAxCTL illustrated in Figure 6.11.

DMA Control 0 Register (DMACTL0)

15	14	13	12	11	10	9	8
reserved	reserved	reserved	DMA1TSEL	DMA1TSEL	DMA1TSEL	DMA1TSEL	DMA1TSEL
r0	r0	r0	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)

7	6	5	4	3	2	1	0
reserved	reserved	reserved	DMA0TSEL	DMA0TSEL	DMA0TSEL	DMA0TSEL	DMA0TSEL
r0	rw-1	rw-0	rw-1	r-1	rw-0	rw-0	rw-0

DMA Control Register 4 (DMACTL4)

15	14	13	12	11	10	9	8
0	0	0	0	0	0	0	0
r0	r0	r0	r0	r0	r0	r0	r0

7	6	5	4	3	2	1	0
0	0	0	0	0	DMARMWDIS	ROUND ROBIN	ENNMI
r0	r0	r0	r0	r0	rw-0	rw-0	rw-0

DMA Interrupt Vector Register (DMAIV)

15	14	13	12	11	10	9	8
0	0	0	0	0	0	0	0
r0	r0	r0	r0	r0	r0	r0	r0

7	6	5	4	3	2	1	0
0	0	DMAIV					0
r0	r0						r0

Figure 6.9: DMA registers [SLAU367O, 2017].

DMA Source Address Register (DMAxSA)

31	30	29	28	27	26	25	24
Reserved							
r-0	r-0	r-0	r-0	r-0	r-0	r-0	r-0
23	22	21	20	19	18	17	16
Reserved				DMAxSA			
r-0	r-0	r-0	r-0	rw	rw	rw	rw
15	14	13	12	11	10	9	8
DMAxSA							
rw	rw	rw	rw	rw	rw	rw	rw
7	6	5	4	3	2	1	0
DMAxSA							
rw	rw	rw	rw	rw	rw	rw	rw

DMA Destination Address Register (DMAxDA)

31	30	29	28	27	26	25	24
Reserved							
r-0	r-0	r-0	r-0	r-0	r-0	r-0	r-0
23	22	21	20	19	18	17	16
Reserved				DMAxDA			
r-0	r-0	r-0	r-0	rw	rw	rw	rw
15	14	13	12	11	10	9	8
DMAxDA							
rw	rw	rw	rw	rw	rw	rw	rw
7	6	5	4	3	2	1	0
DMAxDA							
rw	rw	rw	rw	rw	rw	rw	rw

DMA Size Address Register (DMAxSZ)

15	14	13	12	11	10	9	8
0	0	0	0	0	0	0	0
r-0	r-0	r-0	r-0	r-0	r-0	r-0	r-0
7	6	5	4	3	2	1	0
0	0	DMAIV					0
r-0	r-0	r-(0)	r-(0)	r-(0)	r-(0)	r-(0)	r-0

Figure 6.10: DMA registers [SLAU367O, 2017].

DMA Channel x Control Register (DMAxCTL)

15	14	13	12	11	10	9	8
Reserved	DMADT			DMADSTINCR		DMASRCINCR	
r0	rw-(0)			rw-(0)	rw-(0)	rw-(0)	rw-(0)

DMADT: DMA Transfer Mode
- 000: single transfer
- 001: block transfer
- 010: burst-block transfer
- 011: burst-block transfer
- 100: repeated single transfer
- 101: repeated block transfer
- 110: repeated burst-block transfer
- 111: repeated burst-block transfer

DMASRCINCR: DMA Source Increment
- 00: source address is unchanged
- 01: source address is unchanged
- 10: source address is decremented
- 11: source address is incremented

DMADSTINCR: DMA Destination Increment
- 00: destination address is unchanged
- 01: destination address is unchanged
- 10: destination address is decremented
- 11: destination address is incremented

7	6	5	4	3	2	1	0
DMA DSTBYTE	DMA SRCBYTE	DMALEVEL	DMAEN	DMAIFG	DMAIE	DMAABORT	DMAREQ
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)

DMADSTBYTE: DMA Destination Byte
- 0: word
- 1: byte

DMASRCBYTE: DMA source byte
- 0: word
- 1: byte

DMALEVEL
- 0: rising edge trigger
- 1: high-level trigger

DMAEN: DMA Enable
- 0: Disabled
- 1: Enabled

DMAIFG: DMA Interrupt Flag
- 0: no interrupt pending
- 1: interrupt pending

DMAIE: DMA Interrupt Enable
- 0: disabled
- 1: enabled

DMAABORT: DMA Abort
- 0: DMA not interrupted
- 1: interrupted by NMI

DMAREQ DMA Request
- 0: No DMA start
- 1: Start DMA

Figure 6.11: DMA channel x control register (DMAxCTL) [SLAU367O, 2017].

6.4.2 DMA EXAMPLE: BLOCK TRANSFER

In this example a 16-word block from memory locations 0x1C20-0x1C2F is transferred to locations 0x1C40h-0x1C4fh using DMA0 in a burst block using software DMAREQ as the trigger.

```
//******************************************************************************
// --COPYRIGHT--,BSD_EX
// Copyright (c) 2015, Texas Instruments Incorporated
// All rights reserved.
//
//                  MSP430 CODE EXAMPLE DISCLAIMER
//
//******************************************************************************
//MSP430FR5x9x Demo - DMA0, Repeated Block to-from RAM,
//use Software Trigger
//
//Description:
//- A 16 word block from 1C20-1C2Fh is transferred to 1C40h-1C4fh using
//  DMA0 in a burst block using software DMAREQ trigger.
//- After each transfer, source, destination and DMA size are reset to
//  initial software setting because DMA transfer mode 5 is used.
//- P1.0 is toggled during DMA transfer only for demonstration purposes.
//- RAM location 0x1C00 - 0x1C3F used - make sure no compiler conflict
//- MCLK = SMCLK = default DCO
//
//
//                MSP430FR5994
//             -----------------
//         /|\|                 |
//          | |                 |
//          --|RST              |
//            |                 |
//            |             P1.0|-->LED
//
//William Goh, Texas Instruments Inc., October 2015
//Built with IAR Embedded Workbench V6.30 & Code Composer Studio V6.1
//******************************************************************************

#include <msp430.h>

int main(void)
{
WDTCTL = WDTPW | WDTHOLD;          //Stop WDT
                                   //Configure GPIO for LED
```

```
P1OUT = 0;
P1DIR = BIT0;

//Disable the GPIO power-on default high-impedance mode to activate
//previously configured port settings

PM5CTL0 &= ~LOCKLPM5;

                                    //Configure DMA channel 0
                                    //Source block address
__data16_write_addr((unsigned short) &DMA0SA,(unsigned long) 0x1C20);
                                    //Destination single address
__data16_write_addr((unsigned short) &DMA0DA,(unsigned long) 0x1C40);

DMA0SZ = 16;                        //Block size
DMA0CTL = DMADT_5 | DMASRCINCR_3 | DMADSTINCR_3; // Rpt, inc
DMA0CTL |= DMAEN;                   //Enable DMA0

while(1)
  {
  P1OUT |= 0x01;                    //P1.0 = 1, LED on
  DMA0CTL |= DMAREQ;                //Trigger block transfer
  P1OUT &= ~0x01;                   //P1.0 = 0, LED off
  }
}

//*******************************************************************
```

6.5 MSP430FR5994: MEMORY PROTECTION UNIT AND IP ENCAPSULATION SEGMENT

The MSP430FR5994 is equipped with a memory protection unit (MPU). The MPU provides protection against accidental writes to portions of memory designated as read only. The MPU allows configuration of the main memory into three separate segments. Segment sizes are set by designating the begin and end address for each segment. Furthermore, access rights for each memory segment can be independently set [SLAU367O, 2017].

The MSP430FR5994 also allows protection of intellectual property designated information with the IP Encapsulation Segment. The protected segment is designated by setting the start and stop segment addresses. Once configured, program code can be stored in the protected

area and also accessed via function calls. Furthermore, data stored within the segment can only be accessed via protected code within the segment [SLAU367O, 2017].

6.6 EXTERNAL MEMORY: BULK STORAGE WITH AN MMC/SD CARD

A MMC/SD card provides a handy method of providing a low power, non-volatile, and a small form factor (32 mm x 24 mm x 1.4 mm) bulk memory storage for a microcontroller. The microSD card has an even smaller form factor at 15 mm x 11 mm x 1 mm. The SD card is a smart peripheral device. It contains an onboard controller to manage SD operations. The SD card is useful for data logging applications in remote locations. If our goal was to measure wind resources at remote locations as potential windfarm sites, a MSP430 based data logging system equipped with an SD card could be used. Data could be logged over a long period of time and retrieved for later analysis on a PC.

6.7 LABORATORY EXERCISE: SD CARD OPERATIONS WITH THE MSP-EXP430FR5994

The MSP-EXP430FR5994 LaunchPad is equipped with an onboard microSD card (SD1). The Out-of-Box demo, available in MSPWare via the Resource Explorer, provides SD card support functions. The demo is preloaded to the MSP-EXP430FR5994 LaunchPad at the factory.

The Out-of-Box demo has three different modes of operation [SLAU678A, 2016]:

- Live Temperature Mode
- FRAM Log Mode
- SD Card Log Mode

The different modes of operation are selected via the Out-of-Box demo GUI. Instructions for downloading the GUI is provided in "MSP430FR5994 LaunchPad Development Kit (MSP-EXP430FR5994)" [SLAU678A, 2016]. The GUI is loaded and executed from the host PC.

Load the GUI software to the host PC. Follow the instructions provided in SLAU678A [2016] to examine the different operational modes of the Out-of-Box demo. The focus of this laboratory is the SD Card Log Mode.

The SD Card Mode interacts with the onboard RTC to provide an interrupt every 5 s. At each interrupt the MSP430 transitions out of LPM 3, illuminates the onboard green LED, obtains a time hack from the RTC, and performs ADC conversions to obtain the temperature and the battery level voltage. The time stamp and ADC values are then logged to a text file onboard the microSD card via the SPI. Commands and data are sent from the MSP430 to the SD card via this serial communication link. When data collection is complete, the microSD

card is removed from the MSP-EXP430FR5994. Its contents may be examined with the host PC.

To examine the supporting code in more detail, import the Out-of-Box demo software source code into Code Composer Studio and examine its features. Construct a UML Activity Diagram and structure chart for this program.

Currently, the Out-of-Box demo software is under the control of the GUI. The software has been written in clearly defined compartments so pieces may be adapted for specific applications and also adapted to other microcontrollers in the MSP430 line.

6.8 LABORATORY EXERCISE: MSP-EXP430FR5994 LAUNCHPAD DMA TRANSFER

Procedure:

- In the section on DMA transfer, an example was provided to move 32 integers from a given source address to a destination address using DMA channel 0. Develop a UML activity diagram for this example.
- Execute the code on the MSP430 experimenter board.
- Verify proper code execution by observing memory source and destination addresses using Code Composer Studio features.
- Write a new function that takes the contents of the source address and transfers its contents to the destination address on the first transfer. On subsequent transfers the original value is incremented and stored at subsequent destination addresses. Develop a UML activity diagram for this example.
- Execute the code on the MSP430 experimenter board.
- Verify proper code execution by observing memory source and destination addresses using Code Composer Studio features.

6.9 SUMMARY

In this chapter, with the help of the memory map, we presented the memory system of the MSP430 microcontroller. We demonstrated how contents of a memory location are accessed via read and write operations. We described the types of memories including RAM, ROM, and FRAM memory. We then discussed the organization and operation of onboard flash memory and the DMA system.

6.10 REFERENCES AND FURTHER READING

FRAM–New Generation of Non-Volatile Memory, Texas Instruments, 2010. www.ti.com/fram

MSP430FR2433 LaunchPad Development Kit (MSP-EXP430FR2433), (SLAU739), Texas Instruments, 2017.

MSP430FR2433 Mixed-Signal Microcontrollers, (SLAB034AD), Texas Instruments, 2017.

MSP430FR2433 Mixed-Signal Microcontroller, (SLASE59D), Texas Instruments, 2018.

MSP430FR4xx and MSP430FR2xx Family User's Guide, (SLAU445G), Texas Instruments, 2016.

MSP430FR58xx, MSP430FR59xx, and MSP430FR6xx Family User's Guide, (SLAU367O), Texas Instruments, 2017. 264, 266, 267, 269, 270, 271, 272, 273, 274, 276, 277

MSP430FR5994 LaunchPad Development Kit (MSP-EXP430FR5994), (SLAU678A), Texas Instruments, 2016. 277

MSP430FR599x, MSP430FR596x Mixed-Signal Microcontrollers, (SLASE54C), Texas Instruments, 2018.

MultiMedia Card Product Manual, SanDisk Corporate Headquarters, Sunnyvale, CA, 2000. www.sandisk.com 261

Texas Instruments MSP430 FRAM Quality and Reusability, (SLAA526A), Texas Instruments, 2014. 261

Texas Instruments MSP430 FRAM Technology-How to and Best Practices, (SLAA628), Texas Instruments, 2014. 261

Texas Instruments MSP430x5xx/MSP430x6xx Family User's Guide, (SLAU208Q), Texas Instruments, 2018.

6.11 CHAPTER PROBLEMS

Fundamental

1. Convert CAFEh to binary.
2. Convert $(1101\_1111\_0000\_1001)_2$ to decimal and hexadecimal numbers.
3. Convert $(11341)_{10}$ to binary and hexadecimal numbers.

4. A memory system is equipped with a 12-bit address bus. Using the linear addressing method, how many unique memory addresses are possible? What is the first and last memory addresses specified in binary? In hexadecimal?
5. A processor has a 16-bit data bus. What is the largest unsigned integer that may be carried by the bus? Signed integer?
6. Describe the different memory components available with the MSP430 microcontroller. Provide an application for each memory type.
7. Describe the purpose of the DMA system.

Advanced

1. Research the interface between the MSP430 and a MMC/SD card.
2. Sketch the memory map of the MSP430FR5994 microcontroller.

Challenging

1. Develop a MMC/SD card interface for the MSP430 microcontroller.
2. Write a function to clear a block of memory addresses in RAM memory. The start address and the number of memory locations to clear are passed into the function as arguments.
3. Write a function to transfer a block of memory locations from one address to another using DMA channel 2. The start address and the size of the memory block are passed into the function as arguments.

CHAPTER 7

Timer Systems

Objectives: After reading this chapter, the reader should be able to

- illustrate the use of the Watchdog timer;
- explain the operation of the real-time timer;
- explain the need and operation for the RTC;
- describe the timer features of Timer_A and Timer_B;
- program capture and compare subsystems to interface with external devices; and
- write basic programs using the timer subsystems (Watchdog, RTC, and capture/compare subsystems) and their interrupt modules.

7.1 INTRODUCTION

One of the main reasons for the proliferation of microcontrollers as the "brain" of embedded systems is their ability to interface with multiple external devices such as sensors, actuators, and display units among others. In order to communicate with such devices, however, microcontrollers must have capabilities to meet time constraints enforced by those devices. For example, an actuator which is controlled by a servo motor requires what is called a PWM signal with precise timing requirements as its input, while a communication device may need a unique pulse with a specified width to initiate its process. In other applications, microcontrollers need to capture the time of an external event or distinguish periodic input signals by computing their frequencies and periods. To meet these time constraints, embedded systems must have a fairly sophisticated timer system to generate a variety of clock signals, capture external events, and produce desired output time-related signals. The goal of this chapter is to address these capabilities of MSP430. We first present the clock systems of MSP430 followed by the Watchdog timer, basic timer, RTC, input capture, and output compare timer subsystems.

7.2 MOTIVATION: REAL-TIME LOCATION SYSTEMS (RTLS)

CenTrak, Incorporated provides tracking solutions for the medical community. One of their products, the InTouch Care™ Real Time Location System is used to track doctors, nurses,

medical staff, patients, and medical equipment. The MSP430's ability to operate with minimal power allows the InTouch Care to operate for six months without the need for a change of batteries. The timer system of the MSP430 embedded in the InTouch Care system is used to inform the system's location periodically to a central control location for tracking. Each InTouch Care unit comes with the DualTrack^{TM} communication system that utilizes radio frequency and infrared signals to transmit locations and receive commands. The system is used currently to streamline patient and equipment tracking in a number of medical facilities in the United States.

7.3 TIME-RELATED SIGNAL PARAMETERS

Throughout the history of microcontrollers, one of the main challenges was the need to operate with minimal power. The motivation comes from microcontroller applications that require a controller operating remotely without a continuous external power source. Since the power used by a microcontroller is directly proportional to the speed of transistors switching logic states, computer designers implemented multiple methods to reduce the clock speed. One method was to design a controller such that the CPU operates at a high clock speed while other subsystems run at a lower clock speed.

Such architectures with multiple clock sources can also allow programmers/engineers to turn off subsystems while they are not in use, saving more power for the overall embedded system. The MSP430 designers adopted this philosophy of providing users with multiple clock sources such that, depending on applications, one can have the flexibility to configure his or her controller appropriately. The flexible clock features of the MSP430 were previously discussed in Chapter 5.

Before proceeding forward, we briefly review time-related signal parameters.

7.3.1 FREQUENCY

Consider a signal, $x(t)$, that repeats a pattern over time. We call this signal periodic with period T, if it satisfies the following equation:

$$x(t) = x(t + T).$$

To measure the frequency of a periodic signal, we count the number of times a particular event repeats within one second period. The unit of frequency is the Hertz or cycles per second. For example, a sinusoidal signal with the 60 Hz frequency means that a full cycle of a sinusoid signal repeats itself 60 times each second or once every 16.67 ms.

7.3.2 PERIOD

The reciprocal of frequency is defined as period. If an event occurs with a rate of 1 Hz, the period of that event is 1 s. To find a period, given a frequency, or vice versa, we simply need to remember

their inverse relationship, $f = \frac{1}{T}$, where f and T represent a frequency and the corresponding period, respectively. Both periods and frequencies of signals are often used to specify timing constraints of embedded systems. For example, when your car is on a wintery road and slipping, the engineers who designed your car configured the anti-slippage unit to react within some millisecond period, say 20 ms. The constraint then forces the design team that monitors the slippage to program their monitoring system to check a slippage at a minimum rate of 50 Hz.

7.3.3 DUTY CYCLE

In many applications, periodic pulses are used as control signals of devices. A good example is the use of a periodic pulse to control a servo motor. To control the direction and sometimes the speed of a motor, a periodic pulse signal with a changing duty cycle over time is used. The periodic pulse signal shown in Figure 7.1 frame (a) is on for 50% of the signal period and off for the rest of the period. The pulse shown in frame (b) is on for only 25% of the same period as the signal in frame (a) and off for 75% of the period. The duty cycle is defined as the percentage of one period a signal is on. Therefore, we call the signal in frame (a) in Figure 7.1 as a periodic pulse signal with a 50% duty cycle and the corresponding signal in frame (b), a periodic pulse signal with a 25% duty cycle.

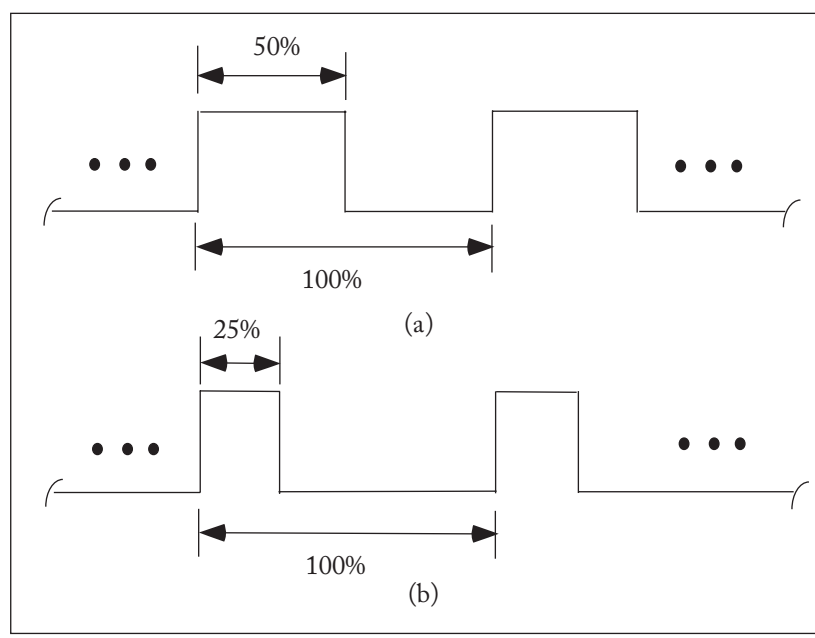


Figure 7.1: Two signals with the same period but different duty cycles. Frame (a) shows a periodic signal with a 50% duty cycle and frame (b) displays a periodic signal with a 25% duty cycle.

7.3.4 PULSE WIDTH MODULATION

In this section, we show how the speed of a DC motor can be controlled by a PWM signal. Suppose you have the circuit setup shown in Figure 7.2. The figure shows that the batteries are connected to power the motor through a switch. It is obvious that when we close the switch the DC motor will rotate and continue to rotate with a speed proportional to the DC voltage provided by the batteries. Now suppose we can open and close the switch rapidly. It will cause the motor to rotate and stop rotating per the switch position. As the time between the closing and opening of the switch decreases, the motor will not have time to make a complete stop and will continue to rotate with a speed proportional to the average time the switch is closed. This is the underlying principle of controlling DC motor speed using the PWM signal. When the logic of the PWM signal is high, the motor is turned on, and when the logic of the waveform is low, the motor is turned off. By controlling the time the motor is on, we can control the speed of the DC motor.

The duty cycle is defined as the fractional time the logic is high with respect to one cycle time (period) of the PWM signal. Thus, 0% duty cycle means the motor is completely turned off while 100% duty cycle means the motor is on all the time. Aside from motor speed control applications, PWM techniques are used in a wide variety of applications such as audio amplifiers, power supplies, heating units, and inverters.

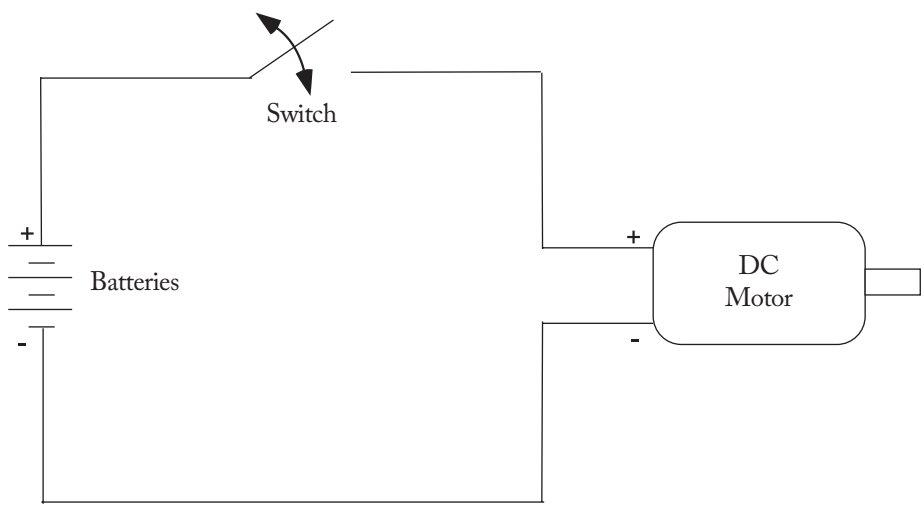


Figure 7.2: An example setup for controlling a DC motor.

7.4 OVERVIEW OF MSP430 TIMER FEATURES

Both the MSP430FR2433 and MSP430FR5994 LaunchPads are equipped with a host of timer features as shown in Figure 7.3. Each of these features will be discussed in upcoming sections along with code examples.

T.I. Product	Processor	Clock and Timer Features
MSP430FR2433 LaunchPad	MSP430FR2433	TIMER0_A3 - 3 capture/compare registers TIMER1_A3 - 3 capture/compare registers TIMER2_A2 - 2 capture/compare registers TIMER3_A2 - 2 capture/compare registers Real-Time Counter - 16-bit Watchdog Timer (WDT_A)
MSP430FR5994 LaunchPad	MSP430FR5994	TA0: Timer_A - 3 capture/compare registers TA1: Timer_A - 3 capture/compare registers TA2: Timer_A - 2 capture/compare registers TA3: Timer_A - 2 capture/compare registers TA4: Timer_A - 3 capture/compare registers TB0: Timer_B - 7 capture/compare registers Real-Time Clock B (RTC_B) Real-Time Clock C (RTC_C) Watchdog Timer (WDT_A)

Figure 7.3: MSP430 variants.

7.5 ENERGIA-RELATED TIME FUNCTIONS

For the remainder of the chapter we investigate time-related peripherals onboard the MSP430 including the Watchdog timer, Timer_A, and the Real-Time Clock (RTC_C). Before doing so, we review time-related functions available within Energia.

The Energia Development Environment has several built-in functions related to timing events, providing delays, or generating PWM signals. The functions include (www.energia.nu) the following.

- **millis():** This function provides the number of milliseconds that has occurred since the processor began running the current program.
- **micros():** This function provides the number of microseconds that has occurred since the processor began running the current program.
- **delay():** Provides a program pause for the specified number of milliseconds.
- **delayMicroseconds():** Provides a program pause for the specified number of microseconds. Note: This function is accurate for values 16,383 μs or less.
- **analogWrite():** The analogWrite function provides a 490 Hz pulse width modulated signal on the specified PWM capable pin. The duty cycle is provided as an argument to the function from 0–255. For example, to specify a 90% duty cycle, the value would be 230.

Example: In this example, time-related Energia functions are used to debounce an external switch input.

```
//*************************************************************************
//This example is provided with the Energia distribution and is used with
//permission of Texas Instruments, Inc.
//
//Debounce
//*************************************************************************
//Each time the input pin goes from LOW to HIGH (e.g., because of a
//push-button press), the output pin is toggled from LOW to HIGH or
//HIGH to LOW.
//
//The circuit:
//- LED attached from pin 13 to ground
//- Pushbutton attached from pin 2 to +3.3V
//- 10K resistor attached from pin 2 to ground
//
//created:  21 Nov 2006, David A. Mellis
```

```
//modified: 30 Aug 2011, Limor Fried
//modified: 27 Apr 2012, Robert Wessels
//
//This example code is in the public domain.
//*****************************************************************

const int buttonPin = PUSH2;        //number of the pushbutton pin
const int ledPin =  GREEN_LED;      //number of the LED pin

int ledState = HIGH;                //current state of the output pin
int buttonState;                    //current reading from the input pin
int lastButtonState = LOW;          //previous reading from the input pin

//the following variables are long's because the time, measured in
//miliseconds, will quickly become a bigger number than can be
//stored in an int.
long lastDebounceTime = 0;          //last time output pin toggled
long debounceDelay = 50;            //the debounce time; increase if the
                                    //output flickers

void setup()
{
pinMode(buttonPin, INPUT_PULLUP);
pinMode(ledPin, OUTPUT);
}

void loop()
{
//read the state of the switch into a local variable:
int reading = digitalRead(buttonPin);

//check to see if you just pressed the button
//(i.e., the input went from LOW to HIGH),  and you've waited
//long enough since the last press to ignore any noise:
//If the switch changed, due to noise or pressing:
if (reading != lastButtonState)
  {
  lastDebounceTime = millis();
  }
```

```
if ((millis() - lastDebounceTime) > debounceDelay)
  {
  //whatever the reading is at, it's been there for longer
  //than the debounce delay, so take it as the actual current state:
  buttonState = reading;
  }

//set the LED using the state of the button:
digitalWrite(ledPin, buttonState);

//save the reading.  Next time through the loop,
//it'll be the lastButtonState:
lastButtonState = reading;
}

//**********************************************************************
```

7.6 WATCHDOG TIMER

As the name implies, the primary purpose of the Watchdog timer in a microcontroller is to watch for and prevent software failure by forcing user code to refresh a designated control register periodically throughout the execution of a program. The secondary purpose of the Watchdog timer is to generate periodic time intervals for applications that require periodic, repeated services.

By software failure, we mean the execution of unintended instructions by MSP430, whether it is an unintended infinite loop or a wrong segment of program being executed due to hardware errors, programmer errors, or noise-related malfunctions. We now present how the reader can configure the Watchdog system to function as a software failure preventer and a periodic interval generator.

7.6.1 PROTECTING FROM SOFTWARE FAILURE

The Watchdog timer prevents software failure by enforcing the following rule. A 16-bit register, called the Watchdog count (WDTCNT) register, counts up at each clock cycle. When it reaches its limit, the Watchdog timer system initiates a power up clear reset (PUCR).[1] Thus, your program must clear the counter periodically before the counter reaches its limit. During normal program execution, counter reset instructions may be placed strategically throughout the code. When the code executes correctly, the Watchdog timer will be reset on a regular basis, indicating normal operation. However, if the code is not operating correctly, the Watchdog timer

[1]Unlike the power-on reset (POR), the power up clear reset (PUCR) does not change the values of the WDTCTL register.

will not be reset as required ,thus generating a flag or an interrupt. A user can select the limit values as 64, 512, 8192, or 32,768 (default), which correspond to using the WDTCNT register as a 6-, 9-, 13-, or 15-bit counter, respectively. The source for the clock cycle can be chosen either from the SMCLK (default) or the ACLK.

The function of the Watchdog timer is governed by programming the Watchdog timer control register (WDTCTL). To avoid accidental write to this register, it is password protected, which means to modify the contents of the register, one must first write 0x5A (password) to the upper byte of WDTCTL before configuring the Watchdog system using the lower byte of the same register. MSP430 designers also implemented another safety mechanism by resetting the controller, if a wrong password is sent to the upper byte of WDTCTL. Figure 7.4 shows the contents of the 16-bit register.

WatchDog Timer Register (WDTCTL)

15	14	13	12	11	10	9	8
Read as 069h WDTPW must be written as 05Ah							

7	6	5	4	3	2	1	0
WDTHOLD	WDTNMIES	WDTNMI	WDTTMSEL	WDTCNTCL	WDTSSEL	WDTISx	
rw-0	rw-0	rw-0	rw-0	r0(w)	rw-0	rw-0	rw-0

Figure 7.4: Watchdog timer register WDTCTL.

The 7th bit (WDTHOLD) is used to turn-on or turn-off the Watchdog timer. Setting this bit disables the Watchdog counter, and clearing this bit configures the system to function normally. Bits 6 and 5 are not used. Bit 4 (WDTTMSEL) determines the mode of operation for the Watchdog timer: setting this bit selects the interval timer mode while clearing this bit designates the Watchdog mode. Writing a logic one to bit 3 (WDTCNTCL) clears the counter. This is how your program can prevent the Watchdog timer from generating a PUCR. Once the WDTCNT is cleared, this bit is reset (0) automatically. The WDTSSEL bit (bit 2) selects the clock source for the counter. Setting this bit chooses the ACLK clock while clearing this bit selects the SMCLK clock. Finally, WDTISx bits (bits 1 and 0) are used to select the Watchdog timer reset periods as shown below:

- 00 – use 15 bit counter and count up to 32,768
- 01 – use 13 bit counter and count up to 8,192
- 10 – use 9 bit counter and count up to 512

- 11 – use 6 bit counter and count up to 64

The default value of the register selects the 15-bit counter using the SMCLK clock as the time source.

Associated with the control register is the IFG1 register. When the WDTCNT register reaches its limit, the WDTIFG flag (bit 0) in the IFG1 register, located at 0x0002, is set. This flag can be polled or can be used to initiate an interrupt when the Watchdog timer is used as a periodic interval timer.

Example: The following C code or the corresponding assembly code turns off the Watchdog timer, which is recommended during program development. Assuming that the registers are already defined with the proper names, the Watchdog timer may be turned off using the following C instruction:

```
WDTCTL = WDTPW + WDTHOLD
```

In Assembly Language, use:

```
MOV.W       #WDTPW+WDTHOLD, &WDTCTL
```

7.6.2 INTERVAL TIMER

The Watchdog timer can also be configured to generate a periodic interval. To do so, the WDTTMSEL bit (bit 4) of the WDTCTL register must be set and the interval period must be selected using the WDTISx bits (bits 2 to 0) and the WDTSSEL bits (bits 6, 5) of the same register. When the WDTCNT register reaches the designated limit, the WDTIFG flag in the Special Function Register SFRIFG1 register is set. If the WDTIE bit (bit 0) in the Special Function Register Interrupt Enable IE1 register 1 (SFRIE1.0) is set and the GIE bit (overall maskable interrupt system enable bit in the Status Register) is set, the Watchdog timer interrupt is triggered. Figure 7.5 shows the components of the interval timer along with the related interrupt system. Once the interrupt is serviced (interrupt service routine is executed), the WDTIFG flag is automatically cleared.

Example: Provided below is the C code segment that configures the MSP430FR5994 microcontroller board to toggle the logic state on port 1 pin 0 (red LED onboard the launchpad) every second.

```
//*******************************************************************
//MSP430FR5994 Example - WDT, Toggle P1.0, Use of WDT as a timer,
//
//Description: Toggle P1.0 using software timed by WDT ISR. Toggle rate
//is 1 sec based on 32kHz ACLK WDT clock source.  In this example, the
//WDT is configured to divide 32768 watch-crystal by 2^15 with an ISR
//triggered at 1Hz.
```

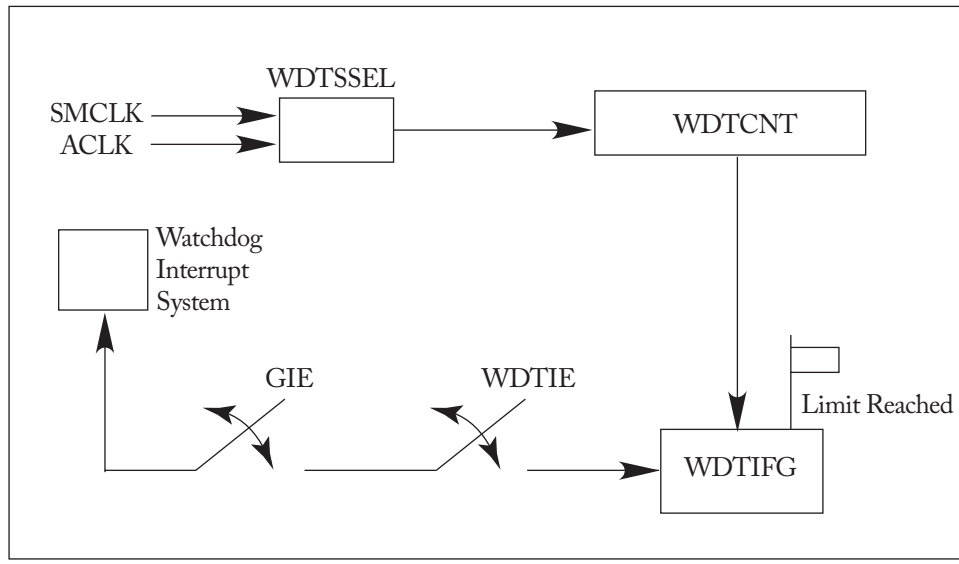


Figure 7.5: Watchdog timer as an interval generator.

```
//
//An external watch crystal is installed on XIN XOUT for the ACLK.
//(ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = default DCO)
//
//D. Pack and S. Barrett, July 2018, Built with CCS v8
//***************************************************************************

#include <msp430.h>

int main(void)
{
WDTCTL = WDTPW + WDTHOLD;       //Stop WDT during initialization
PMSCTL0 &= ~LOCKLPMS;          //Disable GPIO high-impedance mode
P1DIR |= BIT0;                     //Set P1.0 (red LED) to output
P1DIR |= 0x01;                     //Set P1.0 to output direction
WDTCTL = WDT_ADLY_1000;            //WDT 1 sec, ACLK, interval timer
SFRIE1 |= WDTIE;                   //Enable WDT interrupt
__enable_interrupt();              //Enable global interrupt

while(1)
  {
```

```
  }      //Wait for interrupts

}

//*********************************************************************
//Watchdog timer interrupt service routine
//*********************************************************************

#pragma vector=WDT_VECTOR
__interrupt void watchdog_timer(void)
{
P1OUT ^= 0x01;                    //Toggle P1.0 using exclusive-OR
}

//*********************************************************************
```

7.7 REAL-TIME CLOCK

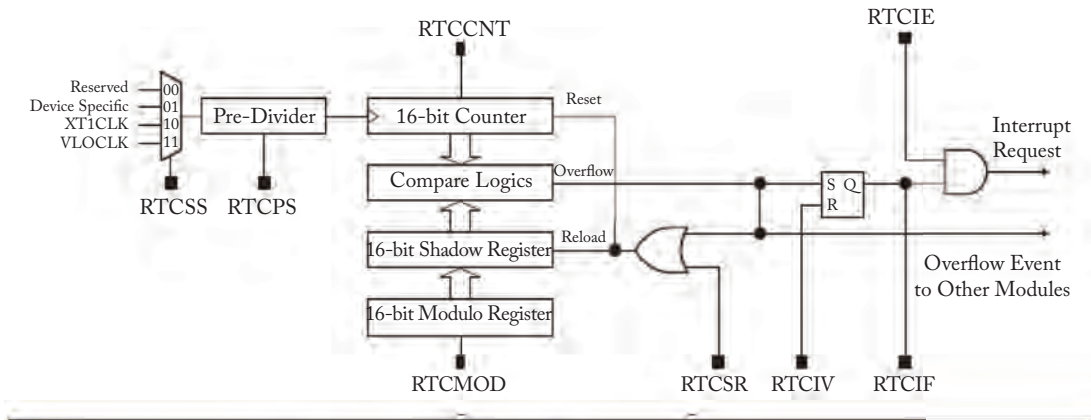
RTC features provide microcontrollers the ability to generate a periodic interrupt to accomplish periodic, important tasks. For example, while operating a motor, it is important to periodically monitor motor current as an indication of safe, non-obstructed operation. RTC features also provide the microcontroller to track calendar time based on seconds, minutes, hours, etc. Both the MSP430FR2433 and the MSP430FR5994 are equipped with RTC features. Since the RTC features onboard these two processors are different, they will be addressed separately.

7.8 REAL-TIME CLOCK-MSP430FR2433

The MSP430FR2433 is not equipped with calendar-based RTC features. Instead, the MSP430FR2433 may be equipped with an external RTC (Maxim DS3234) via the serial peripheral interface. An example is provided in Chapter 10.

The MSP430FR2433 is equipped with a 16-bit counter as shown in Figure 7.6. The counter may be used to generate a periodic interrupt. The time base for the RTC counter may be the ACLK, XT1CLK, or VLOCLK depending on the operating mode. The clock may then be pre-divided using the Real-time clock pre-divider select bits (RTCPS). The divided clock source signal is provided to the 16-bit counter. The counter's value is constantly compared to the value of the 16-bit Shadow register. When the two values are the same, an RTC interrupt is generated, if enabled.

Example: In this example the MSP430FR2433's RTC counter is configured to generate a 1 s periodic interrupt.



15	14	13	12	11	10	9	8
Reserved		RTCSS		Reserved		RTCPS	
r0	r0	rw-(0)	rw-(0)	r0	rw-(0)	rw-(0)	rw-(0)
7	6	5	4	3	2	1	0
Reserved	RTCSR	Reserved				RTCIE	RTCIFG
r0	w-(0)	r0	r0	r0	r0	rw-(0)	r-(0)

Real-Time Clock Control Register (RTCCTL)

RTCCTL[13:12]: RTCSS: clock source: 01 = device specific (SMCLK or ACLK) ,
10 = XT1CLK, 11 = VLOCLK

RTCCTL[10:8]: RTCPS: predivider:
000 = 1, 001 = 10, 010 = 100, 011 = 1000, 100 = 16, 101 = 64, 110 = 256, 111 = 1024

RTCCTL[6]: RTCSR: software reset: 1 = clear counter value,
reloads shadow register from modulo register

RTCCTL[1]: RTC interrupt enable: 0 = disabled, 1 = enabled

RTCCTL[0]: RTC interrupt flag: 0 = none pending, 1= interrupt pending

Figure 7.6: MSP430FR2433 RTC counter. (Illustration used with permission of Texas Instruments (www.TI.com).)

```
//******************************************************************
// --COPYRIGHT--,BSD_EX
// Copyright (c) 2015, Texas Instruments Incorporated
// All rights reserved.
//
//                  MSP430 CODE EXAMPLE DISCLAIMER
//
//******************************************************************
// MSP430FR243x Demo - RTC, toggle P1.0 every 1s
//
```

```
// Description: Configure ACLK to use 32kHz crystal as RTC clock,
//              ACLK=XT1=32kHz, MCLK = SMCLK = default DCODIV = ~1MHz.
//
//            MSP430FR2433
//         -----------------
//     /|\|                 |
//      | |                 |
//      | |        XIN(P2.0)|--
//      --|RST              |  ~32768Hz
//        |       XOUT(P2.1)|--
//        |                 |
//        |             P1.0|-->LED
//
//Ling Zhu, Texas Instruments Inc., Feb 2015
//Built with IAR Embedded Workbench v6.20 & Code Composer Studio v6.0.1
//******************************************************************

#include <msp430.h>

int main(void)
{
WDTCTL = WDTPW | WDTHOLD;               //Stop watchdog timer
P2SEL0 |= BIT0 | BIT1;                  //set XT1 pin as second function
do
  {
  CSCTL7 &= ~(XT1OFFG | DCOFFG);        //Clear XT1 and DCO fault flag
  SFRIFG1 &= ~OFIFG;
  }while (SFRIFG1 & OFIFG);             //Test oscillator fault flag

P1OUT &= ~BIT0;                         //Clear P1.0 output latch for
                                        //a defined power-on state
P1DIR |= BIT0;                          //Set P1.0 to output direction

PM5CTL0 &= ~LOCKLPM5;                   //Disable the GPIO power-on
                                        //default high-impedance mode
                                        //to activate previously
                                        //configured port settings


RTCMOD = 32-1;                          //RTC count re-load compare
```

```
                                        //value at 32.
                                        //1024/32768 * 32 = 1 sec.

                                        //Initialize RTC
                                        //Source = 32kHz crystal,
                                        //divided by 1024
RTCCTL = RTCSS__XT1CLK | RTCSR | RTCPS__1024 | RTCIE;
__bis_SR_register(LPM3_bits | GIE);     // Enter LPM3, enable interrupt
}

//-------------------------------------------------------------------
// RTC interrupt service routine
//-------------------------------------------------------------------

#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector=RTC_VECTOR
__interrupt void RTC_ISR(void)
#elif defined(__GNUC__)
void __attribute__ ((interrupt(RTC_VECTOR))) RTC_ISR (void)
#else
#error Compiler not supported!
#endif
{
switch(__even_in_range(RTCIV,RTCIV_RTCIF))
  {
  case  RTCIV_NONE:   break;          //No interrupt
  case  RTCIV_RTCIF:                  //RTC Overflow
                      P1OUT ^= BIT0;
                      break;
  default:            break;
  }
}

//-------------------------------------------------------------------
```

7.8.1 REAL-TIME CLOCK: RTC_B, RTC_C-MSP430FR5994

RTC_B and RTC_C, the MSP430 RTC, provide a clock based on seconds, minutes, hours, etc. The RTC time base is provided by a 32,768 Hz external crystal. This time base is shown as the BCLK in Figure 7.7. The time base is routed to the RTOPS and RT1PS dividers to provide

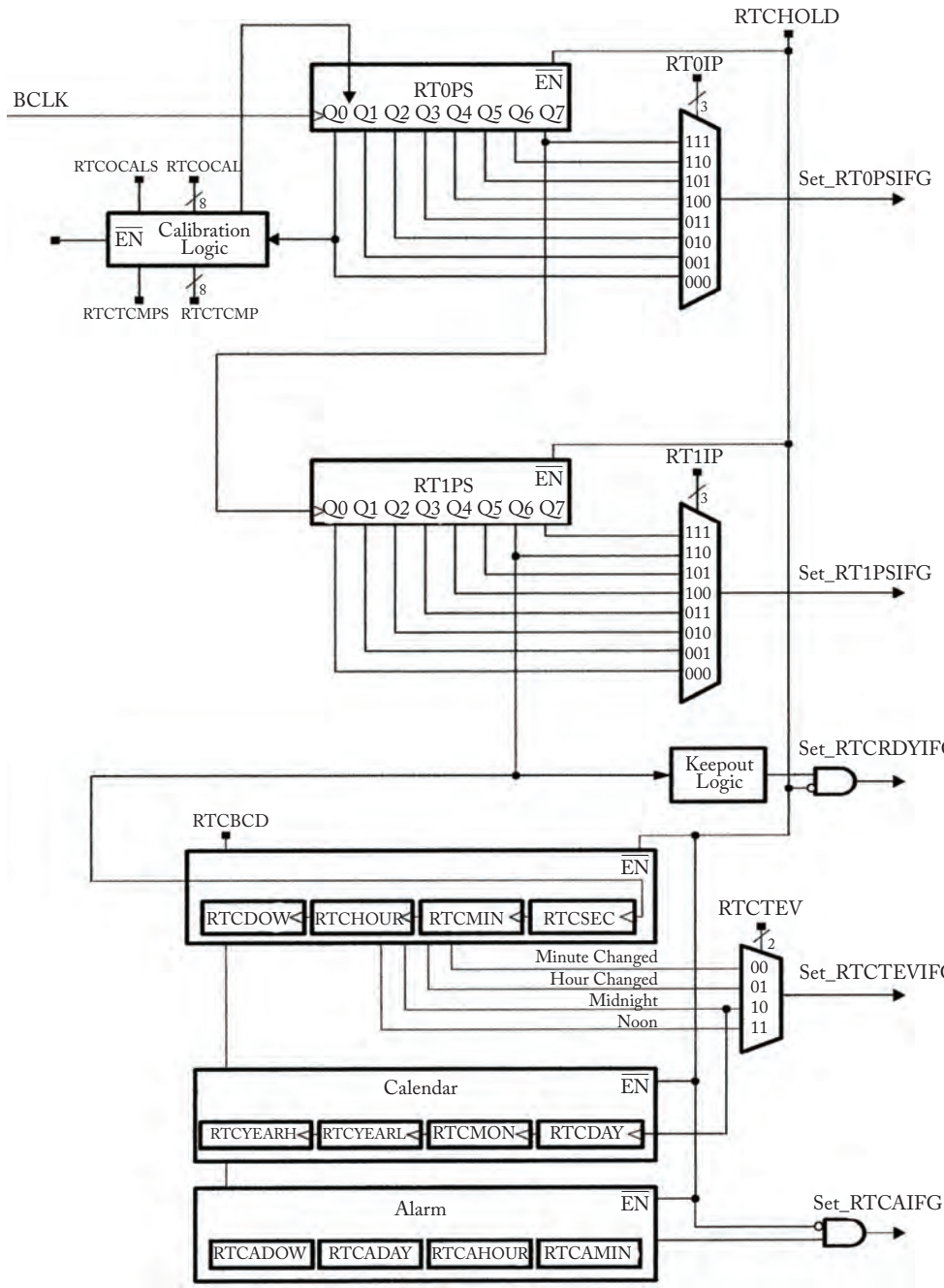


Figure 7.7: RTC [SLAU367O, 2017]. (Illustration used with permission of Texas Instruments (www.ti.com).)

a 1 Hz time base to the time-keeping registers. The register contains place holders for seconds (RTCSEC), minutes (RTCMIN), hours (RTCHOUR), day of the week (RTCDOW), day (RTCDAY), month (RTCMON) , and year (RTCYEARH, RTCYEARL). Data may be stored in BCD or hexadecimal binary format. BCD represents each digit in a number individually from 0–9 [SLAU367O, 2017].

The RTC_C is also equipped with an alarm function. The alarm is configured for a specific minute (RTCAMIN), hour (RTCAHOUR), day (RTCADAY), and day of the week (RTCADOW). The write operation for RTC control, clock, calendar, prescale, and offset error are key protected [SLAU367O, 2017].

The RTC_C is supported by six prioritized interrupts designated RT0PSIFG, RT1PSIFG, RTCRDYIFG, RTCTEVIFG, RTCAIFG, and RTCOFIFG. The six interrupt signal flags are combined to provide a single interrupt signal. When an interrupt occurs the interrupt vector register (RTCIV) provides the specific interrupt source [SLAU367O, 2017].

7.8.2 RTC REGISTERS

RTC_C is supported by a complement of registers. Details of specific register and bits settings are contained in *Texas Instruments MSP430FR58xx, MSP430FR59xx, and MSP430FR6xx Family* SLAU367O [2017] and will not be repeated here. The next two examples illustrate the use of the RTC in counter and clock mode.

Example. In this example the MSP430FR5994 is employed within the counter mode. An LED is toggled every second.

```
//******************************************************************************
// --COPYRIGHT--,BSD_EX
// Copyright (c) 2015, Texas Instruments Incorporated
// All rights reserved.
//
//                  MSP430 CODE EXAMPLE DISCLAIMER
//
//******************************************************************************
//MSP430FR5x9x Demo - RTC in Counter Mode toggles P1.0 every 1s
//
//This program demonstrates operation of the RTC in counter mode
//configured to source from the ACLK to toggle P1.0 LED every 1s.
//
//                MSP430FR5994
//             -----------------
//        /|\ |                 |
//         |  |              XIN|--
```

```
//         ---|RST              |  32768Hz
//            |             XOUT|--
//            |                 |
//            |             P1.0|-->LED
//
//William Goh, Texas Instruments Inc., October 2015
//Built with IAR Embedded Workbench V6.30 & Code Composer Studio V6.1
//******************************************************************

#include <msp430.h>

int main(void)
{
WDTCTL = WDTPW | WDTHOLD;               //Stop WDT
P1OUT &= ~BIT0;
P1DIR |= BIT0;
PJSEL0 = BIT4 | BIT5;                   //Initialize LFXT pins

//Disable the GPIO power-on default high-impedance mode to activate
//previously configured port settings
PM5CTL0 &= ~LOCKLPM5;

                                        //Configure LFXT 32kHz crystal
CSCTL0_H = CSKEY_H;                     //Unlock CS registers
CSCTL4 &= ~LFXTOFF;                     //Enable LFXT
do
  {
  CSCTL5 &= ~LFXTOFFG;                  //Clear LFXT fault flag
  SFRIFG1 &= ~OFIFG;
  }while (SFRIFG1 & OFIFG);             //Test oscillator fault flag
CSCTL0_H = 0;                           //Lock CS registers

                                        //Setup RTC Timer
RTCCTL0_H = RTCKEY_H;                   //Unlock RTC
RTCCTL0_L = RTCTEVIE_L;                 //RTC event interrupt enable

                                        //Ctr Mode, RTC1PS, 8-bit ovf
RTCCTL13 = RTCSSEL_2 | RTCTEV_0 | RTCHOLD;
RTCPS0CTL = RT0PSDIV1;                  //ACLK, /8
```

```
                                        //out from RT0PS, /16
RTCPS1CTL = RT1SSEL1 | RT1PSDIV0 | RT1PSDIV1;
RTCCTL13 &= ~(RTCHOLD);                 //Start RTC

__bis_SR_register(LPM3_bits | GIE);     //Enter LPM3 mode w/int enabled
__no_operation();
return 0;
}

//------------------------------------------------------------------------
#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector=RTC_C_VECTOR
__interrupt void RTC_ISR(void)
#elif defined(__GNUC__)

void __attribute__ ((interrupt(RTC_C_VECTOR))) RTC_ISR (void)
#else
#error Compiler not supported!
#endif
{
switch(__even_in_range(RTCIV, RTCIV__RT1PSIFG))
  {
  case RTCIV__NONE:     break;       //No interrupts
  case RTCIV__RTCOFIFG: break;       //RTCOFIFG
  case RTCIV__RTCRDYIFG:break;       //RTCRDYIFG
  case RTCIV__RTCTEVIFG:             //RTCEVIFG
                        P1OUT^= BIT0;//Toggle P1.0 LED
                        break;
  case RTCIV__RTCAIFG:   break;      //RTCAIFG
  case RTCIV__RT0PSIFG:  break;      //RT0PSIFG
  case RTCIV__RT1PSIFG:  break;      //RT1PSIFG
  default: break;
  }
}

//------------------------------------------------------------------------
```

Example. In this example the RTC is used in clock mode to trigger an interrupt every minute and second.

```
//******************************************************************************
// --COPYRIGHT--,BSD_EX
// Copyright (c) 2015, Texas Instruments Incorporated
// All rights reserved.
//
//                  MSP430 CODE EXAMPLE DISCLAIMER
//
//******************************************************************************
//MSP430FR5x9x Demo - RTC in real time clock mode
//
//Description: This program demonstrates the RTC mode by triggering an
//interrupt every second and minute. This code toggles P1.0 every
//second. This code recommends an external LFXT crystal for RTC
//accuracy.
//
//  ACLK = LFXT = 32768Hz, MCLK = SMCLK = default DCO = 1MHz
//
//               MSP430FR5994
//             -----------------
//        /|\ |              XIN|-
//         |  |                 | 32768Hz
//         ---|RST          XOUT|-
//            |                 |
//            |            P1.0 |--> Toggles every second
//            |                 |
//
//William Goh, Texas Instruments Inc., October 2015
//Built with IAR Embedded Workbench V6.30 & Code Composer Studio V6.1
//******************************************************************************

#include <msp430.h>

int main(void)
{
WDTCTL = WDTPW | WDTHOLD;               //Stop Watchdog timer
P1DIR |= BIT0;                          //Set P1.0 as output
PJSEL0 = BIT4 | BIT5;                   //Initialize LFXT pins

//Disable the GPIO power-on default high-impedance mode to activate
```

```
//previously configured port settings
PM5CTL0 &= ~LOCKLPM5;

                                        //Configure LFXT 32kHz crystal
CSCTL0_H = CSKEY_H;                     //Unlock CS registers
CSCTL4 &= ~LFXTOFF;                     //Enable LFXT
do
  {
  CSCTL5 &= ~LFXTOFFG;                  //Clear LFXT fault flag
  SFRIFG1 &= ~OFIFG;
  }while (SFRIFG1 & OFIFG);             //Test oscillator fault flag
CSCTL0_H = 0;                           //Lock CS registers
                                        //Configure RTC_C
RTCCTL0_H = RTCKEY_H;                   //Unlock RTC
RTCCTL0_L = RTCTEVIE_L | RTCRDYIE_L;    //enable RTC read ready int
                                        //enable RTC time event int
RTCCTL13 = RTCBCD | RTCHOLD | RTCMODE;  //RTC enable, BCD mode, RTC hold

RTCYEAR = 0x2019;                       //Year = 0x2019
RTCMON = 0x4;                           //Month = 0x04 = April
RTCDAY = 0x05;                          //Day = 0x05 = 5th
RTCDOW = 0x01;                          //Day of week = 0x01 = Monday
RTCHOUR = 0x10;                         //Hour = 0x10
RTCMIN = 0x32;                          //Minute = 0x32
RTCSEC = 0x45;                          //Seconds = 0x45
RTCADOWDAY = 0x2;                       //RTC Day of week alarm = 0x2
RTCADAY = 0x20;                         //RTC Day Alarm = 0x20
RTCAHOUR = 0x10;                        //RTC Hour Alarm
RTCAMIN = 0x23;                         //RTC Minute Alarm
RTCCTL13 &= ~(RTCHOLD);                 //Start RTC

__bis_SR_register(LPM3_bits | GIE);     //Enter LPM3 mode w/int enabled
__no_operation();
return 0;
}

//****************************************************************
#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector=RTC_C_VECTOR
```

```
__interrupt void RTC_ISR(void)
#elif defined(__GNUC__)

void __attribute__ ((interrupt(RTC_C_VECTOR))) RTC_ISR (void)
#else
#error Compiler not supported!
#endif
{
switch(__even_in_range(RTCIV, RTCIV__RT1PSIFG))
  {
  case RTCIV__NONE:      break;       //No interrupts
  case RTCIV__RTCOFIFG:  break;       //RTCOFIFG
  case RTCIV__RTCRDYIFG:              //RTCRDYIFG
                         P1OUT^=0x01; //Toggles P1.0 every second
                         break;
  case RTCIV__RTCTEVIFG:              //RTCEVIFG
                         __no_operation();// Int every minute - SET
                                     //BREAKPOINT HERE
                         break;
  case RTCIV__RTCAIFG:   break;       //RTCAIFG
  case RTCIV__RT0PSIFG:  break;       //RT0PSIFG
  case RTCIV__RT1PSIFG:  break;       //RT1PSIFG
  default: break;
  }
}
//*****************************************************************
```

7.9 INPUT CAPTURE AND OUTPUT COMPARE FEATURES

In this section, we study the input capture and output compare features of the MSP430 microcontroller. We begin with background information and the associated theory, followed by MSP430 specific timer information and application examples.

7.9.1 TIMING SYSTEM OVERVIEW AND BACKGROUND THEORY

As we have learned earlier in this chapter, the heart of the timing system is the time base. The time base frequency of an oscillating signal is used to generate a baseline clock signal. For a timer system, the system clock is used to update the contents of a special register called a free running

counter. The job of a free running counter is to count up (increment) each time it receives a rising edge (or a falling edge) of a clock signal. Thus, if a clock is running at the rate of 2 MHz, the free running counter will count up at every 0.5 μs. All other timer-related units reference the contents of the free running counter to perform I/O time-related activities: measurement of time periods, capture of timing events, and generation of time-related signals.

For input time-related activities, all microcontrollers typically have timer hardware components that detect signal logic changes on one or more input pins. Such components rely on a free running counter to capture external event times. We can use such ability to measure the period of an incoming signal, the width of a pulse, and the time of a signal logic change.

You can also use the timer input system to measure the pulse width of an aperiodic signal. For example, suppose that the times for the rising edge and the falling edge of an incoming signal are 1.5 s and 1.6 s, respectively. We can use these values to easily compute the pulse width of 0.1 s.

The second overall goal of the timer system is to generate signals to control external devices. Again, an event simply means a change of logic states on an output pin of a microcontroller at a specified time. Now consider Figure 7.8. Suppose an external device connected to the microcontroller requires a pulse signal to turn itself on. Suppose the particular pulse the external device needs is 2 millisecond wide. In such situations, we can use the free running counter value to synchronize the time of desired logic state changes. Naturally, extending the same capability, we can also generate a periodic pulse with a fixed duty cycle or a varying duty cycle.

For output timer functions, a microcontroller uses a comparator, a free running counter, logic switches, and special purpose registers to generate time-related signals on one or more output pins. A comparator checks the value of the free running counter for a match with the contents of another special purpose register where a programmer stores a specified time in terms of the free running counter value. The checking process is executed at each clock cycle and when a match occurs, the corresponding hardware system induces a programmed logic change on a programmed output port pin. Using such capability, one can generate a simple logic change at a designated time incident: a pulse with a desired time width or a pulse width modulated signal to control servo or direct current (DC) motors.

From the examples we discussed above, you may have wondered how a microcontroller can compute absolute times from the relative free running counter values, say 1.5 s and 1.6 s. The simple answer is that we cannot do so directly. A programmer must use the relative system clock values and derive the absolute time values. Suppose your microcontroller is clocked by a 2 MHz signal and the system clock uses a 16-bit free running counter. For such a system, each clock period represents 0.5 μs, and it takes approximately 32.78 ms to count from 0 to 2^{16} (65,536). The timer input system then uses the clock values to compute frequencies, periods, and pulse widths. Again, suppose you want to measure a pulse width of an incoming aperiodic signal. If the rising edge and the falling edge occurred at count values \$0010 and \$0114,[2] can

[2]The \$ symbol represents that the following value is in a hexadecimal form.

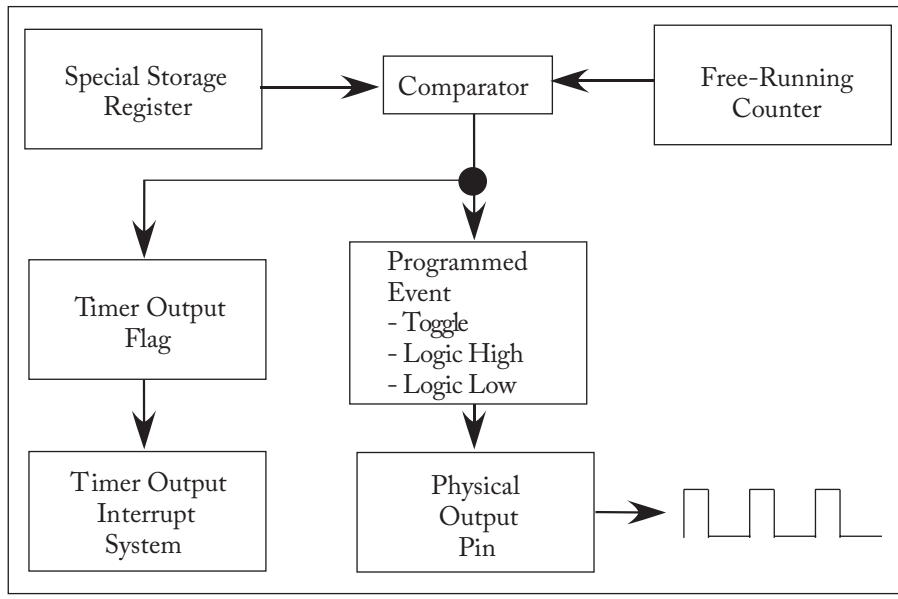


Figure 7.8: A diagram of a timer output system.

you find the pulse width when the free running counter is counting at 2 MHz? You will need to follow through the process similar to the one described next. We first need to convert the two values into their corresponding decimal values, 16 and 276. The pulse width of the signal in the number of counter value is 260. Since we already know how long it takes for the system to count one, we can readily compute the pulse width as $260 \times 0.5\ \mu s = 130\ \mu s$.

Our calculations do not take into account time increments lasting longer than the rollover time of the counter. When a counter rolls over from its maximum value back to zero, a flag is set to notify the processor of this event. In such cases, the rollover incidents are incorported to correctly determine the overall elapsed time of an event.

Elapsed time may be calculated using the following:

$$elapsed\ clock\ ticks = (n \times 2^b) + (stop\ count - start\ count)[clock\ ticks]$$

$$elapsed\ time = (elapsed\ clock\ ticks) \times (FRC\ clock\ period)\ [seconds].$$

In this first equation, "n" is the number of timer overflows that occur between the start and stop of an event, and "b" is the number of bits in the timer counter. The equation yields the elapsed time in clock ticks. To convert it to seconds, the number of clock ticks are multiplied by the period of the clock source of the free running counter.

7.9.2 APPLICATIONS

In this section, we consider important uses of the timer system of a microcontroller to (1) measure an input signal timing event (input capture), (2) to count the number of external signal occurrences (input capture), and (3) to generate timed signals (output compare). The specific implementation details are presented in Section 7.9.2. We present the overall applications in this section, starting with a case of measuring the time duration of an incoming signal.

Input Capture—Measuring External Timing Event

In many applications, we are interested in measuring the elapsed time or the frequency of an external event using a microcontroller. Using the hardware and functional units discussed in the previous sections, we now present a procedure to accomplish the task of computing the frequency of an incoming periodic signal. Suppose that we are interested in calculating the time features of the signal shown in Figure 7.9, an incoming periodic signal to the microcontroller.

The first necessary step for the current task is to turn on the timer system. As discussed, to reduce power consumption, a microcontroller usually does not turn on all of its functional systems after reset until they are needed. In addition to a separate timer module, many microcontroller manufacturers allow a programmer to choose the rate of a separate timer clock that governs the overall functions of a timer module.

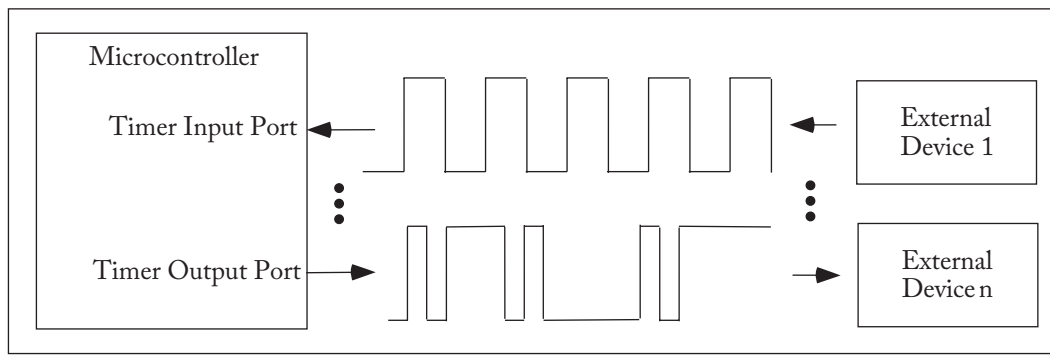


Figure 7.9: Use of the timer input and output systems of a microcontroller. The signal on top is fed into a timer input port. The captured signal is subsequently used to compute the input signal frequency. The signal on the bottom is generated using the timer output system. The signal is used to control an external device.

Once the timer is turned on and the clock rate is selected, a programmer must configure the physical port to which the incoming signal arrives. This step is done using a special input timer port configuration register. The next step is to configure the timer system to capture the intended input event. In the current example, we design our system to capture two consecutive rising edges or falling edges of an incoming signal. Again, the programming portion is done by storing an appropriate setup value to a special register.

Assuming that the input timer system is configured appropriately, you now have two options to accomplish the desired task. The first one is the use of a polling technique; the microcontroller continuously polls a flag, which holds a logic high signal when a programmed event occurs on the physical pin. Once the microcontroller detects the flag, it needs to clear the flag and record the time when the flag was set using another special register that captures the time of the associated free running counter value (see Section 7.10.1). The program needs to continue to wait for the next flag which indicates the end of one period of the incoming signal. A program then needs to record the newly acquired captured time represented in the form of a free running counter value again. The period of the signal can now be computed by calculating the time difference between the two captured event times, and, based on the clock speed of the microcontroller, the programmer can compute the actual time changes and consequently the frequency of the signal.

In many cases, a microcontroller can't afford the time or resources to poll for any one event. Such a situation calls for the second method: interrupt systems. Most microcontroller manufacturers have developed built-in interrupt systems with their timer system. In an interrupt system, instead of continuously polling for a flag, a microcontroller performs other tasks while relying on its interrupt system to detect a programmed event. The task of computing the period and the frequency is the same as the polling technique, except that the microcontroller will not be tied down to constantly checking the flag, increasing the efficient use of the microcontroller resources. To use interrupt systems, of course, we must pay the price by appropriately configuring interrupt systems to be triggered when a desired event is detected. Typically, additional registers must be configured, and a special program called an interrupt service routine must be written.

Suppose that for the input capture scenario of the current interest, the captured times for the two rising edges are $1000 and $5000, respectively. Note that these values are not absolute times but the representations of times reflected as the values of the free running counter. The period of the signal is $4000 or 16384 in decimal. Also, no timer overflows have been detected. If we assume that the timer clock runs at 10 MHz, the period of the signal is 1.6384 ms, and the corresponding frequency of the signal is approximately 610.35 Hz.

Counting Events

The same capability of measuring the period of a signal can also be used to simply count external events. Suppose we want to count the number of logic state changes of an incoming signal for a given period of time, as it may contain valuable information. Again, we can use the polling

technique or the interrupt technique to accomplish the task. For both techniques, the initial steps of turning on a timer and configuring a physical input port pin are the same. In this application, however, the programmed event should be any logic state changes instead of looking for a rising or a falling edge as we have done in the previous scenario. If the polling technique is used, at each event detection, the corresponding flag must be cleared and a counter must be updated. If the interrupt technique is used, one must write an interrupt service routine within which the flag is cleared and a counter is updated.

Output Compare—Generating Timing Signals to Interface External Devices

In the previous two sections, we considered two applications of capturing external incoming signals. In this section and the next one, we consider how a microcontroller can generate time critical signals for use by external devices. Suppose in this application, we want to send a signal shown in Figure 7.9 to turn on an external device. The timing signal is arbitrary, but the application will show that a timer output system can generate any desired time-related signals, permitted under the timer clock speed limit of the microcontroller.

Similar to the use of the timer input system, one must first turn on the timer system and configure a physical pin as a timer output pin using special registers. In addition, one also needs to program the desired external event using a special register associated with the timer output system. To generate the signal shown in Figure 7.9, one must compute the time required between the rising and the falling edges. Suppose also that the external device requires a pulse which is 2 ms wide to be activated. To generate the desired pulse, one must first program the logic state for the particular pin to be low and set the time value using a special register with respect to the contents of the free running counter. As was previously mentioned, at each clock cycle, the special register contents are compared with the contents of the free running counter, and when a match occurs, the programmed logic state appears on the designated hardware pin. Once the rising edge is generated, the program then must reconfigure the event to be a falling edge (logic state low) and change the contents of the special register to be compared with the free running counter. For the particular example in Figure 7.9, let's assume that the main clock runs at 2 MHz, the free running counter is a 16-bit counter, and the name of the special register (16-bit register) where we can put appropriate values is output timer register. To generate the desired pulse, we can put $0000 first to the output timer register, and after the rising edge has been generated, we need to change the program event to a falling edge and put $0FA0 or 4000 in decimal to the output timer register. As was the case with the input timer system module, we can use output timer system interrupts to generate the desired signals as well.

7.10 MSP430 TIMERS: TIMER_A AND TIMER_B

All MSP430 microcontrollers have both Timer_A and Timer_B I/O ports that can be used to capture external signal events and generate time-related signals for external devices. The captured external signal events include time stamped logic state changes, the frequency of a periodic sig-

nal, a width of a pulse to name a few. The time-related output signals range from a simple change of logic levels on an output pin at a designated time to generation of PWM signals. We present both input capture and output compare subsystem capabilities in this section. Both Timer_A and Timer_B systems can be configured to function as capture and compare input/output ports. The Timer_A system is present in all MSP430 controllers while Timer_B, with more advanced capture and compare capabilities, is found in higher-end MSP430 family members. Much of the discussion on Timer_A applies to Timer_B. The MSP430FR2433 and the MSP430FR5994 are both equipped with a complement of Timer_A and Timer_B timers.

The block diagram for Timer_A is shown in Figure 7.10. The main timer feature is a 16-bit configurable up/down timer (TAxR). The timer may be clocked from a variety of sources including the TAxCLK, ACLK, and the SMCLK. The clock source may be reduced by a series dividers (ID, IDEX). Timer_A (and B) may be used for multiple captures and compares, pulse width modulation, interval timing, and timer-based interrupts. Timer features are configured using the Timer_A Control Register (TAxCTL) shown in Figure 7.11. As shown in Figure 7.11, Timer_A (and B) may be configured for up, continuous, and up/down modes.

7.10.1 MSP430 FREE RUNNING COUNTER

In the Timer_A system, there are two to three different I/O subsystems (channels) that can be configured independently. For the Timer_B system, the number of I/O channels vary from three to seven. Since each channel for Timer_A and Timer_B systems has the identical hardware and functional capabilities, we present only a single channel of the Timer_A system. The source of all timer subsystems, whether they are used to capture input signal characteristics or to generate output time-related signals, is a free running 16-bit counter, the TAxR register and TBxR register. The counter counts up (can count down for some applications) at a specified interval, determined by the clock source used and a pre-scalar factor, and works as the universal timer for all time-related events. Figure 7.12 shows the free running counter, TAxR, and the features that govern its operation.

The programming of the counter is done with the help of the Timer_A control register (TACTL). Figure 7.13 shows the contents of the 16 bit register. Referring to Figures 7.12 and 7.13 together, one can use bits 9 and 8 (TASSELx) of the TACTL register to choose the clock used for the free running counter as follows:

- 00 – TACTL (external clock)
- 01 – ACLK (internal “slow” clock)
- 10 – SMCLK (internal “fast” clock)
- 11 – INCLK (external clock)

The Input Divider (IDx) bits (bits 7 and 6 in the TACTL register) are used to scale the clock source before the free running counter updates itself. The pre-scale factors are

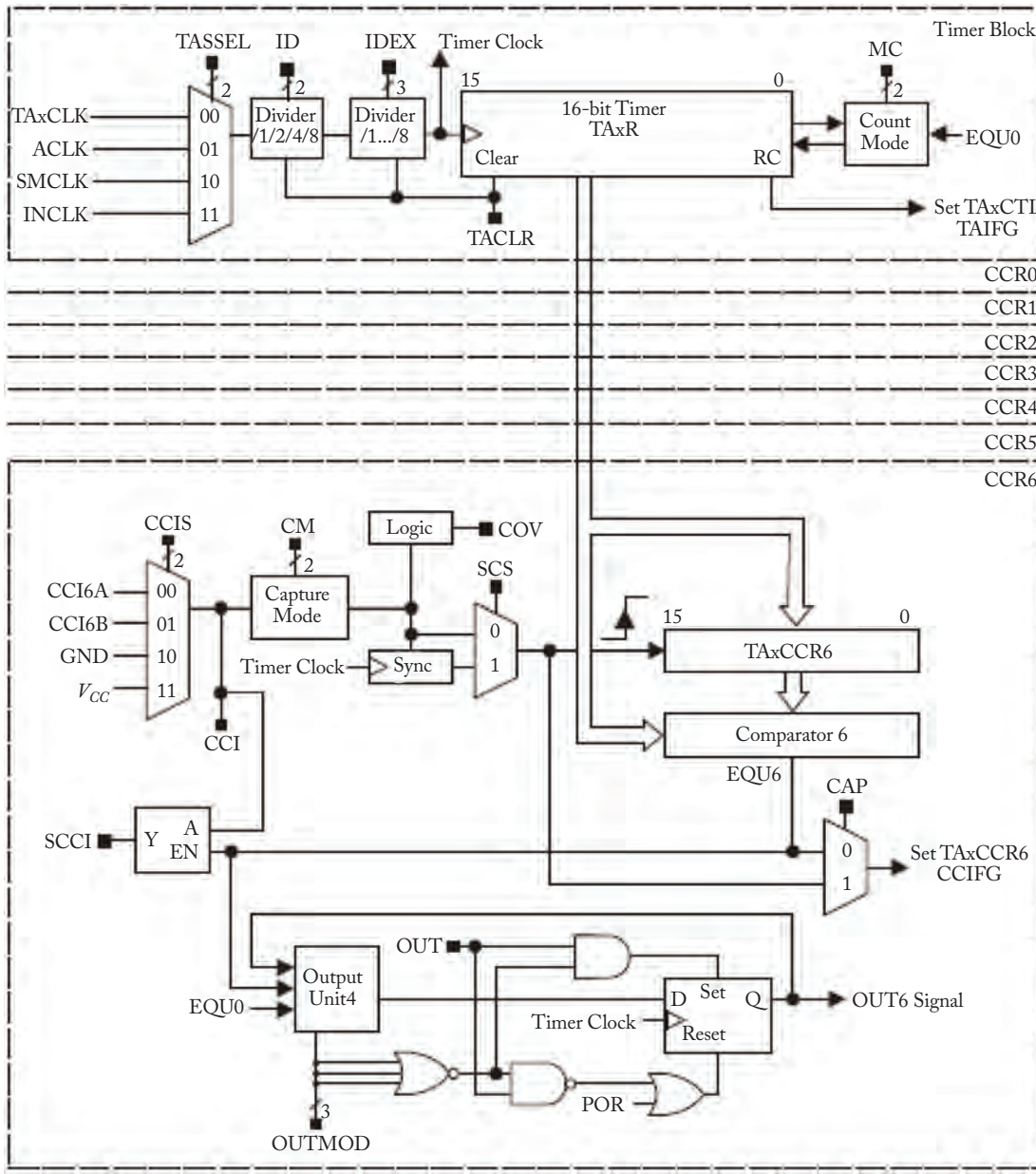


Figure 7.10: Timer_A block diagram [SLAU445G, 2016, SLAU367O, 2017]. (Illustration used with permission of Texas Instruments (www.ti.com).)

15	14	13	12	11	10	9	8
Reserved						TASSEL	
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)
7	6	5	4	3	2	1	0
ID		MC		Reserved	TACLR	TAIE	TAIFG
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	w-(0)	rw-(0)	rw-(0)

Timer_A Control Register (TAxCTL)

TAxCTL[9:8]: TASSEL: Timer_A clock source: 00 = TAxCLK, 01 = ACLK, 10 = SMCLK, 11 = INCLK

TAxCTL[7:6]: ID: input divider: 00 = 1, 01 = 2, 10 = 4, 11 = 8

TAxCTL[5:4]: MC: mode control: 00 = stop, 01 = up, 10 = continuous, 11 = up/down

TAxCTL[2]: TACLR: Timer_A clear: 1 = resets TAxR register

TAxCTL[1]: TAIE: Timer_A interrupt enable: 0 = disabled, 1 = enabled

TAxCTL[0]: TAIFG: Timer_A interrupt flag: 0 = none, 1 = pending

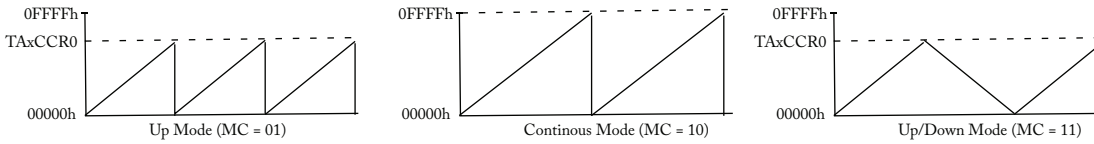


Figure 7.11: Timer_A registers [SLAU445G, 2016, SLAU367O, 2017]. (Illustration used with permission of Texas Instruments (www.ti.com).)

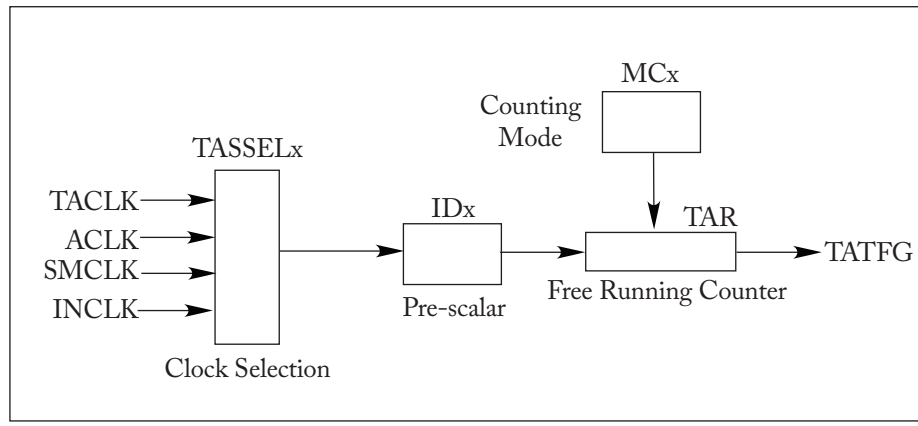


Figure 7.12: 16-bit free running counter.

Timer_A Control Register (TACTL)

15	14	13	12	11	10	9	8
Unused						TASSELx	
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)

7	6	5	4	3	2	1	0
IDx		MCx		Unused	TACLR	TAIE	TAIFG
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	w-(0)	rw-(0)	rw-(0)

Figure 7.13: Timer_A control register.

- 00 – increase (decrease) the TAR counter by one at each (rising/falling edge) clock cycle
- 01 – increase (decrease) the TAR counter by one every two clock cycles
- 10 – increase (decrease) the TAR counter by one every four clock cycles
- 11 – increase (decrease) the TAR counter by one every eight clock cycles

Using the pre-scalar factors, one can slow down the frequency of the free running counter by factor of 1, 2, 4, or 8, respectively. The mode control (MCx) bits (bits 5 and 4) of the TACTL register govern how the counter operates as follows.

- 00 – stop counter
- 01 – count up from 0 to a value stored in the Timer_A Capture/Compare 0 (TACCR0) register
- 10 – count from 0 to 2^{16} (0x0000 to 0xFFFF)
- 11 – count up/down: count from 0 to the value in the TACCR0 register, then count backward to 0. Repeat the process.

Thus, one can setup the counter to operate in one of the four operating modes. The Up mode (MCx bits: 01) increments the TAR value by one until it reaches the value stored in the TACCR0 register. When the value in the TAR register changes from the value in the TACCR0 register - 1 to the value in the TACCR0 register, the TACCRO capture/compare interrupt flag (CCIFG) bit is set, and when the value in the TAR changes from the value in TACCRO to zero during the next clock cycle, the Timer_A Interrupt Flag (TAIFG) flag in the TACTL register is set.

When operating in the continuous mode (MCx bits: 10), the TAIFG flag is set when the value in TAR register changes from 0xFFFFh to 0x0000h. In the Up/Down mode, the

TACCRO CCIFG flag is set when the timer TAR value changes from the value stored in the TACCR0 register -1 to the value stored in the TACCR0 register (Up), and the TAIFG flag is set when the timer TAR value changes from 0x01h to 0x00h (Down). Setting the TACLT bit (bit 2) of the Timer_A Control Register (TACTL) register clears the TAR counter, resets the divider, and changes the direction of the Up/Down counter, if the MCx bits are both set. Each time the free running counter reaches its limit states, the TAIFG bit (bit 0) of the TACTL is set, where the limit state is defined as follows.

- MCx: 00 – no changes to TAIFG
- MCx: 01 – TAR value changes from the value in TACCR0 to 0
- MCx: 10 – TAR value changes from 0xFFFF to 0x0000
- MCx: 11 – no changes to TAIFG

Now that we understand how to configure the counter, TAR, that is used as the basis for all MSP430 Timer_A and Timer_B activities, we now present three different capabilities: input capture, output compare, and a variation of the output compare—pulse width modulation. A similar, a parallel discussion for Timer_B for the current and next sections can be derived using Timer_B registers. For example, the explanation of the free running counter register, TAxR, can be replaced with the one for the Timer_B system and its free running counter, TBxR.

7.10.2 INPUT CAPTURE

As mentioned earlier, the purpose of capture functions of the MSP430 microcontroller is to capture the time of incoming external signal events. Again, all discussion pertaining to Timer_A applies to Timer_B with appropriate register changes. In MSP430FR5994 microcontroller, for each timer system, it contains seven channels that can be configured either as an input capture or an output compare channel with associated interrupt sub-systems.

Suppose you want to capture the time period between two consecutive events, say customers entering your retail store. You must have some means to record the time of the first event and the second event. The input capture system in Timer_A and Timer_B systems give us those capabilities, which we present in this section. Figure 7.14 shows components that make up an input capture channel, and Figure 7.15 shows the contents of the control register to configure the capture system. For the rest of the discussion, refer to both Figures 7.14 and 7.15.
The capture mode (CMx) bits (bits 15 and 14) of the TAxCCTLn register determine the capture events as shown below.

- 00 – no capture
- 01 – capture rising edge
- 10 – capture falling edge

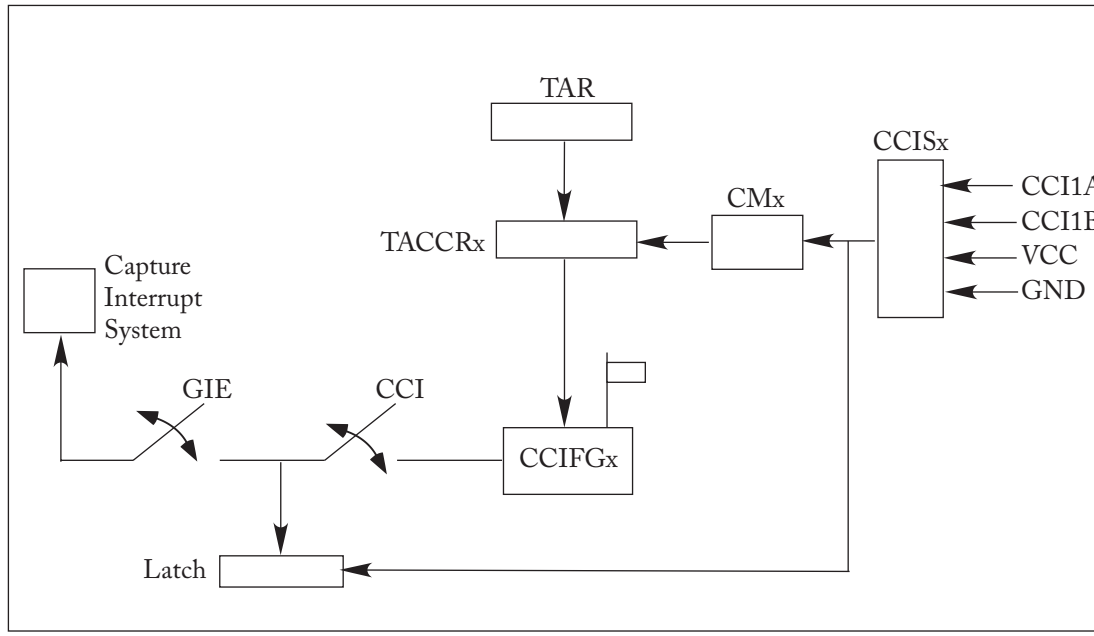


Figure 7.14: Input capture system diagram.

Timer_A Capture Compare Control Register 0, 1, 2 (TACCTL0, 1, 2)

15	14	13	12	11	10	9	8
CMX1	CMXO	CCIS1	CCIS0	SCS	SSCI	Unused	CAP
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-0	r	r0	rw-(0)

7	6	5	4	3	2	1	0
OUTMOD2	OUTMOD1	OUTMOD0	CCIE	CCI	OUT	COV	CCIFG
rw-(0)	rw-(0)	rw-(0)	rw-(0)	r	rw-(0)	rw-(0)	rw-(0)

Figure 7.15: Input capture and output compare control register.

- 11 – capture both edges

The capture/compare input select (CCISx) bits (bits 13 and 12) are used to select the input signal to be captured as follows.

- 00 – capture compare input port A CCIxA
- 01 – capture compare input port B CCIxB
- 10 – Ground
- 11 – Supply voltage Vcc

The Capture Mode (CAP) bit (bit 8) of the Timer_A Capture Control Register (TAxCCTLn) register is used to configure channel n to be either as an input capture channel (1) or as an output compare channel (0). When the event designated by CMx bits appears on the input channel pin, the current value of free running counter TAxR is captured in the Timer_A Capture/Compare Register n (TAxCCRn) register and the corresponding flag, Capture/Compare Interrupt Flag (CCIFG) (bit 0), in the TAxCCTLn register is set. If the capture/compare interrupt enable (CCIE) bit (bit 4) is set and the GIE bit is activated, the interrupt system is configured to service the interrupt. If another input capture event occurs before the TAxCCRn is read, the capture overflow (COV) bit (bit 1) of the TAxCCTLn turns to 1 (set).

Example: In this input capture example, we use the MSP430FR5994 controller to capture the VLO clock signal. The program captures rising edge of the clock signal and stores the free running counter values in memory. When 20 rising edges are captured the logic state on pin P1.0 changes.

```
//******************************************************************
// --COPYRIGHT--,BSD_EX
// Copyright (c) 2015, Texas Instruments Incorporated
// All rights reserved.
//
//                  MSP430 CODE EXAMPLE DISCLAIMER
//
//******************************************************************
//MSP430FR5x9x Demo - Timer0_A3 Capture of VLO Period using DCO SMCLK
//
//Description; Capture a number of periods of the VLO clock and store
//them in an array.  When the set number of periods is captured the
//program is trapped and the LED on P1.0 is toggled. At this point
//halt the program execution read out the values using the debugger.
//
```

```
//  ACLK = VLOCLK = 9.4kHz (typ.),
//  MCLK = SMCLK = default DCO / default divider = 1MHz
//
//                MSP430FR5994
//             -----------------
//         /|\|              XIN|-
//          | |                 |
//          --|RST          XOUT|-
//            |                 |
//            |             P1.0|-->LED
//
//William Goh, Texas Instruments, Inc, October 2015
//Built with IAR Embedded Workbench V6.30 & Code Composer Studio V6.1
//******************************************************************

#include <msp430.h>

#define NUMBER_TIMER_CAPTURES       20

volatile unsigned int timerAcaptureValues[NUMBER_TIMER_CAPTURES];
unsigned int timerAcapturePointer = 0;

int main(void)
{
WDTCTL = WDTPW | WDTHOLD;               //Stop watchdog timer
                                        //Configure GPIO
P1OUT &= ~0x01;                         //Clear P1.0 output
P1DIR |= 0x01;                          //Set P1.0 to output direction

//Disable the GPIO power-on default high-impedance mode to activate
//previously configured port settings
PM5CTL0 &= ~LOCKLPM5;
                                        //Clock System Setup
CSCTL0_H = CSKEY_H;                     //Unlock CS registers
CSCTL2 &= ~SELA_7;
CSCTL2 |= SELA__VLOCLK;                 //Select ACLK=VLOCLK
CSCTL0_H = 0x00;                        //Lock CS module
                                        //use byte mode to upper byte
__delay_cycles(1000);                   //Allow clock system to settle
```

```
                                        //Timer0_A3 Setup
TA0CCTL2 = CM_1 | CCIS_1 | SCS | CAP | CCIE;
                                        //Capture rising edge,
                                        //Use CCI2B=ACLK,
                                        //Synchronous capture,
                                        //Enable capture mode,
                                        //Enable capture interrupt
TA0CTL = TASSEL__SMCLK | MC__CONTINUOUS;//Use SMCLK as clock source,
                                        //Timer in continuous mode
__bis_SR_register(LPM0_bits | GIE);
__no_operation();
}

//------------------------------------------------------------------------
// Timer0_A3 CC1-4, TA Interrupt Handler
//------------------------------------------------------------------------

#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector = TIMER0_A1_VECTOR
__interrupt void Timer0_A1_ISR(void)
#elif defined(__GNUC__)

void __attribute__ ((interrupt(TIMER0_A1_VECTOR))) Timer0_A1_ISR (void)
#else
#error Compiler not supported!
#endif
{
switch (__even_in_range(TA0IV, TAIV__TAIFG))
  {
  case TAIV__TACCR1: break;
  case TAIV__TACCR2:
                timerAcaptureValues[timerAcapturePointer++] = TA0CCR2;
                if(timerAcapturePointer >= 20)
                  {
                  while(1)
                    {
                    P1OUT ^= 0x01;      //Toggle P1.0 (LED)
                    __delay_cycles(100000);
                    }
```

```
                    }
                  break;
  case TAIV__TAIFG: break;
  default: break;
  }
}

//*******************************************************************
```

7.10.3 OUTPUT COMPARE

In this section, we present the MSP430 function opposite to the input capture capabilities. The output compare function is designed to generate desired time critical signals on an output pin. To do so, MSP430 architects designed the compare channels using almost the same registers used in the input capture systems previously described. Refer to Figures 7.16 and 7.15 for the following discussion. At each clock cycle, the comparator compares the current value in the TAR with the one previously stored in the TAxCCRn register. When the two values match (identical), the output logic state (either logic high or logic low) will appear on the output pin based on the

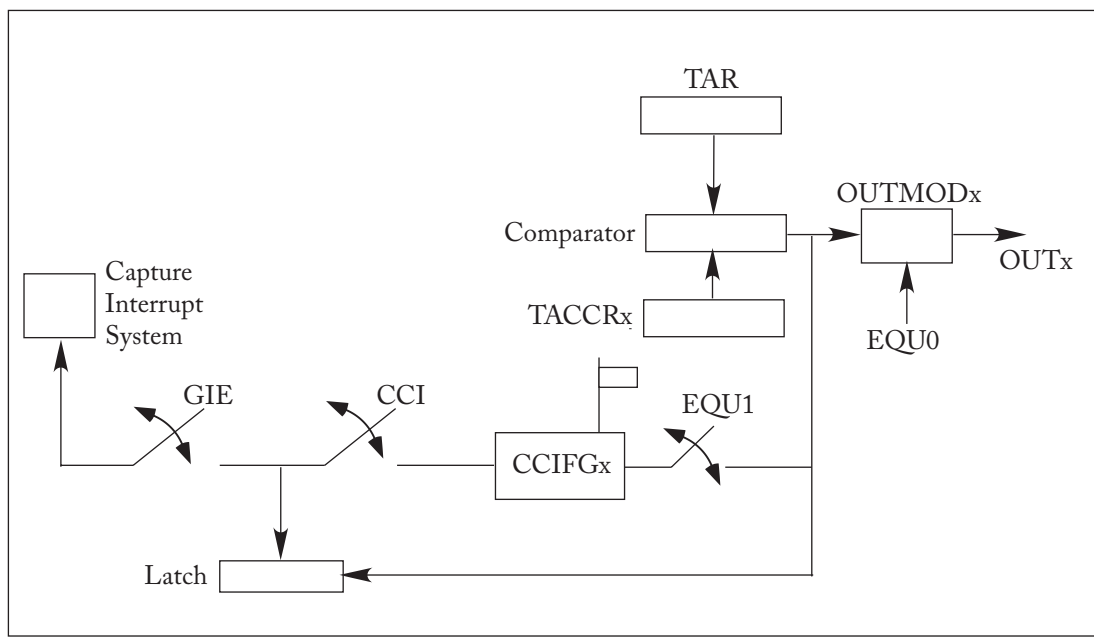


Figure 7.16: Output compare diagram.

programmed states of OUTMODx bits (bits 7, 6, and 5) of the TAxCCTLn register as shown below.

- 000 – output logic is controlled by the OUT bit (bit 2)
- 001 – set the logic on the output pin high (continuous counting mode)
- 010 – toggle the logic state on the output pin (Up/Down counting mode)
- 011 – set or resets the logic state on the output pin (Up counting mode)
- 100 – toggle the logic state on the output pin (continuous counting mode)
- 101 – reset (logic zero) the logic state on the output pin (continuous counting mode)
- 110 – toggle the logic state on the output pin (Up/Down counting mode)
- 111 – set or reset the logic state on the output pin (Up counting mode)

The EQU0 signal, shown in Figure 7.16, governs modes 010, 011, 110, and 111. When the comparison of values in TAxCCRn and TAxR results in a match, the CCIFG flag is also set (logic 1), and if the CCI bit along with the GIE bit is set, the corresponding output compare interrupt system is enabled. One can also use more than one channel to generate a periodic pulse. By directing the output signal onto a single output pin and configuring two channels, say channel 0 with TAxCCR0 and channel 1 with TARCCR1, appropriately, one can generate a periodic signal. For example, suppose we configure both channels to be output compare channels, output event to set and reset the output logic state. If we assume continuous counting mode from 0x0000 to 0xFFFF for the free running counter, by setting the TAxCCR0 value to be zero and the TAxCCR1 value to be 0x8000, we can generate a pulse width modulated signal with 50% duty cycle. One can also achieve the same output signal using a single channel, say channel 0, setting the output mode to toggle the logic states. Note that the frequency of the signal is half of the signal generated using two channels.

In the remainder of this section, we show how register TAxIV (Timer_Ax Interrupt Vector) is used to configure a desired interrupt service routine. The first column of Table 7.1 shows the values need to be loaded to the TAxIV register to program a particular interrupt system. The second column of the same table shows the corresponding interrupt source for the numerical values shown in the first column. When an interrupt occurs, the appropriate interrupt service routine must clear the associated flag of the interrupt, shown in the last column of the table.

Thus, Timer_A capture/compare 1–6 (TAxCCR1 - TAxCCR6) or timer overflow (TAxR) can cause interrupts in the Timer_A system. To have the corresponding interrupt to be enabled, one must set the capture/compare interrupt enable (CCIE) bit, or Timer_A Interrupt Enable (TAIE) bit in the Capture/Compare Control (TAxCCTLn) register or the Timer_A control (TAxCTL) register.

Table 7.1: How register TAxIV (Timer_Ax Interrupt Vector) is used to configure a desired interrupt service routine

Register Contents	Interrupt Source	Associated Flag
00h	No Interrupt	
02h	Channel 1	TAxCCR1 CCIFG
04h	Channel 2	TAxCCR2 CCIFG
06h	Channel 3	TAxCCR3 CCIFG
08h	Channel 4	TAxCCR4 CCIFG
0Ah	Channel 5	TAxCCR5 CCIFG
0Ch	Channel 6	TAxCCR6 CCIFG
0Eh	Timer Overflow	TAxCTL TAIFG

The code snapshot below shows how one can setup the Timer_A system for an interrupt to occur every fixed period. Lines 1 and 2 are directives to define the subroutine TA_wake. Line 3 of the program contains the label of the subroutine, and the following line of code determines the duration of the period using the TAxCCR1 register. The desired period should be calculated based on the clock speed, and the resulting numerical number should replace symbol num on line 4 of this program. Instructions on lines 5–7 enable the local interrupt, set up the counter to choose the ACLK clock and the continuous count mode, and turns on the global interrupt switch, respectively. The ret instruction on line 8 returns the program flow to the portion of the program that called the subroutine. Instructions on lines 9–13 show how one can write a corresponding interrupt service routine. The test instruction on line 10 clears the interrupt flag. Your program code that performs the desired task every time the interrupt occurs will go in the space designated by line 11. The add instruction on line 12 updates the TAxCCR1 register, designating the period for the next interrupt to occur by adding the same amount of time to the current time in TAxCCR1. This program can be used to wake-up the controller, periodically, to perform required tasks using the built-in Timer_A interrupt system.

```
;-------------------------------------------------------------------
1            .def TA_wake
2            .text
3  TA_wake
4            mov.w  #num, &TAxCCR1       ;use appropriate time number
5            mov.w  #CCIE, &TAxCCTL1     ;TACCR1 interrupt enabled
6            mov.w  #TASSEL_1+MC_2, &TAxCTL ;ACLK, continuous mode
7            bis.b  #GIE, SR             ;enable global interrupt
8            ret
```

```
9  TA_ISR
10           tst.w  &TAxIV              ;read clears flag
11                 :                    ;perform required task
12           add.w  #num, &TAxCCR1      ;update next interrupt time
13           reti
;-------------------------------------------------------------------
```

The second interrupt associated with theTimer_A system is the one related to the TAxCCR0 channel. When an TAxCCR0 interrupt is initiated, the CCIFG flag in the TAxCCTL0 register is set, and when the CCIE bit is set, the interrupt is serviced. The following programs, one in C and the other in assembly, show how one can configure an output pin to toggle its logic state using the Timer_A TAxCCRO0 interrupt vector.

```
;-------------------------------------------------------------------
              mov.w    #WDTPW+WDTHOLD,&WDTCTL   ;turn off WDT
              mov.b    #0x01, &P1DIR            ;program P1.0 as output
              mov.w    #0x1000, &TAxCCR0        ;initialize TAxCCR0 value
              mov.b    #TASSEL_1+MC_1,&TAxCTL   ;select ACLK, up mode
              mov.b    #CCIE, &TAxCCTL0         ;enable local interrupt
              mov.b    #GIE, &SR                ;enable global interrupt
Loop          bra      Loop                     ;wait

Timer_A0_ISR                                    ;ISR routine
              xorb     #Toggle, &P1OUT          ;toggle logic state
              rti

;-------------------------------------------------------------------
```

In C:

```
//*******************************************************************

void main(void)
{
WDTCTL = WDTPW + WDTHOLD;              //disable watchdog timer
P1DIR |= 0x01;                         //program P1.0 as output
TAxCCR0 = 0x1000;                      //initialize TACCR0 value
TAxCTL = TASSEL_1 + MC_1;              //select ACLK, count up mode
TAxCCTL0 = CCIE;                       //enable TAxCCR0 interrupt
SR = GIE;                              //turn on global interrupt switch
```

```
while(1)                                  //wait for interrupt to occur
  {
  ;
  }
}

//*******************************************************************
//Timer_A TAxCCR0 interrupt service routine
//*******************************************************************

Interrupt(TIMERA0_VECTOR) TimerA_procedure(void)
{
P1OUT ^= 0x01;                            //toggle logic state
}

//*******************************************************************
```

7.10.4 TIMER_B SYSTEM

The Timer_B system can be used as input capture and output compare timer units as we have done with the Timer_A system. It contains up to seven different subsystems (channels) that can be configured as capture or compare systems. The primary difference between Timer_A and Timer_B system is that an extra buffer is introduced in the Timer_B system along with a means to update the value of the register used to compare the free running timer value when a channel is used as an output compare system and to capture the free running counter value when configured as an input capture system.

Figure 7.17 shows the extra buffer used in Timer_B systems. Note that the free running counter is now called TBxR instead of TAxR as you saw in the previous section. The TAxCTL and TAxCCRn registers are replaced by 16 bit registers TBxCTL and TBxCCRn. When a Timer_B channel is configured as an input capture channel and a programmed event appears on the input pin, the free running counter value is captured in the TBxCCRn register as was the case in the Timer_A system, but you have an option to upload that value into another 16-bit register, TBxCLn. Why do you need this extra register? Suppose you have external events that occur very quickly and you need to capture both events. By loading the first event time and storing it quickly will allow the TBxCCRn register to be free to capture the second event. Similarly, when a channel is configured as an output compare channel, the extra register allows a programmer to generate output signals whose time values are separated by a "small1" number.[3]

[3]The small free running counter difference value is governed by the time required to upload a new value from TBxCCRn to TBxCLn.

Referring to Figure 7.17, note that the CCLDx bits (bits 10 and 9) of the TBxCCTLn register govern the time when the value from the TBxCCRn register is transferred to the TBxCLn register as shown below.

- 00 – immediate
- 01 – update when TBxR value is zero
- 10 – same as 01 for continuous up count mode. If Up/Down count mode is chosen, the transfer occurs either when TBxR = 0 or TBxCLn = TBxR
- 11 – update when TBxR = old TBxCLn

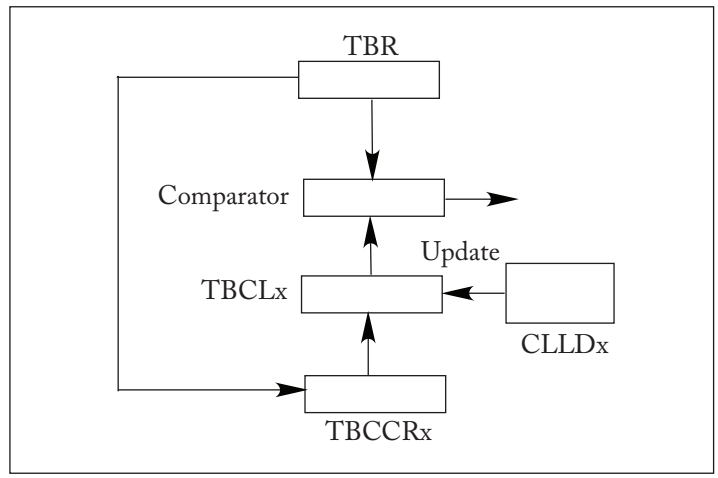


Figure 7.17: Timer_B additional components.

Before we leave this section, we show an example program that uses the capabilities of the Timer_A system of the MSP430FR2433 to generate two pulse-width modulated signals on pins P1.4 and P1.5. The signal out of P1.4 has 75% duty cycle and the signal out of P1.5 has 25% duty cycle. The program utilizes the counting up mode along with the TA1CCR0, TA1CCR1, and TA1CCR2 systems to generate the pulses. The duration specified by the contents of the TA1CCR0 register determine the pulse periods while the values in TA1CCR1 and TA1CCR2 registers determine the duty cycles for the two pulses. We assume that the ACLK clock is connected to the LFX1CLK clock signal generator with 32 kHz crystal.

```
//*******************************************************************
// --COPYRIGHT--,BSD_EX
// Copyright (c) 2015, Texas Instruments Incorporated
// All rights reserved.
```

```
//
//                  MSP430 CODE EXAMPLE DISCLAIMER
//
//*******************************************************************
//MSP430FR243x Demo - Timer1_A3, PWM TA1.1-2, Up Mode, DCO SMCLK
//
//Description: This program generates two PWM outputs on P1.4,P1.5
//using Timer1_A configured for up mode. The value in CCR0, 1000-1,
//defines the PWM period and the values in CCR1 and CCR2 the PWM duty
//cycles. Using ~1MHz SMCLK as TACLK, the timer period is ~1ms with
//a 75
//
//  ACLK = n/a, SMCLK = MCLK = TACLK = 1MHz
//
//           MSP430FR2433
//         ---------------
//     /|\|               |
//      | |               |
//      --|RST            |
//        |               |
//        |     P1.5/TA1.1|--> CCR1 - 75
//        |     P1.4/TA1.2|--> CCR2 - 25
//
//
//Ling Zhu, Texas Instruments Inc., Feb 2015
//Built with IAR Embedded Workbench v6.20 & Code Composer Studio v6.0.1
//*******************************************************************
#include <msp430.h>

int main(void)
{
WDTCTL = WDTPW | WDTHOLD;                 //Stop WDT
                                          //Configure GPIO
P1DIR |= BIT4 + BIT5;
P1SEL1 |= BIT4 + BIT5;

//Disable the GPIO power-on default high-impedance mode to activate
//previously configured port settings
PM5CTL0 &= ~LOCKLPM5;
```

```
TA1CCR0 = 1000-1;                          //PWM Period
TA1CCTL1 = OUTMOD_7;                       //CCR1 reset/set
TA1CCR1 = 750;                             //CCR1 PWM duty cycle
TA1CCTL2 = OUTMOD_7;                       //CCR2 reset/set
TA1CCR2 = 250;                             //CCR2 PWM duty cycle
TA1CTL = TASSEL__SMCLK | MC__UP | TACLR; //SMCLK, up mode, clear TAR
__bis_SR_register(LPM0_bits);              //Enter LPM0
__no_operation();                          //For debugger
}



//*******************************************************************
```

7.11 LABORATORY EXERCISE: GENERATION OF VARYING PULSE WIDTH MODULATED SIGNALS TO CONTROL DC MOTORS

Purpose: The purpose of this laboratory exercise is to program an MSP430 series controller to generate a pulse-width modulated signal/waveform with varying duty cycle to control the speed of a DC motor. The program requires you to use the input capture and output compare capabilities of the controller. To meet the requirements of this laboratory exercise, you must

- use the output compare system to modify the duty cycle of the output waveform,
- use the input capture system to monitor the number of pulses generated by the output compare system and adjust the duty cycle of the output waveform appropriately,
- change the duty cycle based on a desired DC motor speed profile,
- configure Port 1 pins, and
- verify the output waveform by connecting the output pin on Port 1 to an oscilloscope.

Documentation: User Manual of your MSP430 microcontroller board.

Prelab: For the prelab, complete a flowchart and pseudocode for your program.

Description: As we learned in this chapter, the timer system of the MSP430 series microcontroller is used for signal generation, measurement, and timing. In this lab, you are asked to configure an MSP430 series controller to generate desired output waveforms using its output

compare system. The contents of the output compare system registers are programmed to set and clear the logic states of an output pin and to cause an interrupt, related to the output pin.

The output event occurs when the free running counter (TAxR) value matches the value stored in the designated output compare register (TAxCCRn). By adjusting the value in the TAxCCRn register, one can program MSP430 to change the time when an output event occurs. The input capture system is used to monitor the incoming signal. In this laboratory exercise, it is used to count the number of pulses being generated by the output compare pin. By counting the number of pulses, the input capture system can keep track of the time the output waveform is generated and modify the duty cycle accordingly. The physical pins used in this laboratory are P1.1 and P1.2, where P1.1 will be used as the input capture pin and the P1.2 pin will be used as the output compare pin.

Tasks: In this lab, you need to write a program to modify the duty cycle of an output periodic signal in real time. The desired speed profile (velocity vs. time) is shown in Figure 7.18. The y-

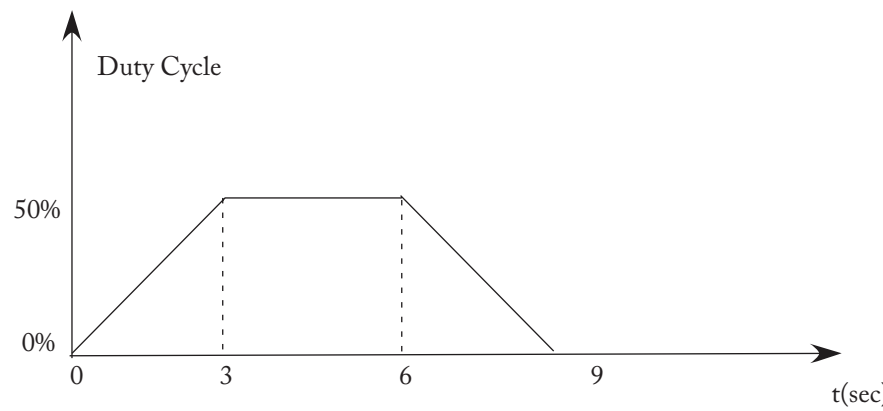


Figure 7.18: Desired speed profile for the DC motor.

axis shows the desired duty cycle, and the x-axis shows the time duration for the entire profile. The duty cycle should increase from 0–50% during the first three seconds, maintain the 50% duty cycle for another 3 s, and decrease to 0% duty cycle during the last three second period. The waveform period should be adjusted to 20 ms. (This is the time for the TA1R register to count from 0–65,536.). You must generate approximately 457 total 20 ms pulses for the total 9 s. The duty cycle should increase linearly from 0–50% in 3 s, or in 152 pulses. Using P1.2 pin as the output pin, the TA1CCR2 register value should change from 0000–8000h in 152 pulses or by adding approximately 216 counts to the TA1CCR2 register at each pulse.

Procedure: To generate the desired pulse width modulated signal, use P1.2 pin as the output compare pin and P1.1 as the input capture pin.

- Turn on the Timer System.
- Turn off the Watchdog timer.
- Generate the pulse width modulated signal.
  - Set the logic states on the output pin to be low (off) when successful compares are made by configuring bits in the TA1CCTL2 register.
  - Use the TA1CCR1 register to set logic state of the pulse to be high (on) using the TA1CCTL1 register.
- Set up the duty cycle.
  - We are using the square wave period of 20 ms. We can find a corresponding number that represents a desired duty cycle. For example, 8000h represents 50% duty cycle. When 8000h is stored in the TA1CCR2 register and the TA1R register value matches with the one in TA1CCR2, the designated action (logic off) takes place on the output pin.
  - During the acceleration period, you must add 216 to the current value in TA1CCR2, which will increase the duty cycle at each of the 152 changes between time 0–3 s, assuming that the TT1CCR2 value started with 0000h. During the deceleration period, the opposite action must occur. At each pulse, the value in TA1CCR2 is decremented by 216 counts. At the end of the three second period of deceleration, the duty cycle decreases to zero.
- Measuring the number of pulses arriving at an input capture pin.
  - Use the input capture system P1.1 to monitor the incoming pulses. You should connect P1.2 to P1.1, which feeds the output compare signal back to input capture pin. Use the TA1CCTL1 register to configure the input capture system to capture each pulse entering. By counting the pulses, we can keep tract of the current time with respect to the desired time profile. Thus, during the first 152 pulses, the input capture system interrupt should be the one who modifies the TA1CCR2 register contents. During the next 152 pulses, no changes should be made to the TA1CCR2 register, and during the last 152 pulses, the value in TA1CCR2 should be decreased by 216 counts after each pulse arrives on the P1.1 pin using the input capture interrupt.
- Once the controller is configured with the steps shown above, the P1.2 pin should be connected to an oscilloscope, and the output waveform with varying duty cycle should be verified.

7.12 SUMMARY

In this chapter, we showed the clock system of MSP430 microcontrollers and the timer-related capabilities to include the Watchdog timer, basic timer, RTC, input capture system, output compare system, and PWM system. The architects of the controller seek to provide embedded system designers with flexibility and minimum power usage by implementing three different clock signal generators (LFX1CLK, XT2CLK, and DCOCLK) and three clocks (ACLK, MCLK, and SMCLK) that can be configured for specific application use. The controller also allows peripheral systems, which usually run slower than the CPU, to run on a separate clock (ACLK), different from the clock used by the central processing unit, allowing the minimum use of power. This chapter also showed how the Watchdog timer can be used either to maintain software execution integrity or to generate periodic time intervals.

The RTC is used to keep track of calendar time with ability to inform year, month, day, hour, minute, and second. The input capture and output compare capabilities of the controller are used to interact with external world with time-related events. The input capture system can capture the time of an incoming event, which can be used to measure the pulse width of a signal and compute the period or frequency of an incoming periodic signal. It can also be used to count the number of event occurring externally. The output compare system is used to generate desired events on external pins at a desired time. For example, the system can generate a logic change, a pulse, a periodic pulse with a desired duty cycle. The timer system with its clock system and the capture/compare capabilities allows programmers to implement any time critical applications using MSP430.

7.13 REFERENCES AND FURTHER READING

Barrett S. F. and Pack D. J. *Embedded Systems Design with the Atmel Microcontroller*, Morgan & Claypool Publishers, 2010. DOI: 10.2200/S00138ED1V01Y200910DCS024.

Texas Instruments MSP430FG461x Code Examples, (SLAC118D). www.TI.com

Texas Instruments MSP430FR2433 Mixed-Signal Microcontroller, (SLASE59D), Texas Instruments, Revised 2018.

Texas Instruments MSP430FR4xx and MSP430FR2xx Family User's Guide, (SLAU445G), Texas Instruments, 2016. 309, 310

Texas Instruments MSP430FR58xx, MSP430FR59xx, and MSP430FR6xx Family, (SLAU367O), Texas Instruments, 2017. 296, 297, 309, 310

Texas Instruments MSP430FR599x, MSP430FR596x Mixed-Signal Microcontrollers, (SLASE54C), Texas Instruments, 2018.

7.14 CHAPTER PROBLEMS

Fundamental

1. What is the motivation for having three clock signal generators and three different clocks in MSP430 controllers?
2. To save power usage, how does one turn off the LFXT1CLK clock signal generator?
3. We want to configure the MCLK clock to run on the XT2CLK clock signal generator. Which register should be modified? What value should be in the register?
4. Suppose the LFXT1CLK is connected to a high-frequency watch crystal. Identify the register and the particular bit to configure the LFX1CLK clock signal generator on a high frequency mode.
5. How does one select the clock for the Basic Timer?
6. The Basic Timer 2 (BTCNT 2) has two scaling factors, which control register is used for the two scaling factors?
7. What is the password value and where should you write it to access the Watchdog timer system control register?
8. Give an example application where one might use the count Up/Down count mode for the free running counter, TAR.
9. The TAIFG flag when set indicates the free running counter TAR reached its limit. Why would you not want the flag to set when you are operating in the Up/Down counter mode?

Advanced

1. Program your MSP430 to generate clock signal frequency of 1.2 MHz.
2. Program your MSP430 controller to accept a pulse on the P1.0 pin and compute the pulse width.
3. Given a periodic pulse-width input signal, write a segment of code to compute the duty cycle using the input capture interrupt system of the MSP430 controller.
4. Program your MSP430 controller to generate a pulse (0–5 V and back down to 0 V) with 2 ms width using the Timer_A system.
5. Program your MSP430 using Timer_B system to generate a pulse-width modulated signal with frequency of 50 Hz and duty cycle of 40%.

Challenging

1. Program your MSP430 to accept any input periodic signal with varying frequency ranging from 10–1000 Hz and compute the input signal frequency.

2. Write a program that only activates itself if your MSP430 controller receives a 200 μs pulse (10% tolerance on the pulse width) from an external device on P1.0 pin, updates the number of times the designated pulse was received, displays the number on an LCD display unit for 5 s, and "sleeps" until the next pulse arrives.

CHAPTER 8

Resets and Interrupts

Objectives: After reading this chapter, the reader should be able to:

- describe MSP430 resets and their functions;
- explain the general concept of and the need for interrupts;
- describe in general terms the steps required to implement an interrupt service routine;
- identify MSP430 microcontroller's maskable and non-maskable interrupts;
- illustrate the process to assign priorities among resets and interrupts in the MSP430 microcontroller;
- explain the process to identify the source of resets and interrupts;
- describe the process to service interrupts; and
- properly configure the MSP430 microcontroller and write interrupt service routines to respond to interrupts

In any computer operation, it is often necessary to bring the internal processing state of a computer back to a known state due to program or system errors, or simply because it serves the purpose of an application at hand. Bringing the internal processing state of a computer back to a known state involves re-initializing registers, executing start-up instructions, and configuring peripheral devices, including I/O sub-systems. This process is called a reset. In other applications, there arises a need to stop executing the current task of a computer and taking care of an urgent request made by internal devices, external signals, or the result of the current or other software programs. These requests are called interrupts.

Resets and interrupts are closely related. In fact, the process of bringing the internal processing state of a computer back to a known state and performing a service routine as a response to an urgent request is almost identical, as we will see in this chapter.

8.1 MOTIVATION

One of the primary reasons to select a MSP430 microcontroller is its ability to minimize power usage by allowing a programmer to configure the microcontroller to run with different operational modes based on environmental factors or timed events. The subject of this chapter is

closely related to different means for the controller to implement mechanisms to switch between two or more different power consumption operating modes. Typically, while a controller is waiting for a designated event to occur, whether it is an arrival of a particular external signal or after a programmed elapsed time period, it operates in a power save mode with all or most of its clocks turned off. When the time comes, the microcontroller switches the operating mode (wake up from "sleep"), performs necessary tasks, and switches back to the power saving mode until the next designated event occurs. The back and forth switching between operating modes is accomplished using the built-in interrupt system, which is the topic of this chapter.

8.2 BACKGROUND

Typical embedded systems operate in environments where electrical and mechanical noises abound. These noise sources can often interfere with the proper operation of a microcontroller in an embedded system, which can cause skipping of intended instructions and unintentional change of register contents. One of the primary means to combat such undesired operation in the MSP430 microcontroller is the Watchdog timer system. By forcing the program to update special registers periodically, one can make sure that intended instructions are executed in a proper order. Otherwise, the microcontroller resets itself before resuming its operation. The Watchdog timer system reset example illustrates the function of microcontroller resets. They are used to bring the internal processing state of the controller to a default state.

An interrupt, on the other hand, is a software or hardware induced request which is expected to occur during the controller operation but whose time of occurrence is not known in advance. It is the programmer's responsibility to plan (write a special program called an interrupt service routine) to respond to an interrupt when it occurs. For example, suppose that you know a user will push an external button to halt a process sometime during the course of an operation of your MSP430 microcontroller but do not know the exact time when it will occur. A button push, in this example, is a hardware induced interrupt, and a programmer must write a separate "program" that will respond to the event appropriately to halt the process.

In general, there is another way for a microcontroller to detect an event, called polling. The polling method relies on using the resources of the controller to continuously monitor whether or not an event has occurred. This can be in the form of checking a flag continuously to see the flag status change (bit changes from 1 to 0, or vice versa) or the change of the logic level on an input pin. As the reader can imagine, the resources of the controller is "tied up" when polling is used to "wait" for an event to occur. The advantage of using the interrupt system, which we focus in this chapter, compared to the polling technique is the better usage of resources. Using the interrupt system, the controller does not have to poll an event but perform other operations or even turn itself off to save power. When an event occurs, the controller initiates a special routine associated with the interrupt.

Naturally, the polling method is simple to implement compared to the steps required to implement the interrupt method. The benefit, however, of the interrupt method is the conservation of limited, precious power resources.

8.3 MSP430 RESETS/INTERRUPTS OVERVIEW

The system control module (SYS) of the MSP430 governs the functions of resets and interrupts. For resets, there are three types: BOR, POR, and PUC. For interrupts, two types exist: non-maskable interrupts (NMI) or maskable interrupts (MI). The three different resets allow the MSP430 to start at three different start up states, providing the desired flexibility. The non-maskable interrupts are those that MSP430 controllers cannot or should not ignore, such as the critical power level indication. The maskable interrupts, on the other hand, are those requests, if necessary, that can be masked (ignored) by the CPU. The maskable interrupts require a programmer to activate them by writing to specific registers [SLAU445G, 2016, SLAU367O, 2017].

8.4 MSP430 RESETS

The BOR is triggered by five different events for the MSP430FR5994 and MSP430FR2433 controllers. The first is when the microcontroller is turned on. The BOR also occurs when a logic low is applied to the reset pin ($\overline{RTS}$/*NMI*), configured as a reset pin by the SYSNMI bit in SFRRPCR (Special Function Register Reset Pin Control Register), which is the second event. Figure 8.1 shows SFRRPCR. Note that a programmer can also use the same register to configure (SYSRSTRE—Reset enable/disable, SYSRSTUP—Reset pin pull-up/pull-down, and SYSNMIEES—edge select) the use of the pin. The third possible event that can cause the BOR is when the MSP430 controller wakes up from operating mode LPM3.5 or LPM4.5. The

System Function Register Reset Pin Control Register (SFRRPCR) at $0104

15	14	13	12	11	10	9	8
Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
r0	r0	r0	r0	r0	r0	r0	r0

7	6	5	4	3	2	1	0
Reserved	Reserved	Reserved	Reserved	SYSRSTRE	SYSRSTUP	SYSNMIIES	SYSNMI
r0	r0	r0	r0	rw-1	rw-1	rw-0	rw-0

Figure 8.1: Special function register reset pin control register.

fourth possible event for the reset is when the power management module (PMM) detects the power level of the controller (SVS) falls below a threshold value. Finally, the last event that triggers the BOR is software BOR events. For some applications, it is desired to trigger a Brownout reset using a software instruction. Setting PMMSWBOR (Power Management Module Soft-Ware BOR) bit in the PMMCTL0 register (Power Management Module Control Register 0) initiates a software generated BOR [SLAU445G, 2016, SLAU367O, 2017].

The second type of reset, the POR, is automatically triggered when the BOR occurs as shown in Figure 8.2. The POR is typically associated with the hardware system while the PUC reset is generally linked to software events. In addition to a BOR event, the POR is triggered by a software POR event.

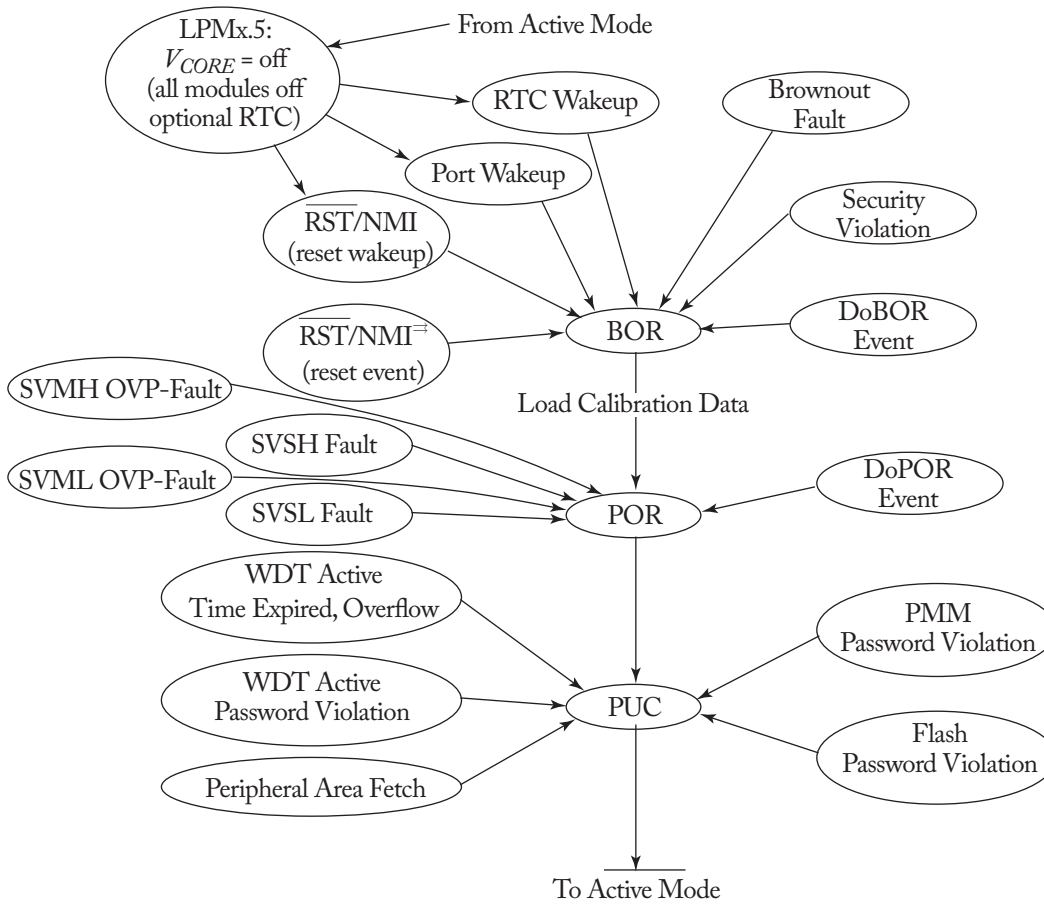


Figure 8.2: Reset activity of the MSP430 microcontroller. (Figure used with permission of Texas Instruments (www.ti.com).)

The third type of resets, the PUC reset, are initiated, in addition to the POR signal, by nine and five different events for the MSP430FR5994 and MSP430FR2433 controllers, respectively. For both controllers, whenever the controller detects a POR, the PUC reset is also triggered. For both controllers, the first and the second PUC reset events are associated with the Watchdog timer system. When the Watchdog timer expires or the Watchdog timer password is violated, the PUC reset is triggered. The other three common events that trigger the PUC reset for both controllers are the password violation to access the onboard flash memory, the password violation to access the PMM, and fetching from memory areas not populated.[1] A password to access the flash memory is necessary to prevent a runaway program from corrupting stored software. For the MSP430FR5994 controller, the following four additional events trigger the PUC reset: (1) memory protection unit password violation, (2) memory segmentation violation, (3) CS password violation, and (4) uncorrectable FRAM bit error.

In terms of the level of resets, the BOR initializes all systems while the POR and the PUC resets restore MSP430 conditions partially. Throughout the documents for the MSP430 microcontroller, one finds the POR and PUC reset values annotated using symbols such as rw-(1 or 0). For example, rw-0 indicates that the register bit value can be read or written and the initial value is 0 after the PUC reset, while rw-(0) denotes that the register bit can be read and written and the initial value is 0 after the POR (the parentheses are used to distinguish between POR and PUC). For the latter example case, the register value remains the same after the PUC reset. The general conditions of the MSP430 microcontroller after a system reset are:

- the $\overline{RST}/NMI$ pin is configured as the reset mode,
- all input and output pins are configured as input,
- the program counter register is loaded with the boot code start address (ex. 0xFFFE),
- the status register (SR) is cleared,
- all peripheral modules and registers are initialized, and the
- Watchdog timer is initialized (Watchdog mode).

Due to the steps taken by the MSP430 microcontroller after a reset, the programmer must make sure that, at the start of the proper program module, the stack pointer, the Watchdog specifications, and peripheral modules are initialized [SLAU445G, 2016, SLAU367O, 2017].

8.5 INTERRUPTS

A microcontroller normally executes instructions in an orderly fetch-decode-execute sequence as dictated by a user-written program as shown in Figure 8.3. All microcontrollers, however,

[1]The VMAIE (vacant memory access interrupt enable flag) bit must be set(1) for this reset to initiate.

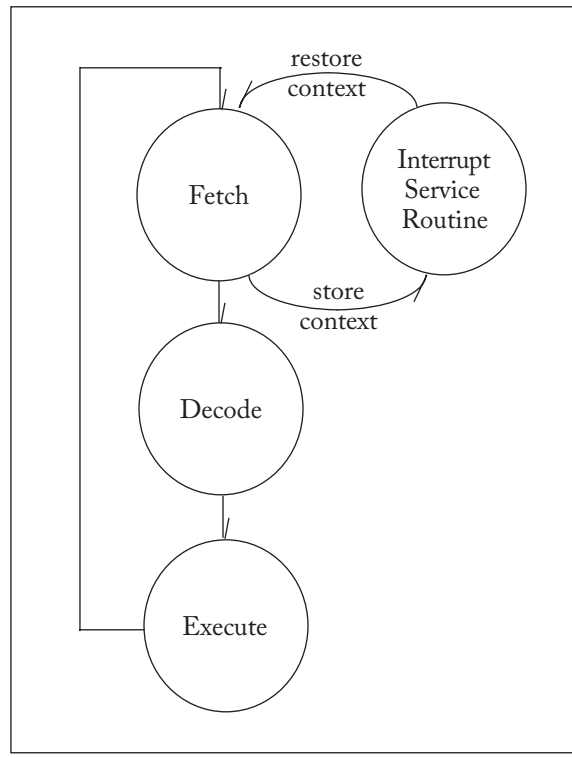


Figure 8.3: Microcontroller interrupt response.

are equipped to handle unscheduled, higher priority events that might occur inside or outside a microcontroller. To process such events, a microcontroller requires an interrupt system.

The interrupt system onboard a microcontroller allows it to respond to higher priority events. These events are expected events, but we do not know when they will occur. When an interrupt event does occurs, a microcontroller normally completes the instruction it is currently executing, stores key register values (context) on the stack, and transitions its program control to a special routine written to respond to the interrupt event. The special routine is a function called an interrupt service routine (ISR). Each interrupt will normally have its own interrupt specific ISR. Once the ISR is completed, the microcontroller will restore key register values from the stack and resume processing where it left off before the interrupt event occurred.

Applying the general concept of an interrupt, one can consider resets as interrupts with two exceptions. A reset does not cause the program counter to return to the point of operation when the reset was detected, and reset routines are fixed, not available to a programmer. Stretching the discussion a bit more, resets may be considered as non-maskable interrupts (NMI) with pre-programmed initialization routines.

Besides resets, there are two other types of NMIs supported by MSP430 microcontroller. The first type is the system generated NMIs (SNMI), and the second type are the ones generated by the user (UNMI). One example of an SNMI type interrupt, is the JTAG mailbox event. Recall that the JTAG interface is available for all MSP430 microcontrollers for the purpose of programming, debugging and testing the MSP430. The JTAG interface allows access to the CPU during program execution. One can configure the interface such that when data is read through the interface, a non-maskable interrupt occurs. The second SNMI occurs when FRAM errors occur and the last type of SNMI is caused by accessing a vacant memory location [SLAU445G, 2016, SLAU367O, 2017].

For the user-specified NMIs, there are two sources that can generate an UNMI. The first one is caused by an oscillator fault. The controller monitors the crystal oscillator frequency. An UNMI is triggered when the frequency falls outside an acceptable range. The second UNMI is caused by the logic state on the $\overline{RST}/NMI$ pin when the pin is configured for the NMI mode.

The MSP430 microcontroller has many MI sources. The difference between NMIs and MIs is that unlike NMIs, MIs can be programmed to be ignored by the CPU by turning off the GIE bit of the status register. To enable a maskable interrupt, not only the GIE bit must be set, but also each subsystem interrupt in use must be enabled. These subsystems are enabled using appropriate bits in the interrupt enable register (SFRIE1), shown in Figure 8.4. When one of these interrupts occurs, the corresponding flag in the interrupt flag register (SFRIFG1), shown in Figure 8.5, is set.

In most microcontrollers, including the MSP430, the starting address for each interrupt service routine, the special function to perform in response to an interrupt, is stored in a predesignated location which the CPU recognizes. The addresses reside in consecutive memory locations and are collectively designated as interrupt vectors. For the MSP430 microcontroller, the memory locations are 0xFFFF through 0xFF80, where each vector takes up two memory locations. Memory space is available in the interrupt vector table for up to 64 different interrupt sources. Figure 8.6 provides the table of interrupt vectors for MSP430 microcontroller.

During the development phase, it is handy to configure the top of RAM space as alternative locations for interrupt vectors by setting the SYSRIVECT (RAM-based interrupt vectors) bit in the System Control Register (SYSCTL) register, shown in Figure 8.7.

8.5.1 INTERRUPT HANDLING PROCESS

In this section, we describe the process of handling an interrupt event. Once a maskable interrupt is configured to be active, and an interrupt event occurs, a flag that corresponds to the particular interrupt event is asserted to indicate to the CPU that there is an interrupt waiting to be serviced. The CPU then takes the following actions in the order shown to provide an orderly transition from normal program operation to the interrupt service routine and back again.

1. Complete the current instruction.

Interrupt Enable Register (SFRIE1)

15	14	13	12	11	10	9	8
Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
r0	r0	r0	r0	r0	r0	r0	r0

7	6	5	4	3	2	1	0
JMBOUTIE	JMBINIE		NMIIE	VMAIE	Reserved	OFIE	WDTIE
rw-0	rw-0	rw-0	rw-0	rw-0	r0	rw-0	rw-0

0 - interrupt disabled 1 - interrupt enabled

JMBOUTIE - JTAG mailbox output interrupt enable
JMBINIE - JTAG mailbox input interrupt enable
NMIIE - NMI pin interrupt enable
VMAIE - Vacant memory access interrupt enable
OFIE - Oscillator fault interrupt enable
WDTIE - Watchdog timer interrupt enable

Figure 8.4: Interrupt enable register.

2. Store the contents of the program counter (PC) onto the stack.
3. Store the contents of the status register (SR) onto the stack.
4. Choose the highest priority interrupt if multiple interrupts are pending.
5. Reset the interrupt request flag.
6. Clear the Status Register to prevent additional interrupts from occurring and to switch from the low power mode to the normal power mode (if configured).
7. Load the contents of the interrupt vector onto the program counter.
8. Execute the specified interrupt service routine.
9. Once the service routine is finished (with the RETI instruction), restore the SR and then PC values from the stack.
10. Resume normal operation.

Interrupt Flag Register (SFRIFG1)

15	14	13	12	11	10	9	8
Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
r0	r0	r0	r0	r0	r0	r0	r0

7	6	5	4	3	2	1	0
JMBOUTIFG	JMBINIFG	Reserved	NMIIFG	VMAIFG	Reserved	OFIFG	WDTIFG
rw-(1)	rw-0	r0	rw-0	rw-0	r0	rw-(1)	rw-0

0 - no interrupt 1 - interrupt pending

JMBOUTIFG - JTAG mailbox output interrupt flag
JMBINIFG - JTAG mailbox input interrupt flag
NMIIFG - NMI pin interrupt flag
VMAIFG - Vacant memory access interrupt flag
OFIFG - Oscillator fault interrupt flag
WDTIFG - Watchdog timer interrupt flag

Figure 8.5: Interrupt flag register.

Interrupt Source	Flag	Priority	Vector Location
Resets	WDTIFG, KEYV, ..	Highest	FFFE
System NMI		:	FFFC
User NMI	NMIFG, OFIFG, ACCVIFG,...	:	FFFA
Device Specific		:	FFF8
Watchdog Timer	WDTIFG	:	:
:	:	:	:

Figure 8.6: MSP430 interrupt vector table [SLASE54C, 2018].

System Control Register (SYSCTL)

15	14	13	12	11	10	9	8
Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
r0	r0	r0	r0	r0	r0	r0	r0

7	6	5	4	3	2	1	0
Reserved	Reserved	SYSJTAGPIN	SYSBSLIND	Reserved	SYSPMMPE	Reserved	SYSRIVECT
r0	r0	rw-(0)	r-0	r0	rw-(0)	r0	rw-(0)

Figure 8.7: System control register.

Once an interrupt is detected, it takes six clock cycles for the MSP controller to begin interrupt processing, and it takes five clock cycles to restore the SR and PC values and resume executing normally after the interrupt service ends. Figure 8.8 shows the stack configuration before and after step 3 and step 9.

Following the procedure described above, everything seems to be straightforward if for each interrupt source, there was only a single and unique interrupt flag. Unfortunately, that is not always the case for the MSP430 microcontroller. For example, for all interrupts associated with the Timer_A module, a single flag, TAIFG (Timer_A interrupt flag), is set. It becomes a programmer's responsibility to resolve the ambiguity as a part of the interrupt service routine

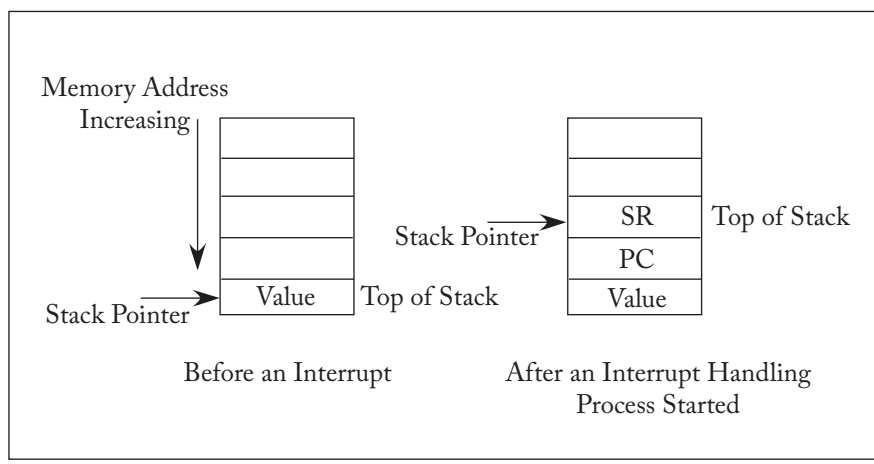


Figure 8.8: Stack before and after an interrupt.

by checking the TAIV register (Timer_A Interrupt Vector Register), which contains a value identifying the interrupt source. Similarly, the source of a non-maskable interrupt or a reset is resolved with the help of three interrupt vector registers: SYSRSTIV (Reset Interrupt Vector register), SYSSNIV (System NMI Vector register), and SYSUNIV (User NMI Vector register). When a reset occurs, based on the source, it generates an interrupt vector offset value in the SYSRSTIV register, as shown in Figure 8.9.

Reset Interrupt Vector Registor (SYSRSTIV) at $019E

15	14	13	12	11	10	9	8
0	0	0	0	0	0	0	0
r0	r0	r0	r0	r0	r0	r0	r0

7	6	5	4	3	2	1	0
0	0	SYSRSTVEC					0
r0	r0	r-0	r-0	r-0	r-0	r-1	r0

SYSRSTIV	Value	Interrupt Type
	0000h	No interrupt
	0002h	Brownout (BOR)
	0004h	RST/NMI (BOR)
	0006h	PMMSWBOR (BOR)
	0008h	Wakeup from LPMx.5 (BOR)
	000Ah	Security violation (BOR)
	000Ch	SVSL (POR)
	000Eh	SVSH (POR)
	0010h	SVML_OVP (POR)
	0012h	SVMH_OVP (POR)
	0014h	PMMSWPOR (POR)
	0016h	WDT time out (PUC)
	0018h	WDT password violation (PUC)
	001Ah	Flash password violation (PUC)
	001Ch	PLL unlock (PUC)
	001Eh	PERF peripheral/configuration area fetch (PUC)
	0020h	PMM password violation (PUC)
	0022–003Eh	Reserved for future use

Figure 8.9: Reset interrupt vector register.

The SYSRSTVEC bits (Reset interrupt vectors—bits 1–5) determine the interrupt vector offset value. Similarly, the SYSSNIV and the SYSUNIV registers are used to identify the source of an interrupt; see Figures 8.10 and 8.11. For example, when dealing with a user non-maskable interrupt, as the first step of the service routine, the contents of the SYSUNIV and PC are added as shown in Figure 8.12. The offset value governs the execution of the appropriate portion of the interrupt service routine.

System NMI Vector Registor (SYSSNIV) at $019C

15	14	13	12	11	10	9	8
0	0	0	0	0	0	0	0
r0	r0	r0	r0	r0	r0	r0	r0

7	6	5	4	3	2	1	0
0	0	0	SYSSNVEC				0
r0	r0	r0	r-0	r-0	r-0	r-0	r0

SYSRSTIV	Value	Interrupt Type
	0000h	No interrupt
	0002h	SVMLIFG interrupt
	0004h	SVMHIFG interrupt
	0006h	SVSMLDLYIFG interrupt
	0008h	SVSMHDLYIFG interrupt
	000Ah	VMAIFG interrupt
	000Ch	JMBINIFG interrupt
	000Eh	JMBOUTIFG interrupt
	0010h	SVMLVLRIFG interrupt
	0012h	SVMHVLRIFG interrupt
	0014h	Reserved for future use

Figure 8.10: System non-maskable interrupt vector register.

When a MSP430 microcontroller is shipped, the interrupt service routines for resets and system non-maskable interrupts are already programmed and should not be altered. We described the above process to illustrate a similar process to resolve the source of a non-maskable user interrupt or maskable interrupt in your interrupt service routine. We should also note that some devices have separate bus error interrupts, which can also be serviced in the same manner as shown above using the SYSBERRIV (Bus Error Interrupt Vector) register.

User NMI Vector Registor (SYSUNIV) at $019A

15	14	13	12	11	10	9	8
0	0	0	0	0	0	0	0
r0	r0	r0	r0	r0	r0	r0	r0

7	6	5	4	3	2	1	0
0	0	0	SYSUNVEC				0
r0	r0	r0	r-0	r-0	r-0	r-0	r0

SYSRSTIV	Value	Interrupt Type
	0000h	No interrupt
	0002h	NMIIFG interrupt
	0004h	OFIFG interrupt
	0006h	ACCVIFG interrupt
	0008h	Reserved for future use

Figure 8.11: User non-maskable interrupt vector register.

```
SNI_ISR:        ADD    &SSSNIV,PC          ; Add offset to jump table
                RETI                       ; No interrupt
                JMP    SVML_ISR            ; vector 2
                JMP    SVMH_ISR            ; vector 4
                :
                JMP    SVMHV_ISR           ; vector 12
Invalid_ISR     RETI                       ; vector 14
SVML_ISR:                                  ; ISR for vector 2
                :
                RETI
SVMH_ISR:                                  ; ISR for vector 4
                :
                RETI
:
SVMHV_ISR:                                 ; ISR for vector 12
                :
                RETI
```

Figure 8.12: Sample ISR for user non-maskable interrupts.

8.5.2 INTERRUPT PRIORITY

As there are multiple reset and interrupt sources associated with the MSP430 microcontroller, shown in Figure 8.6, priorities among interrupts must be defined in advance to handle situations when more than one interrupt occurs simultaneously. The MSP430 microcontroller sets the priority in the following manner. The resets hold the highest priority, followed by the system NMIs, the user NMIs, and the device specific maskable interrupts. Figures 8.13, 8.14, and 8.15 show the priority list for the MSP430FR5994 microcontroller. Since each version of the MSP430 controller has a different number of interrupts associated with its subsystems, one should always consult the datasheet for the particular controller to find the interrupt priority list.

A typical MSP430 microcontroller configuration has built-in interrupt systems for a direct memory access (DMA) controller, a DAC, an analog comparator (Comp_B), digital I/O ports (P1 and P2), one or more Timer_A system, a Timer_B system, a real-time clock A (RTC_A) system, a real-time clock B (RTC_B) system, analog-to-digital converter (ADC), a UART system, and a USB system. Since it requires a detailed understanding of each system to use the associated interrupt, we defer the discussion of each interrupt system to chapters that cover the subsystems.

8.5.3 INTERRUPT SERVICE ROUTINE (ISR)

Most of the interrupt handling process described in this chapter takes place automatically (you, as a programmer, do not need to program them). In fact, for the resets, all processing is completed automatically. For maskable interrupts, however, your responsibility as a programmer is to (1) turn on the global interrupt enable (GIE), (2) initialize the stack pointer, (3) configure the interrupt vector table (initialize the start address of your ISR), (4) enable the appropriate interrupt local enable bit (SFRIE1 register), and (5) write the corresponding interrupt service routine. In this section, we present the last task, writing an ISR.

Examples: In this example, we write an ISR using the Timer_A interrupt system. Recall that all Timer_A system related interrupts have the same interrupt vector. Thus, a Timer_A ISR must identify the source before executing a desired task. In MSP430-related microcontrollers, the Timer_A system contains two separate subsystems, Timer0_A and Timer1_A. Each subsystem has I/O channels, where each channel has the same interrupt vector table entry. Using assembly language, we can write an ISR similar to the one provided in Figure 8.12. The ISR is shown in Figure 8.16.

The next example shows how to implement a Timer0_A0 related ISR. Note how the ISR is configured. The code snapshot below shows how to tie the starting address of the ISR to the proper location in the ISR. We assume definitions for all MSP430FR5994-related interrupts are properly made.

Interupt Source	Interrupt Flag	System Interrupt	Word Address	Priority
System Reset Power up, brownout, supply supervisor External reset RST Watchdog time-out (watchdog mode) WDT, FRCTL, MPU, CS, PMM password violation FRAM uncorrectable bit error detection MPU segment violation Software POR, BOR	SVSHIFG PMMRSTIFG WDTIFG WDTPW, FRCTLPW, MPUPW, CSPW, PMMPW, UBDIFG MPUSEGIIFG, MPUSEG1IFG, MPUSEG2IFG, MPUSEG3IFG PMMPORIFG, PMMBORIFG (SYSRSTIV)(1)(2)	Reset	0FFFEh	Highest
System NMI Vacant memory access JTAG mailbox FRAM access time error FRAM write protection error FRAM bit error detection MMPU segment violation	VMAIFG JMBINIFG, JMBOUTIFG ACCTEIFG, WPIFG MPUSEGIIFG, MPUSEG1IFG, MPUSEG2IFG, MPUSEG3IFG (sYSSNIV)(1)(3)	(Non)maskable	0FFFCh	
User NMI External NMI Oscillator fault	NMIFG, OFFIFG (SYSUNIV)(1)(3)	(Non)maskable	0FFFAh	
Comparator_E	CEIFG. CEIIFG (CEIV)(1)	Maskable	0FFF8h	
TB0	TB0CCR0.CCIFG	Maskable	0FFF6h	
TB0	TB0CCR1.IFG ... TB0CCR6. CCIFG, TB0CTL. TBIFG (TB0IV)(1)	Maskable	0FFF4h	
Watchdog timer (interval timer mode)	WDTIFG	Maskable	0FFF2h	
eUSCI_A0 receive or transmit	UCA0IFG: UCRXIFG, UCTXIFG (SPI mode) UCA0IFG: UCSTTIFG, UCTXCPTIFG, UCRXIFG, UCTXIFG (UART MODE) (UCADIV)(1)	Maskable	0FFF0h	
eUSCI_B0 receive or transmit	UCB0IFG: UCRXIFG, UCTXIFG (SPI mode) UCB0IFG: UCALIFG, UCNACKIFG, UCSTTIFG, UCSTPIFG, UCRXIFG0, UCTXIFG0, UCRXIFG1, UCTXIFG1, UCRXIFG2, UCTXIFG2, UCRXIFG3, UCTXIFG3, UCCNTIFG, UCBIT9IFG (I^2C mode) (UCB0IV)(1)	Maskable	0FFEEh	
ADC12_B	ADC12IFG0 to ADC12IFG31 ADC12LOIFG, ADC12INIFG, ADC12HIIFG, ADC12RDYIFG, ADC21OVIFG, ADC12TOVIFG (ADC12IV)(1)(4)	Maskable	0FFECh	
TA0	TA0CCR0.CCIFG	Maskable	0FFEAh	

(1) Multiple source flags
(2) A reset is generated if the CPU tries to fetch instructions from peripheral space.
(3) (Non)maskable: the individual interrupt enable bit can disable an interrupt event, but the general interrupt enable bit cannot disable it.
(4) Only on devices with ADC, otherwise reserved.

Figure 8.13: Interrupt priority list for MSP430FR5994. (Illustration used with permission of Texas Instruments (www.ti.com).)

Interupt Source	Interrupt Flag	System Interrupt	Word Address	Priority
TA0	TA0CCR1.CCIFG, TA0CCR2.CCIFG, TA0CTL.TAIFG (TA0IV)(1)	Maskable	0FFE8h	
EUSCI_A1 receive or transmit	UCA1IFG: UCRXIFG, UCTXIFG (SPI mode) UCA1IFG: UCSTTIFG, UCTXCPTIFG, UCRXIFG, UCTXIFG (UART mode) (UCA1IV)(1)	Maskable	0FFE6h	
DMA	DMA0CTL.DMAIFG, DMA1CTL.DMAIFG, DMA2CTL.DMAIFG (DMAIV)(1)	Maskable	0FFE4h	
TA1	TA1CCR0.CCIFG	Maskable	0FFE2h	
TA1	TA1CCR1.CCIFG, TA1CCR2.CCIFG, TA1CTL.TAIFG (TA1IV)(1)	Maskable	0FFE0h	
I/O port P1	P1IFG.0 to P1IFG.7 (P1IV)(1)	Maskable	0FFDEh	
TA2	TA2CCR0.CCIFG	Maskable	0FFDCh	
TA2	TA2CCR1.CCIFG TA2CTL.TAIFG (TA2IV)(1)	Maskable	0FFDAh	
I/O port P2	P2IFG.0 to P2IFG.7 (P2IV)(1)	Maskable	0FFD8h	
TA3	TA3CCR0.CCIFG	Maskable	0FFD6h	
TA3	TA3CCR1.CCIFG TA3CTL.TAIFG (TA3IV)(1)	Maskable	0FFD4h	
I/O port P3	P3IFG.0 to P3IFG.7 (P3IV)(1)	Maskable	0FFD2h	
I/O port P4	P4IFG.0 to P4IFG.2 (P4IV)(1)	Maskable	0FFD0h	
RTC_C	RTCRDYIFG, RTCTEVIFG, RTCAIFG, RT0PSIFG, RT1PSIFG, RTCOFIFG (RTCIV)(1)	Maskable	0FFCEh	
AES	AESRDYIFG	Maskable	0FFCCh	
TA4	TA4CCR0.CCIFG	Maskable	0FFCAh	
TA4	TA4CCR1.CCIFG TA4CTL.TAIFG (TA4IV)(1)	Maskable	0FFC8h	
I/O port P5	P5IFG.0 to P5IFG.2 (P5IV)(1)	Maskable	0FFC6h	
I/O port P6	P6IFG.0 to P6IFG.2 (P6IV)(1)	Maskable	0FFC4h	
eUSCI_A2 receive or transmit	UCA2IFG: UCRXIFG, UCTXIFG (SPI mode) UCA2IFG: UCSTTIFG, UCTXCPTIFG, UCRXIFG, UCTXIFG (UART mode) (UCA2IV)(1)	Maskable	0FFC2h	
eUSCI_A3 receive or transmit	UCA3IFG: UCRXIFG, UCTXIFG (SPI mode) UCA3IFG: UCSTTIFG, UCTXCPTIFG, UCRXIFG, UCTXIFG (UART mode) (UCA3IV)(1)	Maskable	0FFC0h	
eUSCI_B1 receive or transmit	UCB1IFG: UCRXIFG, UCTXIFG (SPI mode) UCB1IFG: UCALIFG, UCNACKIFG, UCSTTIFG, UCSTPIFG, UCRXIFG0, UCTXIFG0, UCRXIFG1, UCTXIFG1, UCRXIFG2, UCTXIFG2, UCRXIFG3, UCTXIFG3, UCCNTIFG, UCBIT9IFG (I2C mode) (UCB1IV)(1)	Maskable	0FFBEh	

(1) Multiple source flags

Figure 8.14: Interrupt priority list for MSP430FR5994. (Illustration used with permission of Texas Instruments (www.ti.com).)

Interupt Source	Interrupt Flag	System Interrupt	Word Address	Priority
eUSCI_B2 receive or transmit	UCB2IFG: UCRXIFG, UCTXIFG (SPI mode) UCB2IFG: UCALIFG, UCNACKIFG, UCSTTIFG, UCSTPIFG, UCRXIFG0, UCTXIFG0, UCRXIFG1, UCTXIFG1, UCRXIFG2, UCTXIFG2, UCRXIFG3, UCTXIFG3, UCCNTIFG, UCBIT9IFG (I2C mode) (UCB2IV)(1)	Maskable	0FFBCh	
eUSCI_B3 receive or transmit	UCB3IFG: UCRXIFG, UCTXIFG (SPI mode) UCB3IFG: UCALIFG, UCNACKIFG, UCSTTIFG, UCSTPIFG, UCRXIFG0, UCTXIFG0, UCRXIFG1, UCTXIFG1, UCRXIFG2, UCTXIFG2, UCRXIFG3, UCTXIFG3, UCCNTIFG, UCBIT9IFG (I2C mode) (UCB3IV)(1)	Maskable	0FFBAh	
I/O port P7	P7IFG.0 to P7IFG.2 (P7IV)(1)	Maskable	0FFB8h	
I/O port P8	P6IFG.0 to P6IFG.2 (P8IV)(1)	Maskable	0FFB6h	
LEA (MSP430FR599x only)	CMDIFG, SDIIFG, OORIFG,TIFG, COVLIFG LEAIV(1)	Maskable	0FFB4h	
(1) Multiple source flags				

Figure 8.15: Interrupt priority list for MSP430FR5994. (Illustration used with permission of Texas Instruments (www.ti.com).)

```
TA_ISR:         ADD     &TA0IV,PC       ; Add offset to jump table
                RETI                    ; No interrupt
                JMP     CCIFG1_ISR      ; vector 2
                JMP     CCIFG2_ISR      ; vector 4
                :
                JMP     CCIFG6_ISR      ; vector 12
TA0IFG_ISR                              ; vector 14
                :
                RETI
CCIFG1_ISR:                             ; ISR for vector 2
                :
                RETI
CCIFG2_ISR:                             ; ISR for vector 4
                :
                RETI
:
CCIFG6_ISR:                             ; ISR for vector 12
                :
                RETI
```

Figure 8.16: Timer_A interrupt service routine.

```
#pragma vector=TIMER0_A0_VECTOR
__interrupt void TIMER0_A0_ISR(void)
```

In this example, only a single Timer0_A0 related ISR is employed.

```
// *************************************************************************
//                          MSP430 CODE EXAMPLE DISCLAIMER
//MSP430 code examples are self-contained low-level programs that
//typically demonstrate a single peripheral function or device feature
//in a highly concise manner. For this the code may rely on the
//device's power-on default register values and settings such as the
//clock configuration and care must be taken when combining code from
//several examples to avoid potential side effects. Also see
//www.ti.com/grace for a GUI- and www.ti.com/msp430ware for an API
//functional library-approach to peripheral configuration.
//
// --/COPYRIGHT--
//*************************************************************************
//MSP430FR5x9x Demo - Timer0_A3, Toggle P1.0, CCR0 Cont Mode ISR,
//                    DCO SMCLK
//
//Description: Toggle P1.0 using software and TA_0 ISR. Timer0_A is
//configured for continuous mode, thus the timer overflows when TAR
//counts to CCR0. In this example, CCR0 is loaded with 50000.
//ACLK = n/o, MCLK = SMCLK = TACLK = default DCO = ~1MHz
//
//                    MSP430FR5994
//                 -------------------------
//           /|\   |                         |
//            |    |                         |
//            --   | RST                 |
//                 |                         |
//                 |                 P1.0    | --> LED
//
//William Goh, Texas Instruments Inc., October 2015
//Built with IAF Embedded Workbench V6.30 & Code Composer Studio V6.1
//*************************************************************************

#include <msp430.h>

int main(void)
```

```
{
WDTCTL = WDTPW | WDTHOLD;           //Stop WDT

P1DIR   |= BIT0;                    //Configure GPIO
P1OUT  |= BIT0;

//Disable the GPIO power-on default high-impedance mode to activate
//previously configured port settings
PM5CTL0 &= ~LOCKLPM5;

TA0CCTL0 = CCIE;                    //TACCR0 Interrupt enabled
TA0CCR0 = 50000;
TA0CTL = TASSEL__SMCLK | MC__CONTINOUS; //SMCLK, continuous mode
__bis_SR_register(LPM0_bits | GIE); //Enter LPM0 w/ interrupt
__no_operation();                   //For debugger
}

//------------------------------------------------------------------------
//Timer0_A1 interrupt service routine
//------------------------------------------------------------------------

#if defined(__TI_COMPILER_VERSION__)||defined(__IAR_SYSTEMS_ICC__)
#pragma vector = TIMER0_A0_VECTOR
__interrupt void Timer0_A0_ISR (void)
#elif defined(__GNUC__)
void __attribute__ ((interrupt(TIMER0_A0_VECTOR)))Timer0_A0_ISR (void)
#else
#error Compiler not supported!
#endif
{
P1OUT ^= BIT0;
TA0CCR0 += 50000;                    //Add Offset to TA0CCR0
}

//------------------------------------------------------------------------
```

In this example, the MSP430F5438 Timer0_A is again used with the associated interrupt vector generator demonstrated.

```
//******************************************************************************
//                          MSP430 CODE EXAMPLE DISCLAIMER
//MSP430 code examples are self-contained low-level programs that
//typically demonstrate a single peripheral function or device feature
//in a highly concise manner. For this the code may rely on the
//device's power-on default register values and settings such as the
//clock configuration and care must be taken when combining code from
//several examples to avoid potential side effects. Also see
//www.ti.com/grace for a GUI- and www.ti.com/msp430ware for an API
//functional library-approach to peripheral configuration.
//
// --/COPYRIGHT--
//******************************************************************************
//MSP430FR5x9x Demo - Timer0_A3, Toggle P1.0, Overflow ISR, 32kHz ACLK
//
//Description: Toggle P1.0 using software and the Timer0_A overflow ISR.
//In this example, an ISR triggers when TA overflows.  Inside the ISR
//P1.0 is toggled.  Toggle rate is exactly 0.5Hz.  Proper use of the
//TAIV interrupt vector generator is demonstrated.
//
//ACLK = TACLK = 32768Hz, MCLK = SMCLK = CD0/2 = 8 MHz/2 = 4MHz
//
//                   MSP430FR5994
//                 --------------------------
//           /|\   |                     XIN | -
//            |    |                         |
//            --   | RST                XOUT | -
//                 |                         |
//                 |                  P1.0   | --> LED
//
//William Goh, Texas Instruments Inc., October 2015
//Built with IAF Embedded Workbench V6.30 & Code Composer Studio V6.1
//******************************************************************************

#include <msp430.h>

int main(void)
{
WDTCTL = WDTPW | WDTHOLD;            //Stop WDT
```

```
P1DIR |= BIT0;                          //Configure GPIO
P1OUT |= BIT0;
PJSEL0 |= BIT4 | BIT5;

//Disable the GPIO power-on default high0impedance mode to activate
//previously configured port settings
PMSCTL0 &= ~LOCKLPM5;                   //Setup XT1
CSCTL0_H = CSKEY_H;                     //Unlock CS registers
CSCTL1 = DCOFSEL_6;                     //Set DC0 to 8MHz
                                        //set ACLK = XT1; MCLK = DCO
CSCTL2 = SELA__ LFXTCLK | SELS__DCOCLK | SELM__DCOCLK;
CSCTL3=DIVA__1 | DIVS__2 | DIVM__2;//Set all dividers
CSCTL4 &= ~LFXTOFF;
do
   {
   CSCTL5 &= ~LFXTOFFG;                 //Clear XT1 fault flag
   SFRIFG1 &= ~OFIFG;
   }while (SFRIFG1 & OFIFG);            //Test oscillator fault flag
   CSCTL0_H = 0;

//ACLK, contmode, clear TAR, enable overflow interrupt
TA0CTL = TASSEL__ACLK | MC__CONTINUOUS | TACLR | TAIE;
__bis_SR_register(LPM0_bits | GIE);  //Enter LPM0 w/ interrupt
__no_operation();                       //For debugger
}

//------------------------------------------------------------------------
// Timer0_A1 Interrupt Vector (TAIV) handler
//------------------------------------------------------------------------

#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector = TIMER0_A1_VECTOR
__interrupt void Timer0_A1_ISR (void)
#elif defined(__GNUC__)
void __attribute__ ((interrupt(TIMER0_A1_VECTOR))) Timer0_A1_ISR (void)
#else
#error Compiler not supported!
#endif
```

```
{
   switch(__even_in_range(TA0IV, TAIV__TAIFG))
     {
case TAIV__NONE;    break;          //No interrupt
case TAIV__TACCR1;  break;          //CCR1 not used
case TAIV__TACCR2;  break;          //CCR2 not used
case TAIV__TACCR3;  break;          //reserved
case TAIV__TACCR4;  break;          //reserved
case TAIV__TACCR5;  break;          //reserved
case TAIV__TACCR6;  break;          //reserved
case TAIV__TAIFG; P1OUT ^= BIT0;  //overflow
                        break;
default: break;
}
}

//*******************************************************************
```

In this final example, the Watchdog timer interrupt is used to toggle MSP430 microcontroller pin P1.0.

```
//*******************************************************************
//                         MSP430 CODE EXAMPLE DISCLAIMER
//MSP430 code examples are self-contained low-level programs that
//typically demonstrate a single peripheral function or device feature
//in a highly concise manner. For this the code may rely on the
//device's power-on default register values and settings such as the
//clock configuration and care must be taken when combining code from
//several examples to avoid potential side effects. Also see
//www.ti.com/grace for a GUI- and www.ti.com/msp430ware for an API
//functional library-approach to peripheral configuration.
//
// --/COPYRIGHT--
//*******************************************************************
//MSP430FR5x9x Demo-WDT, Toggle P1.0, Interval Overflow ISR, DCO SMCLK
//
//Description: Toggle P1.0 using software timed by the WDT ISR. Toggle
//rate is approximately 30ms = {(1MHz) / 32768} based on DCO = 8MHz
//clock source used in this example for the WDT.
// ACLK = n/a, SMCLK =1MHz
//
```

```
//                   MSP430FR5994
//                 -------------------------
//           /|\   |                         |
//            |    |                         |
//            --   | RST                     |
//                 |                         |
//                 |                 P1.0    | --> LED
//
//William Goh, Texas Instruments Inc., October 2015
// Built with IAF Embedded Workbench V6.30 & Code Composer Studio V6.1
//******************************************************************************

#include <msp430.h>

void main(void)
{
WDTCTL = WDTPW | WDTSSEL__SMCLK | WDTTMSEL | WDTCNTCL | WDTIS__32k;

P1DIR |= BIT0;                       //Configure GPIO

//Disable the GPIO power-on default high impedance mode to activate
//previously configured port settings
PMSCTL0 &= ~LOCKLPM5;

CSCTL0_H = CSKEY_H;         //Unlock CS registers
CSCTL1 = DCOFSEL_6;  //Set DCO = 8MHz
                                  //Set ACLK = VL0, MCLK = DCO
CSCTL2 = SELA__VLOCLK | SELS__DCOCLK | SELM__DCOCLK;
                                  //Set all dividers
CSCTL3 = DIVA__1 | DIVS__8 | DIVM__8;
CSCTL0_H = 0;
SFRIE1 |= WDTIE;                  //Enable WDT interrupt
__bis_SR_register(LPM0_bits+GIE);  //Enter LPM0, enable interrupts
__no_operation();                 //For debugger
}

//******************************************************************************
// Watchdog timer interrupt service routine
//******************************************************************************
```

```
#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector = WDT_VECTOR
__interrupt void WDT_ISR (void)
#elif defined(__GNUC__)
void __attribute__ ((interrupt(WDT_VECTOR))) WDT_ISR (void)
#else
#error Compiler not supported!
#endif
{
P1OUT ^= BIT0;
}

//*******************************************************************
```

It should be noted, if recursive interrupts or nesting interrupts are desired, the GIE bit in SR should be set inside an ISR. If nesting interrupts are allowed on purpose, a programmer must be sure that the stack will have enough space to accommodate the number of consecutive nested interrupts.

8.6 LABORATORY EXERCISE

In many smart homes, electronic appliances are networked together to monitor power usage throughout the day. The idea is to minimize collective power usage during the peak time and perform tasks during the low power usage time. For example, a refrigerator can delay generating ice cubes, if it detects that the electric fan is operating at the same time. An LCD may turn itself off when it is not being used. Furthermore, assume that some of these electronic systems in house are battery operated. In this laboratory exercise, you are to implement the controller for a battery operated electronic temperature controller for a living room, which must periodically transmit wirelessly its status (power level) to a central station, check to see whether the temperature is within a programmed range, and turn on a heater/air conditioner if necessary until the desired temperature is reached.

While it is not in use, the controller should be in a power saving mode, LPM3.5, and should be turned on every 5 min to perform the periodic task. It should use its internal clock as the timer and use the related interrupt system to "wake up" and "sleep" during and in between consecutive tasks. Since we have not covered the wireless transmission capabilities of MSP430, simply write 0x01h to Port 1 to indicate that the power level is healthy and 0xFFh to let the central controller know that the power is below the desired level. Assume that the power level is constantly updated at memory locations 0x1000h-0x1001h and values above 0x8000 represent sufficient power capacity. To turn on the heater or the air conditioner, your program must write 0x01 or 0xFF to Port 2, respectively. Writing 0x00h to Port 2 turns off both systems.

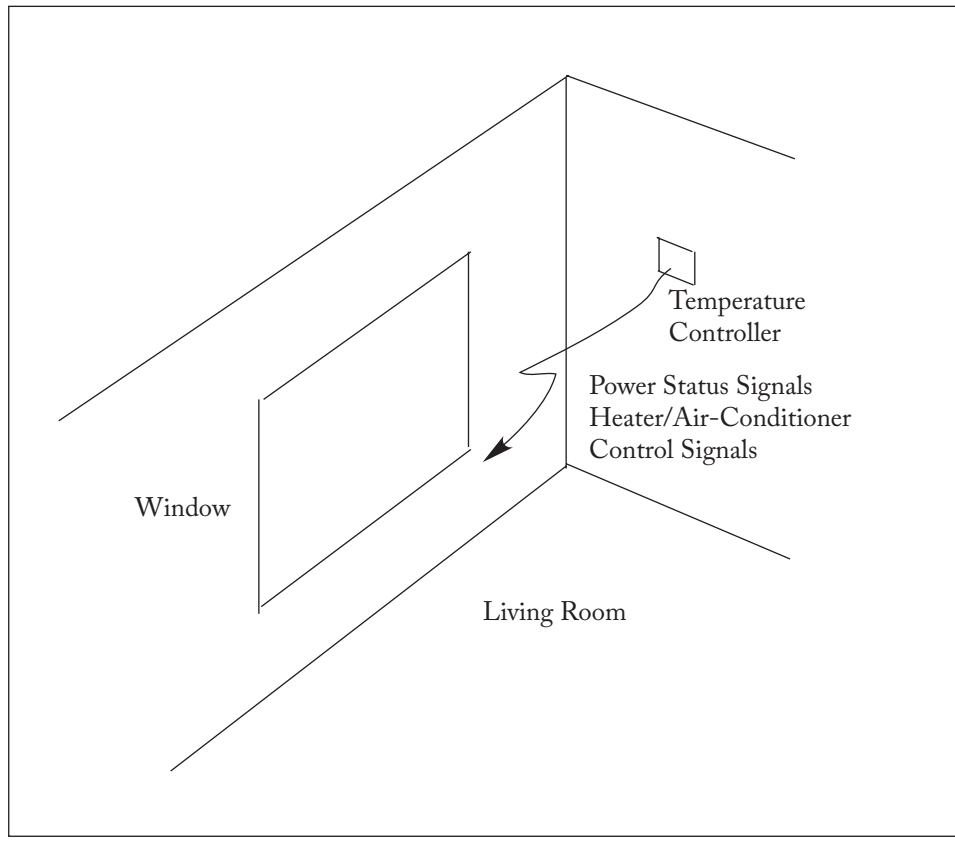


Figure 8.17: Temperature controller for a smart home.

Figure 8.17 shows the setup for your temperature controller.

8.7 REFERENCES AND FURTHER READING

MSP430FR2433 LaunchPad Development Kit (MSP-EXP430FR2433), (SLAU739), Texas Instruments, 2017.

MSP430FR2433 Mixed-Signal Microcontroller, (SLASE59D), Texas Instruments, 2018.

MSP430FR4xx and MSP430FR2xx Family User's Guide, (SLAU445G), Texas Instruments, 2016. 333, 334, 335, 337

MSP430FR58xx, MSP430FR59xx, and MSP430FR6xx Family User's Guide, (SLAU367O), Texas Instruments, 2017. 333, 334, 335, 337

MSP430FR5994 LaunchPad Development Kit (MSP-EXP430FR5994), (SLAU678A), Texas Instruments, 2016.

MSP430FR599x, MSP430FR596x Mixed-Signal Microcontrollers, (SLASE54C), Texas Instruments, 2018. 339

8.8 CHAPTER PROBLEMS

Fundamental

1. List three different types of resets in the MSP430 microcontroller.
2. State the purpose of resets and interrupts.
3. What is the main difference between a reset and an interrupt?
4. What is the differences between maskable and nonmaskable interrupts?
5. In addition to setting up a local interrupt enable bit, you must also set the global enable bit. Where is the global enable bit for all MSP430 maskable interrupts? Write an instruction to enable this global maskable interrupt enable bit.
6. What are the steps one must take to properly configure a maskable interrupt?
7. When more than one maskable interrupt occurs simultaneously, how does the MSP430 controller decide the order in which the controller service the interrupts?

Advanced

1. Why did the designers of MSP430 come up with three different types of resets?
2. Refer to the Interrupt Handling Process (Section 8.5.1) and explain the purpose for each of the ten steps.
3. Write a segment of code to initialize the MSP430 microcontroller to operate in the power save LPM3.5 mode and only operate in the normal mode during an interrupt.

Challenging

1. It is challenging to handle nested interrupts. What might be some applications where nested interrupts are necessary?
2. Consider the exercise in Section 8.6. Write a program for the central station that accepts the data from the temperature controller and log them in memory. Use an interrupt service routine since the power level data is only sent every 5 min.

CHAPTER 9

Analog Peripherals

Objectives: After reading this chapter, the reader should be able to:

- describe the function of an ADC and a DAC;
- explain the method used to perform analog conversions in the MSP430 microcontroller;
- configure the MSP430 microcontroller to accept analog signals and convert them into digital forms;
- describe the operation of the MSP430 comparators;
- use interrupts associated with the MSP430 microcontroller's ADC systems; and
- configure the MSP430 analog conversion process by writing an appropriate program.

We live in an analog world! When we represent physical phenomena as signals over time, the signal may have any of an infinite number of values. For example, if we display the pitch of your voice over time, it will be a continuous signal with your pitch varying over the period. The intensity of sunlight, wind speed, air temperature, the rate of a bird flapping its wings, and the speed of your car are all analog in nature. On the other hand, digital signals have a finite number of values over time. For a binary signal, the value is either logic one or zero, which computers use. To interact with the analog world, computers must have the capability to accept and generate analog signals.

In this chapter, we discuss the subsystems of a microcontroller that allows input of an analog signal: the MSP430's ADC system. We also explore the DAC process. We also describe the comparator system, which indicates whether a signal sample is within a set of defined voltage thresholds.

9.1 ANALOG-TO-DIGITAL CONVERSION PROCESS

Before a computer can process any physical signals, those signals must first be converted to their corresponding digital forms. Figure 9.1 shows an example of an analog signal. Notice that for a given time, t, shown on the x axis, the signal can hold any value of magnitude shown on the y axis. So, how do analog signals such as the one shown in the figure get converted into digital signals? The ADC process consists of the three separate sub-processes: sampling, quantization, and encoding.

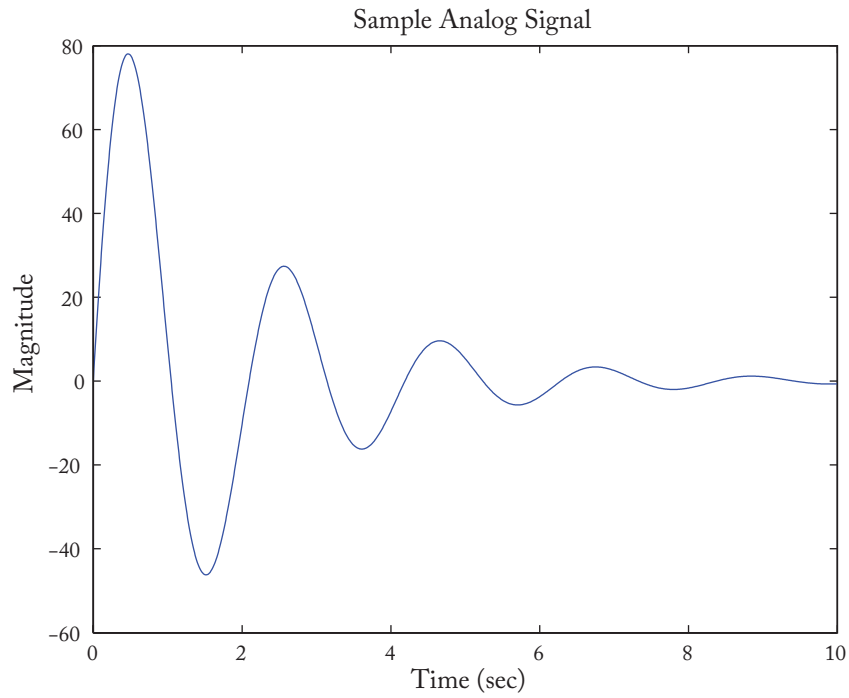


Figure 9.1: Sample analog signal.

9.1.1 SAMPLING

The sampling process allows for a digital system to capture an analog signal at a specific time. One can consider the sampling process as similar to taking snap shots of changing scenery using a camera.

Suppose we want to capture the movement of a baseball pitcher as he throws a ball toward home plate. Assume the only means for you to capture the motion of the pitcher is a camera. Suppose it takes 2 s for the pitcher to throw a baseball. If you take a picture at the start of the pitch and another one 2 s later, you have missed most of the action and will not be able to accurately represent the motion of the pitcher and the details of the pitch path.

The time between sampled snap shots is the period specified in seconds. The inverse of the period is the sampling frequency with the unit of Hertz (Hz). In this example, since there is a two second interval between samples, the sampling rate is $1/2 = 0.5$ Hz ($f = 1/T$). As you can imagine, the faster you take the pictures the more accurately you can recreate the pitcher's motion and the pitch path by sequencing photos.

The example illustrates the primary issue of the sampling process, that of the sampling frequency. A correct sampling frequency depends on the characteristics of the analog signal. If

the analog signal changes quickly, the sampling frequency must be high, while if the signal does not change rapidly the sampling frequency can be slow and one can still capture the essence of the incoming signal.

You may wonder what harm is there, then, in sampling at the highest possible rate, regardless of the frequency content of the analog signal? Just as it would be a waste of resources to take multiple pictures of the same stationary object, it would not be a good use of resources to sample with a high frequency rate regardless of the nature of an analog signal. In the 1940s, Henry Nyquist, who worked at Bell Laboratory, developed the concept that the minimum required sampling rate to capture the essence of an analog signal is a function of the highest input analog signal frequency: $f_s \geq 2 \times f_h$. The frequency f_s and f_h are the sampling frequency and the highest frequency of the signal we want to capture, respectively. That is, the sampling frequency must be greater than or equal to two times the highest frequency component of the input signal. We illustrate the Nyquist sampling rate using Figure 9.2.

Figure 9.2a shows the analog signal of interest, a sinusoidal signal. Frame (b) of the figure shows sampling points with a rate slower than the Nyquist rate and frame (c) shows the reconstruction of the original signals using only the sampled points shown in frame (b). Frame (d) shows the sampled points at the Nyquist rate and the corresponding reconstructed signal is shown in frame (e). Finally, frame (f) shows sampled points at a rate higher than the Nyquist sampling rate and frame (g) shows the reconstructed signal. As can be seen from this simple example, when we sample an analog signal at the Nyquist sampling rate, we can barely generate the characteristics of the original analog signal. While at a higher sampling rate, we can retain the nature of an input analog signal more accurately, but at a higher cost, requiring a faster clock, additional processing power, and more storage for the accumulated data. Let us examine one final example before we move on to the quantization process.

Example: An average human voice contains frequencies ranging from about 200 Hz to 3.5 kHz. What should be the minimum sampling rate for an ADC system?

Answer: According to the Nyquist sampling rate rule, we should sample at 3.5 kHz $\times$ 2 = 7 kHz, which translates to taking a sample every 142.9 usec. Your telephone company uses a sampling rate of 8 kHz to sample your voice.

9.1.2 QUANTIZATION

Once a sample is captured, then the second step of the conversion process, quantization, can commence. Before we explain the process, we first need to define the term *quantization level*. Suppose we are working with an analog voltage signal whose values can change from 0–5 V. Now suppose we pick a point in time. The analog signal, at that point in time, can have any value between 0–5 V, an infinite number of possibilities (think of real numbers). Since we do not have a means to represent an infinite number of different values in a digital system, we limit the possible values to a finite number. So, what should this number be? Previously, we saw that

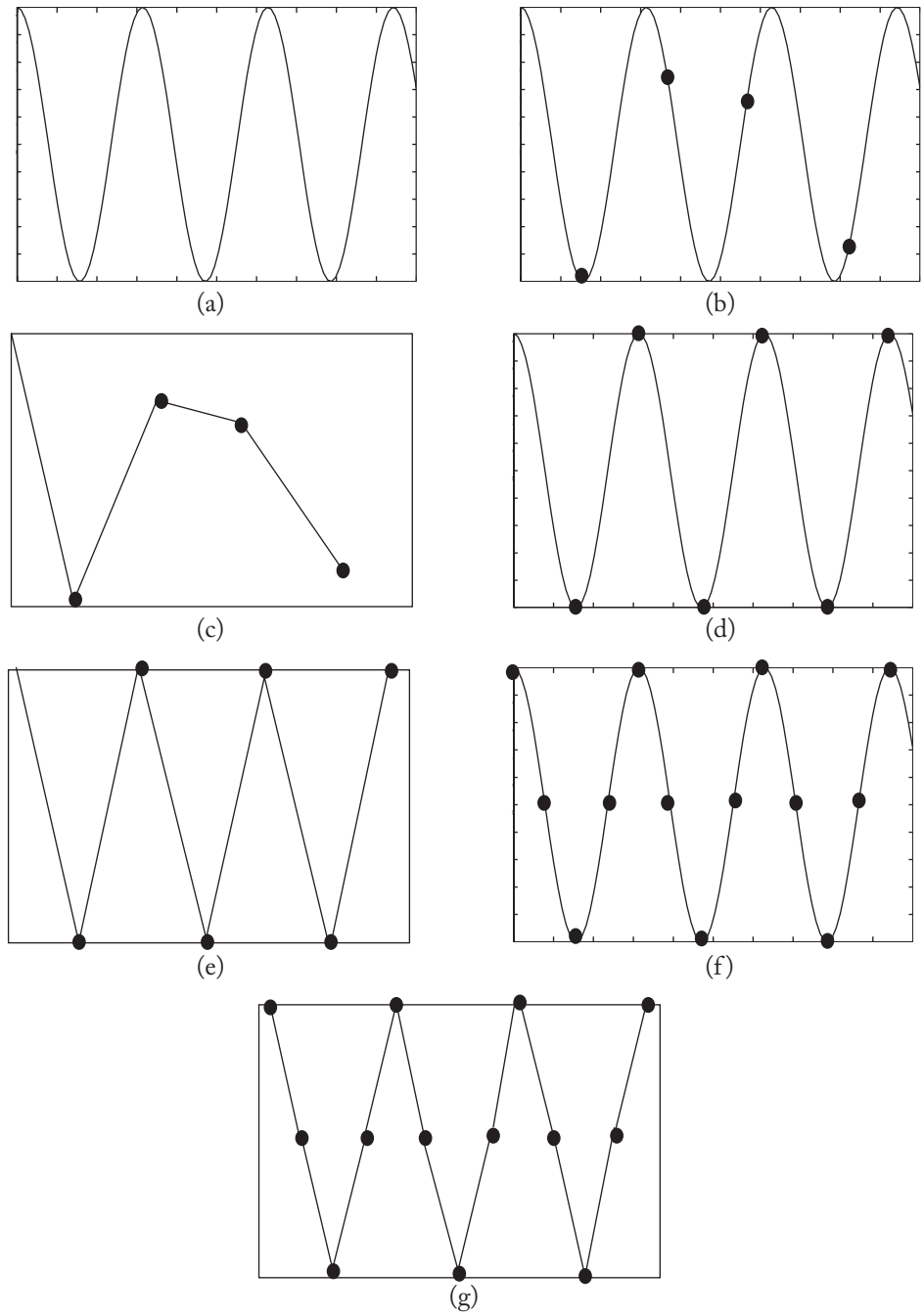


Figure 9.2: Sampling rate illustration.

there are 2^b number of values we can represent with b bits. If we have a 2-bit analog-to-digital converter, 2 bits are used to represent the analog signal; there are four different representations we can choose from. In a 4-bit converter, we have 16 different ways to do so. If we have an 8-bit converter, we have 256 different representations, and so on. Figure 9.3 shows both quantization levels and the corresponding resolution for a sample input signal for a 3-bit ADC.

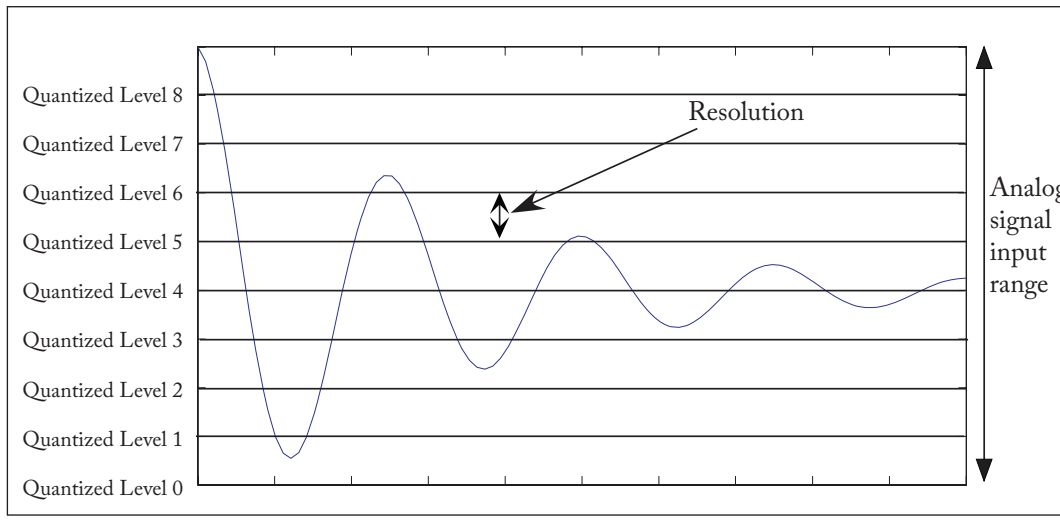


Figure 9.3: Quantization levels and the resolution of an ADC.

As you may suspect, there is a tradeoff between using a large number of bits for an accurate representation vs. the hardware cost of designing and manufacturing a converter. A converter which employs more bits will yield more accurate representations of the sampled signal values. A decision made by an ADC designer determines the accuracy of a sampled data and the cost to manufacture the converter. The number of bits used to quantize a sampled value determines the available quantization levels. Therefore, a converter which uses 8 bits has 256 quantization levels while a converter that uses 10 bits has 1024 quantization levels.

We need to also define what is known as the resolution of an ADC. Simply put, the resolution is the smallest quantity a converter can represent or the "distance" between two adjacent quantization levels. The resolution will naturally vary depending on the range of input analog signal values. Suppose the input range is from 0–5 V, a typical input range of an ADC, and we have an 8-bit converter. The resolution, δ, is then

$$\delta = \frac{\textit{analog input highest value} - \textit{analog input lowest value}}{2^b} = \frac{5-0}{256} = 19.5312 \text{ mV}.$$

Example: Given an application that requires an input signal range of 10 V and the resolution less than 5 mV, what is the minimum number of bits required for the ADC?

Answer: $\frac{10-0}{2^{10}} = 9.77$ mV and $\frac{10-0}{2^{11}} = 4.88$ mV. Thus, the minimum required number of bits for the ADC is 11 bits.

We can now combine both sampling and quantization processes together to identify a quantized level of a sampled value. Suppose a sampled analog signal value is 3.4 V and the input signal range is 0–5 V. Using a converter with 8 bits, we can find the quantized level of the sampled value using the following equation:

$$\textit{Quantized level} = \frac{\textit{sampled input value} - \textit{lowest possible input value}}{\delta}.$$

Thus, given the input sample value of 3.4 V, the quantized level becomes $\frac{3.4\text{ V}-0\text{ V}}{19.53\text{ mV}} \cong 174.09$. Since we can only have integer levels, the quantized level becomes 174.

The sampling error is the difference between the true analog value and the sampled value. It is the amount of approximation the converter had to make. See Figure 9.4 for a pictorial view of this concept. For the example, the input sampled value 3.4 V is represented as the quantized level 174 and the quantized error is $0.09 \times \delta = 1.76$ mV. Note that the maximum quantization error is the resolution of the converter.

Example: Given a sampled signal value of 7.21 V, using a 10-bit ADC with input range of 0 V and 10 V, find the corresponding quantization level and the associated quantization error.

Answer: First, we find the quantized level:

$$\textit{Quantized level} = \frac{7.21 - 0}{\delta},$$

where $\delta = \frac{10}{2^{10}} = 9.77$ mV. Thus, the quantized level is 738.3059. Since we always round down, the quantized level is 738 and the associated quantization error is $0.3059 \times 9.77\text{ mV} \cong 2.987$ mV.

9.1.3 ENCODING

The last step of the ADC process is the encoding. The encoding process converts the quantized level of a sampled analog signal value into a binary number representation. Consider the following simple case. Suppose we have a converter with four bits. The available quantization levels for this converter are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, and 15. Using 4 bits, we can represent the quantization levels in binary as 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, and 1111. Once we identify a quantization level, we can uniquely represent the quantization level as a binary number. This process is called encoding. Similar to a decimal number, the position of each bit in a binary number represents a different value. For example, binary number 1100 is decimal number $(1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (0 \times 2^0) = 12$. Knowing the weight of each bit, it is a straight forward process to represent a decimal number as a binary number.

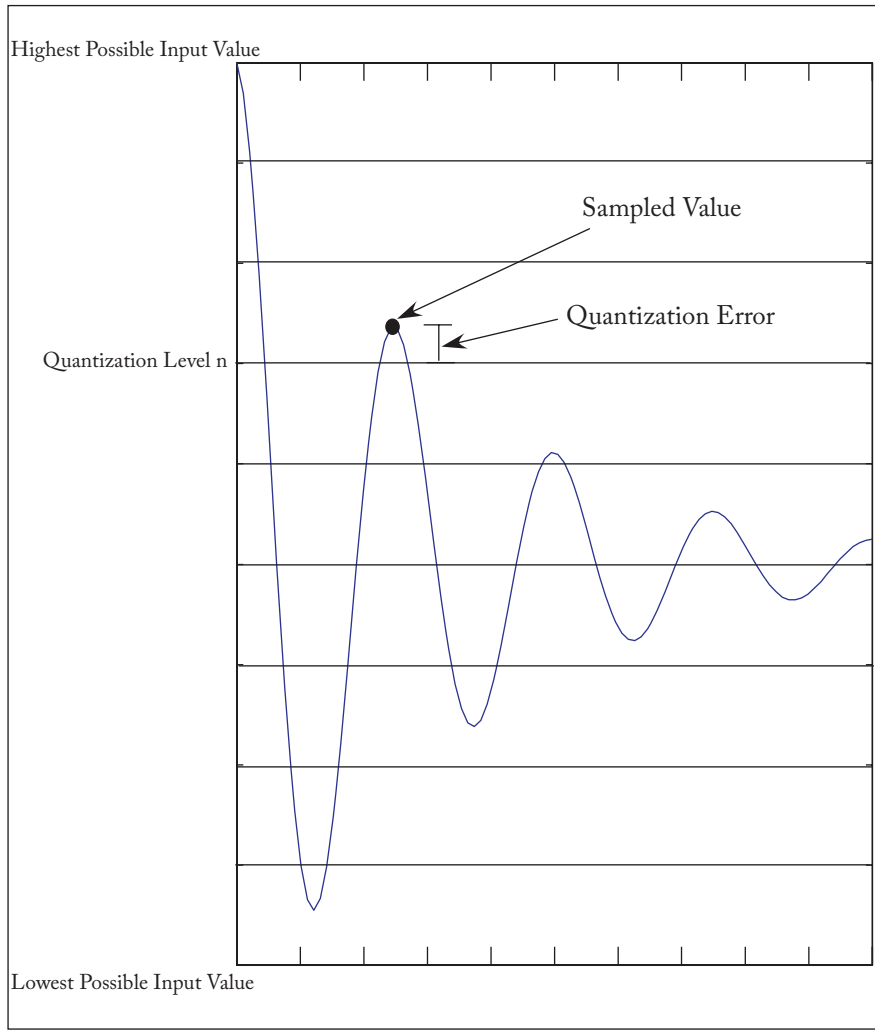


Figure 9.4: Quantized error of a sampled input signal.

Example: Find the encoded value of the quantization found in the previous example: 738. Recall we are using 10 bits.

Answer: Since we are using 10 bits to represent this number, the encoded value is
$(1 \times 2^9) + (0 \times 2^8) + (1 \times 2^7) + (1 \times 2^6) + (1 \times 2^5) + (0 \times 2^4) + (0 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) = 738$.
Thus, the encoded value is 1011100010.

9.2 DIGITAL-TO-ANALOG CONVERTER PROCESS

The opposite function of an ADC is performed by a DAC. Some variants of the MSP430 are equipped with an onboard DAC. Neither the MSP430FR2433 or the MSP430FR5994 microcontrollers are equipped with an onboard DAC. However, the theory behind a DAC system is important to understand. This allows the selection of an appropriate external DAC for a given application.

The input to a DAC converter is an encoded value which specifies the desired output analog value. Similar to the ADC, a DAC must have both minimum and maximum reference analog voltages. The job of a DAC is then to map a minimum digital representation to its corresponding minimum analog value, a maximum digital representation to the maximum reference analog value, and representations in between minimum and maximum digital values to their appropriate analog counterparts. The most common method used to perform a DAC conversion is to pre-designate the analog weight of each bit in the digital input representation and then sum up the contributions to form an analog output. For example, suppose the range of output values for a DAC is from 0–5 V. If we have a 4-bit DAC, from the most to the LSBs, each specific bit would be weighted 2.5 V, 1.25 V, 0.625 V, and 0.3125 V, respectively. Thus, a digital input of 1010 to this converter will result in $2.5 + 0.625 = 3.125$ V, and a digital input of 1111 to the DAC converter would result in $2.5 + 1.25 + 0.625 + 0.3125 = 4.6875$ V. Given an N-bit converter, it is straightforward to develop the following equation to describe the relationships among the input, the number of bits used, and the output:

$$Analog\ output = \frac{digital\ input}{2^N} V_{refmax},$$

where N stands for the number of bits used in the converter and V_{refmax} represents the maximum analog reference voltage of the converter. The DAC uses the summing technique where each of the input digital bits asserts a switch to turn on its associated weighted voltage. All voltages are summed to generate an output analog voltage. Figure 9.5 illustrates a 4-bit DAC converter with the maximum and minimum output values of 2.5 V and 0 V, respectively.

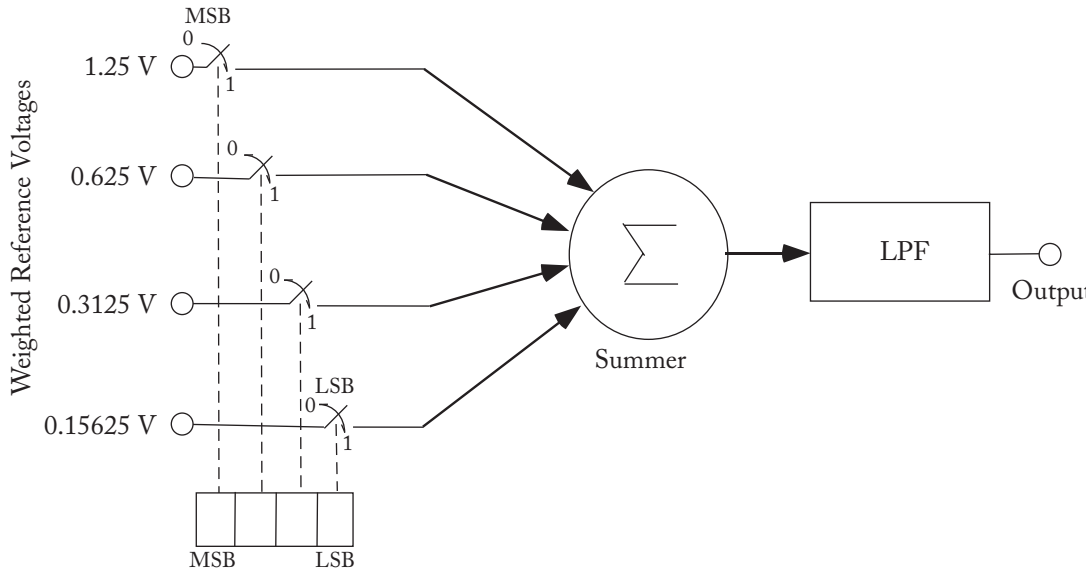


Figure 9.5: A sample 4-bit digital to analog converter. The DAC digital input governs the positions of the switches. The digital input values determine if the corresponding reference voltage values should contribute to the converter output value. The output of the summer is typically connected to a low pass filter (LPF) to reduce sharp signal edges resulting from the conversion process.

9.3 MSP430 ADC SYSTEMS

In this section we begin with a general discussion of the MSP430 ADC block diagram followed by a detailed discussion of the MSP430FR2433 ADC and the MSP430FR5994 ADC and comparator systems.

9.3.1 MSP 430 ADC BLOCK DIAGRAM

A basic block diagram of the MSP430 ADC system is shown in Figure 9.6a. An input analog channel is selected for conversion by the input voltage select multiplexer (mux). The selected signal is held constant by the sample and hold (S/H) circuitry during the conversion process. The stable signal is then fed to the successive approximation converter. The SA converter receives input from the reference voltage select, the timing source, and trigger source for conversion.

The digital result of the conversion, provided as n bits, is stored in the result register. Specific interrupts may be selected to signal different significant events in the ADC process.

A block diagram of SA converter operation is provided in Figure 9.6b. As its name implies, the SA converter will make successive guesses at the unknown sample voltage value. It begins with a guess of one-half of the reference voltage, as shown in Figure 9.6c. This digital guess is converted to a corresponding analog value by the DAC. The analog guess is compared to the unknown sample voltage by the voltage comparator. The output from the comparator prompts the SA to guess higher or lower. This process continues n times (one for each bit in the SA register). The guess progresses from one-half of the reference voltage to one-fourth to one-eighth, etc. When the conversion is complete, the end of conversion signal goes logic high. As we present a detailed discussion of the MSP430FR2433 and MSP430FR5994 ADC systems, it may be helpful to refer back to the block diagrams.

9.3.2 MSP430FR2433 10-BIT ANALOG-TO-DIGITAL CONVERTER

The MSP430FR2433 is equipped with a flexible and powerful ADC system. It has the following features [SLAU445G, 2016]:

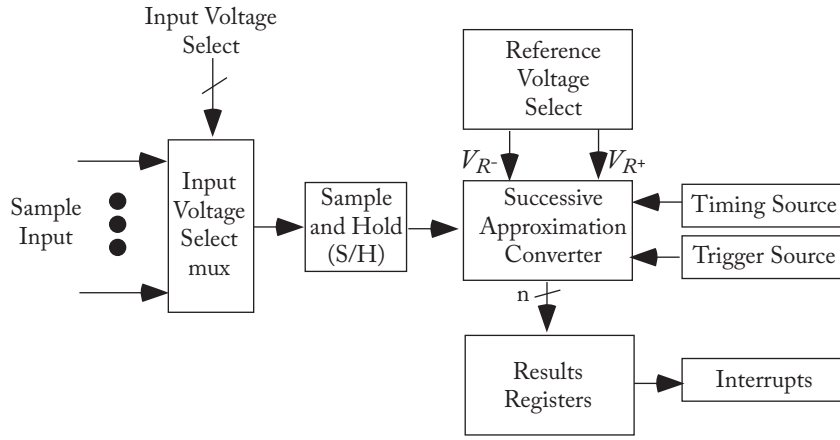
- eight channels for external conversion,
- converter resolution of 10-bits per sample,
- successive approximation converter,
- selectable reference voltages, and
- input voltage range from 0 to DVcc.

The ADC is quite flexible and may be configured for the following operations:

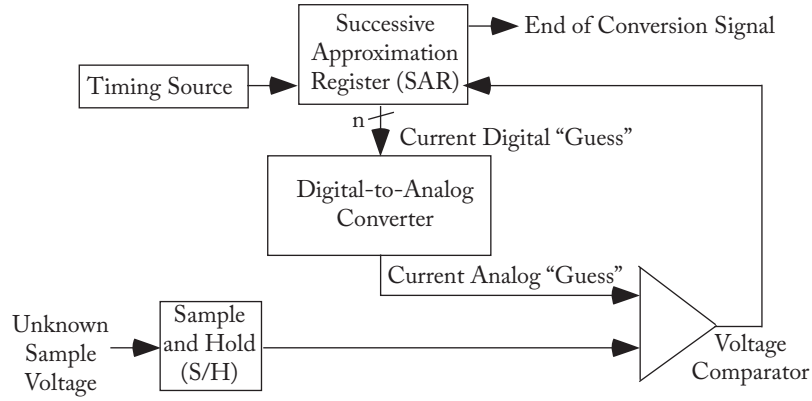
- a single conversion on a single ADC channel,
- a single conversion on a sequence of channels,
- a repeated conversion on a single channel, or
- a repeated conversion on a sequence of channels.

The block diagram of the MSP430FR2433 ADC is shown in Figure 9.7. Like the block diagram provided in Figure 9.6, the MSP430FR2433 ADC system may be divided into input select, voltage reference select, time base select, trigger source select, and the output processing sections. The ADC process is configured and monitored by a set of ADC registers.

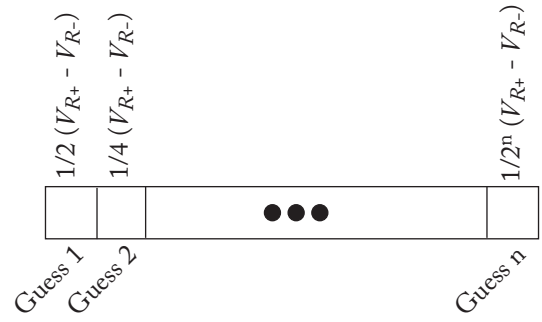
Input select. There are 16 different inputs to the converter as shown in the left of Figure 9.7: A0–A15. Of the 16 inputs, 8 inputs (A0–A7) are used to route analog input signals to the converter. Inputs A14 and A15 are connected to a positive reference voltage and a negative



(a) Successive approximation ADC block diagram



(b) Successive approximation ADC converter



(c) Successive approximation register (SAR)

Figure 9.6: Basic ADC block diagram.

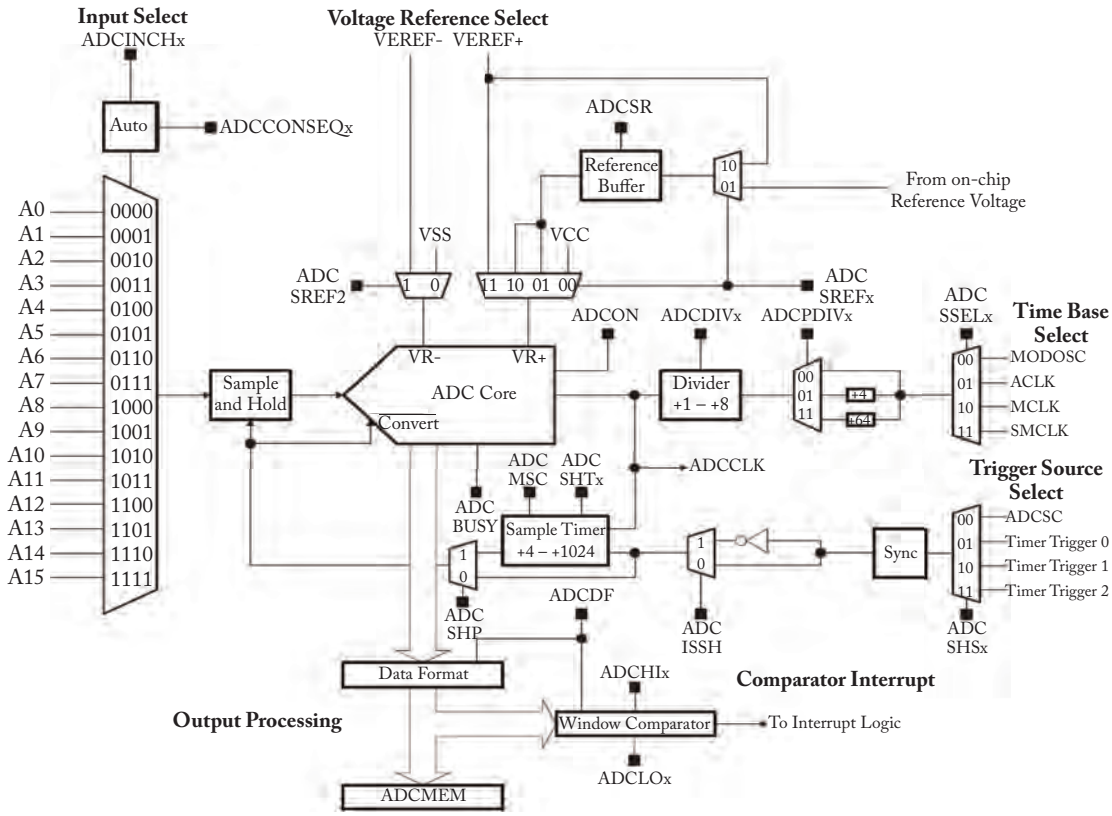


Figure 9.7: Block diagram of the MSP430 10-bit analog-to-digital converter. (Illustration used with permission of Texas Instruments (www.ti.com).)

reference voltage, which can be supplied by a user with external voltage sources or using the MSP430 internal reference voltage generator module.

Input A12 is connected to a built-in temperature diode which measures the environmental temperature of the controller (useful for a variety of meters where MSP430 controllers are used), and input A13 is connected to a DC voltage whose value is 0.5 * (AVcc - AVss), if required for an application.

Once an analog signal (or multiple signals) enters the converter, bits ADCINCHx (bits 3-0) of the ADC memory control register (ADCMCTLx) determine which input channel or a starting channel of a group of input channels will be used in the converter.

Reference voltage select. The ADCSREFx bits (bits 6-4) in the ADC memory control register x (ADCMCTLx) are used to select one of eight possible voltage reference selections as shown in Table 9.1. The ADC uses 10 bits to sample analog signals, providing 1024 different

Table 9.1: Voltage reference

ADCSREFx	Reference Voltages
001	$V_{R+} = AV_{cc}$ and $V_{R-} = AV_{ss}$
001	$V_{R+} = V_{REF+}$ and $V_{R-} = AV_{ss}$
010	$V_{R+} = V_{eREF+}$ and $V_{R-} = AV_{ss}$
011	$V_{R+} = V_{eREF+}+$ and $V_{R-} = AV_{ss}$
100	$V_{R+} = AV_{cc}$ and $V_{R-} = V_{REF-}/V_{eREF-}$
101	$V_{R+} = V_{REF+}$ and $V_{R-} = V_{REF-}/V_{eREF-}$
110	$V_{R+} = V_{eREF+}$ and $V_{R-} = V_{REF-}/V_{eREF-}$
111	$V_{R+} = V_{eREF+}$ and $V_{R-} = V_{REF-}/V_{eREF-}$

quantized levels (000h - 3FFh) to represent a sample value. If an input voltage value is equal to reference low (REF−) value, 000h results while an input voltage equal to reference high (REF+) voltage is represented as 3FFh. The results are encoded either as unsigned or 2'complement binary values as specified by the ADC read back format (ADCDF) bit in ADC Control 2 (ADCCTL2) register.

Clock source select. ADC clock source selection is determined by the ADCSSELx bits (bits 4-3) of the ADC control 1 (ADCTL1) register. Selecting these bits determines whether the source clock is MODOSC (00), ACLK (01), MCLK (10), or SMCLK (11). The selected clock source can be divided by two different stages: the ADC predivider (ADCPDIV bits) and the ADC clock divider. The ADCPDIV bits (bits 9-8) of ADC control 2 (ADCCTL2) register provide for a divide by 1 (00), 4 (01), or 64 (10) of the selected clock source. The ADCDIVx bits (bits 7-5) in the ADCCTL1 register allow for an additional divide factor of 1 (000) to 8 (111).

ADC interrupts. Monitoring the ADC system is a series of six interrupts to flag different events during ADC operation. These monitoring interrupts are normally off and must be individually enabled as shown in Figure 9.8. Also, as with all interrupts, the GIE bit in the SR must also be enabled. If a specific interrupt is enabled for a given application, a corresponding interrupt service routine (ISR) must be provided.

The six ADC interrupts are [SLAU445G, 2016]:

- ADC memory interrupt flag (ADCIFGO)
- ADCMEM0 overflow (ADCOVIFG)
- Conversion time overflow (ADCTOVIFG)
- ADCLO interrupt flag (ADCLOIFG)

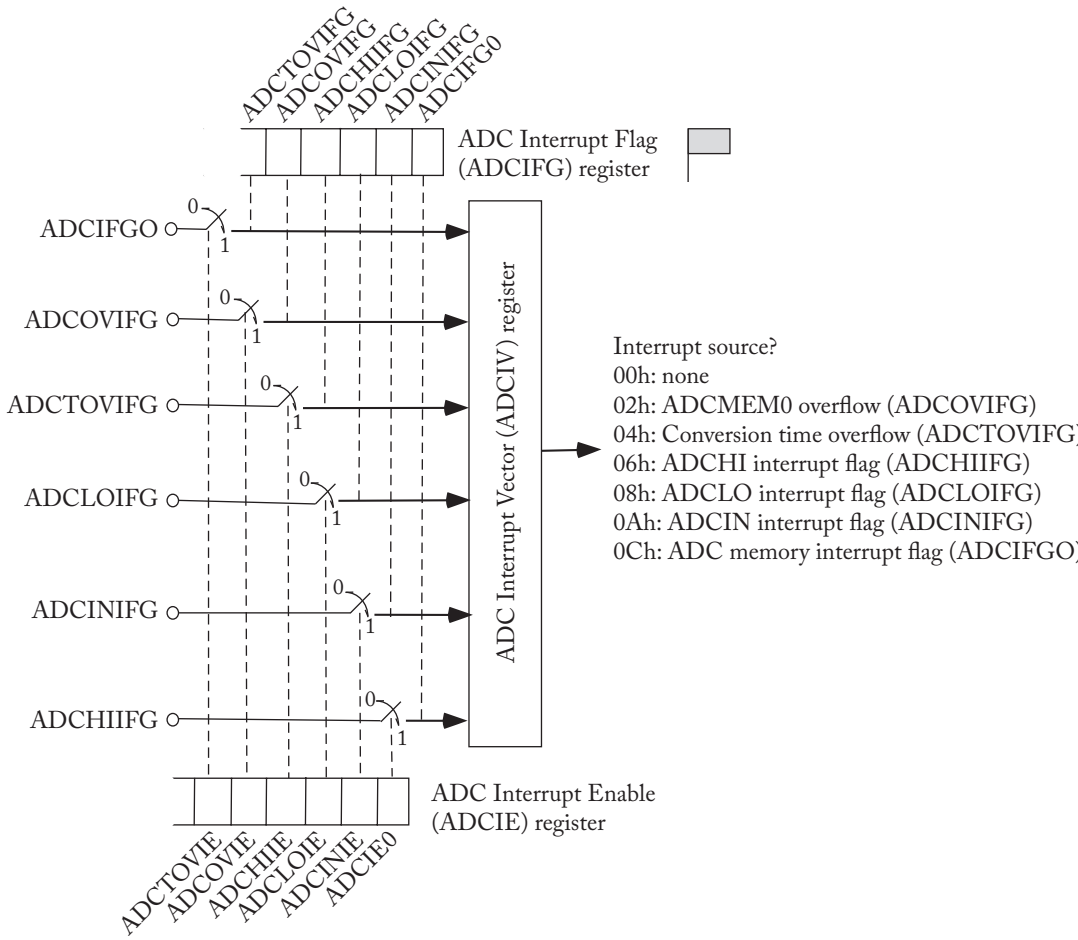


Figure 9.8: MSP430FR2433 ADC interrupts overview.

- ADCIN interrupt flag (ADCINIFG)
- ADCHI interrupt flag (ADCHIIFG)

As shown in Figure 9.8, a specific interrupt must be enabled by asserting its corresponding bit in the ADC interrupt enable register (ADCIE). When an enabled interrupt event occurs, the corresponding flag bit is set in the ADC interrupt flag (ADCIFG) register. The numerical value of the highest priority active interrupt may be read from the ADC interrupt vector (ADCIV) register.

All ADC interrupt sources are provided to the ADC interrupt vector (ADCIV) register. When an ADC interrupt(s) occurs, the interrupt source are sent to the ADCIV register where it is prioritized.

Results register. The result(s) of sampled and converted signal(s) is stored in the ADC conversion memory (ADCMEM0) register. Prior to storage in the result register, the result is formatted to a user specified configuration. Also, if the comparator features are asserted, appropriate interrupts are set.

9.3.3 MSP430FR2433 REGISTER SUMMARY

ADC operation and feature selection are determined by user selected settings in the ADC registers. The ADC register set includes the following registers [SLAU445G, 2016]:

- ADCCTL0 ADC Control 0 register
- ADCCTL1 ADC Control 1 register
- ADCCTL2 ADC Control 2 register
- ADCMCTL0 ADC Memory Control register
- ADCMEM0 ADC Conversion Memory register. Conversion results are stored here.
- ADCIE ADC Interrupt Enable register. Used to enable individual interrupts.
- ADCIFG ADC Interrupt Flag register. Corresponding flags for individual ADC interrupts are provided here.
- ADCIV ADC Interrupt Vector register. Provides numerical value of active interrupt.

Details of configuring each register is provided in SLAU445G. We highlight the settings of the ADC Control Registers in Figure 9.9.

Programming the MSP430FR2433 ADC in Energia

The Energia library contains several functions to support analog conversions including ADC- and DAC-related functions (`www.energia.nu`):

- **analogRead():** The analogRead function performs an ADC conversion on the indicated analog pin. The measured voltage is converted to an integer value between 0 and 1023, where 0 corresponds to 0 VDC and 1023 corresponds to 3.3 VDC.
- **analogReference():** The analogReference function provides for changing the high level reference voltage for ADC conversion. The different settings include:
  - DEFAULT: sets ADC high reference level to VCC 3.3 V.

15	14	13	12	11	10	9	8
Reserved				ADCSHTx			
r0	r0	r0	r0	rw-(0)	rw-(0)	rw-(0)	rw-(1)
7	6	5	4	3	2	1	0
ADCMSC	Reserved		ADCON	Reserved		ADCENC	ADCSC
rw-(0)	r0	r0	rw-(0)	r0	r0	rw-(0)	rw-(0)

(a) ADC Control 0 register (ADCCTL0)
ADCCTL0[4]: ADC on: ADCON: 0 = ADC off, 1 = ADC on
ADCCTL0[1]: ADC enable conversion: ADCENC: 0 = disabled, 1 = enabled
ADCCTL0[0]: ADC start conversion: ADCSC: 0 = no conversion, 1 = start conversion

15	14	13	12	11	10	9	8
Reserved				ADCSHSx		ADCSHP	ADCISSH
r0	r0	r0	r0	rw-(0)	rw-(0)	rw-(0)	rw-(0)
7	6	5	4	3	2	1	0
ADCDIVx			ADCSSELx		ADCCONSEQx		ADCBUSY
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	r-(0)

(b) ADC Control 1 register (ADCCTL1)
ADCCTL1[7-5]: ADC clock divider: ADCDIVx: 000 = 1 to 111 = 8
ADCCTL1[4-3]: ADC clock source select: ADCSSELx: MODCLK (00), ACLK (01), MCLK (10), SMCLK (11)
ADCCTL1[2-1]: ADC conversion sequence: ADCCONSEQx:
single channel, single conversion (00): sequence of channels, single conversion (01),
single channel, repeat conversion (10): sequence of channels, repeat conversion (11)
ADCCTL1[0]: ADC busy: 0 = inactive, 1 = active

15	14	13	12	11	10	9	8
Reserved						ADCPDIVx	
r0	r0	r0	r0	r0	r0	rw-(0)	rw-(0)
7	6	5	4	3	2	1	0
Reserved		ADCRES		ADCDF	ADCSR	Reserved	
r0	r0	rw-(0)	rw-(1)	rw-(0)	rw-(0)	r0	rw-(0)

(c) ADC Control 2 register (ADCCTL2)
ADCCTL2[9-8]: ADC predivider: ADCPDIVx: 00 = 1, 01 = 4, 10 = 64
ADCCTL2[5-4]: ADC resolution: ADCRES: 00 = 8 bit, 01 = 10 bit, 10 = 12 bit
ADCCTL2[3]: ADC read back format: ADCDF: 0: unsigned binary, 1= 2's complement signed binary
ADCCTL2[2]: ADC sampling rate: ADCSR: 0 = 200 ksps, 1 = 50 ksps

15	14	13	12	11	10	9	8
Reserved							Reserved
r0	r0	r0	r0	r0	r0	r0	rw-(0)
7	6	5	4	3	2	1	0
Reserved	ADCSREFx			ADCINCHx			
r0	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)

(d) ADC Conversion Memory Control Register (ADCMCTL0)
ADCSREFx [6-4]: Voltage reference select
ADCINCHx [3-0]: Input channel select 0000 = A0 through 1111 = A15

Figure 9.9: MSP430FR2433 ADC control registers [SLAU445G, 2016]. (Illustration used with permission of Texas Instruments (www.ti.com).)

  - INTERNAL1V5: sets ADC high reference level to internal 1.5 VDC reference.
  - INTERNAL2V5: sets ADC high reference level to internal 2.5 VDC reference.
  - EXTERNAL: sets ADC high reference level to the VREF pin value.

- **map:** As its name implies the map function maps a range of integers (fromLow, fromHigh) to a new range of integers (toLow, toHigh).

- **analogWrite:** The analogWrite function generates a pseudo analog output signal using a pulse width modulated signal. The analogWrite function generates a 490 Hz signal on the specified pin with a duty cycle specified from 0–255.

9.3.4 PROGRAMMING THE MSP430FR2433 ADC IN C

To successfully configure and program the converter, the following steps should be followed.

1. ADC pins are multiplexed with other I/O functions. Configure (set to logic one) the proper field of the MSP430FR2433 System Configuration Register 2 (SYSCFG2) for ADC access.
2. Select the desired ADC reference voltage via ADCSREFx bits in the ADCMCTLx register.
3. Connect analog signal(s) for conversion to appropriate input pins (A0-A7).
4. Turn on the ADC: ADCON in ADCCTL0.
5. Select the clock source and the sampling mode in ADCCTL1.
6. Configure the converter for proper operation: single channel, single conversion; single channel, multiple conversion; multiple channels, single conversion; or multiple channels, multiple conversions using ADCCONSEQx bits in ADCTL1.
7. Initiate conversion: ADCENC bit in ADCCTL0.
8. Monitor for conversion completion using the ADCIFG0 flag.
9. Use the results of the conversion located in the corresponding result register (ADC12MEM0).
10. Repeat the process starting at step 6.

Example: In this example, we show how to configure the analog-to-digital converter for a single-channel, single-conversion mode. A single sample is made on input A1 with default reference to AVcc. In the mainloop, the MSP430 waits in LPM0 to save power until the ADC conversion is

complete. The ADC_ISR will force exit from low power mode LPM0. If A1 > 0.5*AVcc, P1.0 set, else P1.0 is reset.

Note how the #if, #elif, #else, and #endif directives are used to implement conditional compilation. This allows the same interrupt service routine to be used by the TI, IAR, or the GNU compiler.

```
//******************************************************************
// --COPYRIGHT--,BSD_EX
// Copyright (c) 2014, Texas Instruments Incorporated
// All rights reserved.
//
//                  MSP430 CODE EXAMPLE DISCLAIMER
//
//******************************************************************
//MSP430FR24xx Demo - ADC, Sample A1, AVcc Ref, Set LED
//                     if A1 > 0.5*AVcc
//
//Description: This example works on Single-Channel Single-Conversion
//Mode.  A single sample is made on A1 with default reference to AVcc.
//Software sets ADCSC to start sample and conversion.  ADCSC
//automatically cleared at EOC. ADC internal oscillator times sample
//(16x) and conversion.
//In mainloop MSP430 waits in LPM0 to save power until the ADC
//conversion complete, ADC_ISR will force exit from LPM0 in mainloop
//on reti.  If A1 > 0.5*AVcc, P1.0 set, else reset.
//
//ACLK = default REFO ~32768Hz, MCLK = SMCLK = default DCODIV ~1MHz.
//
//               MSP430FR2433
//            -----------------
//        /|\|                 |
//         | |                 |
//         --|RST              |
//           |                 |
//       >---|P1.1/A1      P1.0|--> LED
//
//
//Wei Zhao, Texas Instruments Inc., Jan 2014
//Built with IAR Embedded WB v6.20 & Code Composer Studio v6.0.1
//******************************************************************
```

```
#include <msp430.h>

unsigned int ADC_Result;

int main(void)
{
WDTCTL = WDTPW | WDTHOLD;                 //Stop WDT

                                          //Configure GPIO
P1DIR |= BIT0;                            //Set P1.0/LED to output
P1OUT &= ~BIT0;                           //P1.0 LED off

SYSCFG2 |= ADCPCTL1;                      //Configure ADC A1 pin


//Disable the GPIO power-on default high-impedance mode to activate
//previously configured port settings
M5CTL0 &= ~LOCKLPM5;
                                        //Configure ADC10
ADCCTL0 |= ADCSHT_2 | ADCON;            //ADCON, S&H=16 ADC clks
ADCCTL1 |= ADCSHP;                      //ADCCLK = MODOSC, sampling timer
ADCCTL2 |= ADCRES;                      //10-bit conversion results
ADCMCTL0 |= ADCINCH_1;                  //A1 ADC input select, Vref=AVC
ADCIE |= ADCIE0;                        //Enable ADC conv complete
                                        //interrupt
while(1)
  {
  ADCCTL0 |= ADCENC | ADCSC;            //Sampling and conversion start
  bis_SR_register(LPM0_bits | GIE);     //LPM0, ADC_ISR will force exit
  __no_operation();                     //For debug only
  if(ADC_Result < 0x1FF)
    P1OUT &= ~BIT0;                     //Clear P1.0 LED off
  else
    P1OUT |= BIT0;                      //Set P1.0 LED on
    __delay_cycles(5000);
  }
}
```

```
//******************************************************************
// ADC interrupt service routine
//******************************************************************
#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector=ADC_VECTOR

__interrupt void ADC_ISR(void)
#elif defined(__GNUC__)

void __attribute__ ((interrupt(ADC_VECTOR))) ADC_ISR (void)
#else
#error Compiler not supported!
#endif
{
  switch(__even_in_range(ADCIV,ADCIV_ADCIFG))
    {
    case ADCIV_NONE:      break;
    case ADCIV_ADCOVIFG:  break;
    case ADCIV_ADCTOVIFG: break;
    case ADCIV_ADCHIIFG:  break;
    case ADCIV_ADCLOIFG:  break;
    case ADCIV_ADCINIFG:  break;
    case ADCIV_ADCIFG:    ADC_Result = ADCMEM0;
                          //Clear CPUOFF bit from LPM0
                          __bic_SR_register_on_exit(LPM0_bits);
                          break;
    default:              break;
    }
}

//******************************************************************
```

9.4 MSP430FR5994 ANALOG-TO-DIGITAL CONVERTER

In this section we introduce the MSP430FR5994 system ADC referred to as ADC12_B. We describe its features, operation, registers, and conclude with several programming examples.

9.4.1 ADC12_B FEATURES

The ADC12_B system is a flexible and powerful ADC system with an extensive list of features [SLAU367O, 2017].

- ADC12_B provides 14-bits of ADC resolution. Recall, the resolution of a converter provides 2^b number of incremental steps between the high and low reference voltages.
- ADC12_B is a SAR type converter. A SAR converter takes the same amount of time for converting an unknown voltage regardless of its magnitude. We discuss the operation of a SAR type converter in the next section.
- The maximum conversion rate of ADC12_B is 200 kilo samples per second (ksps).
- ADC12_B is equipped with 32 individual input channels. The inputs may be configured for single-ended conversion where the input signal is referenced to ground. The inputs may also be configured for differential input. In this type of conversion, two signals are subtracted from one another and their difference is converted. This is especially useful in noisy environments. Signals that are common to both inputs (noise) are canceled and the actual signal is amplified.
- Specific internal signals within the MSP432 processor may be selected for ADC conversion.
- The ADC12_B may be set to provide conversion on a single channel, multiple conversions of a single channel, a single conversion of a sequence of channels, or multiple conversions of a sequence of channels.
- ADC12_B is supported by a variety of interrupts.

9.4.2 MSP430FR5994 ADC12_B OPERATION

A basic block diagram of ADC12_B is shown in Figure 9.6a. An input analog channel is selected for conversion by the input voltage select multiplexer (mux). The selected signal is held constant by the sample and hold (S/H) circuitry during the conversion process. The stable signal is then fed to the SA converter. The SA converter receives input from the reference voltage select, the timing source, and trigger source for conversion. The digital result of the conversion, provided as n bits, is stored in result registers. Specific interrupts may be selected to signal different significant events in the ADC process.

A block diagram of SA converter operation is provided in Figure 9.6b. As its name implies, the SA converter will make successive guesses at the unknown sample voltage value. It begins with a guess of one-half of the reference voltage. This digital guess is converted to a corresponding analog value by the DAC. The analog guess is compared to the unknown sample voltage by the voltage comparator. The output from the comparator prompts the SA to guess higher or lower. This process continues n times (one for each bit in the SA register). The guess progresses from one-half of the reference voltage to one-fourth to one-eighth, etc. as shown in Figure 9.6c. When the conversion is complete, the end of conversion signal goes logic high.

The detailed block diagram of ADC12_B is provided in Figure 9.10. It may appear a bit overwhelming at first; however, it is simply a more detailed version of the basic block diagram you have seen in Figure 9.6. The operation of the ADC12_B is configured and controlled by registers ADC12CTL0 through ADC12CTL4. The bit designators from these registers are shown at various points on the diagram.

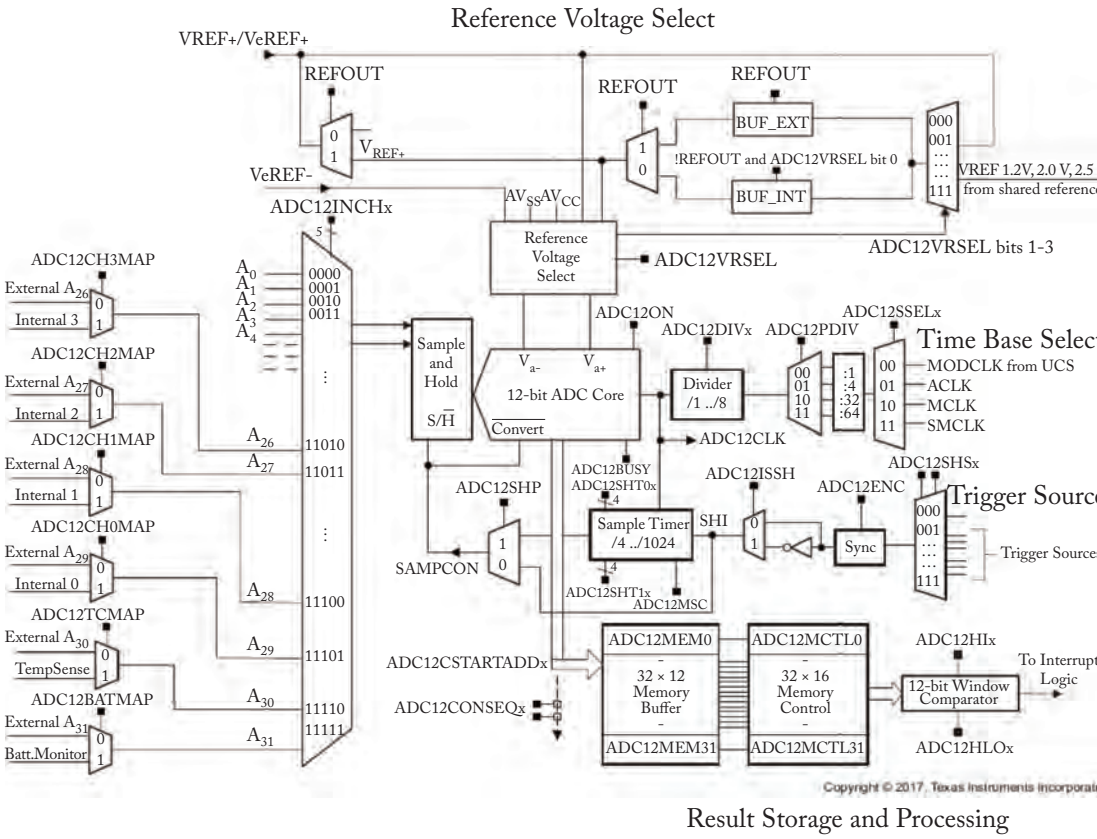


Figure 9.10: ADC12_B block diagram. (Illustration used with permission of Texas Instruments (www.ti.com).)

As seen in the figure, an input analog channel is selected for conversion by the input voltage select multiplexer by the ADC12INCHx bits. The selected signal is held constant by the sample and hold (S/H) circuitry during the conversion process. The stable signal is then fed to the SA converter. The SA converter receives input from the reference voltage select, the timing source, and trigger source for conversion. The specific reference voltage is selected by the ADC12VRSEL bits. The specific timing source (MODCLK, SYSCLK, ACLK, MCLK, SMCLK, or HSMCLK) is selected by the ADC12SSELx bits. The selected clock source may be

further divided by the ADC12PDIV and the ADC12DIVx bit settings. The overall result is the ADC12CLK signal. The trigger source to initiate the analog-to-digital conversion is the SAMPCON signal. The specific trigger source is selected by the ADC12SHSx bits. The digital result of the conversion provided as n bits is stored in the ADC12MEM0 result registers. Specific interrupts may be selected to signal different significant events in the ADC process [SLAU367O, 2017].

MSP430FR5994 ADC Interrupts. Monitoring the ADC12_B system is a series of multiple interrupts to flag different events during ADC operation. These monitoring interrupts are normally off and must be individually enabled, as shown in Figure 9.11. Also, as with all interrupts, the GIE bit in the SR must also be enabled. If a specific interrupt is enabled for a given application, a corresponding interrupt service routine (ISR) must be provided.

The ADC interrupts include [SLAU367O, 2017]:

- ADC memory interrupt flags (ADC12IFG0 to ADC12IFG31)
- ADC12MEMx overflow (ADC12OVIFG)
- Conversion time overflow (ADC12TOVIFG)
- ADC12LO interrupt flag (ADC12LOIFG)
- ADC12IN interrupt flag (ADC12INIFG)
- ADC12HI interrupt flag (ADC12HIIFG)

As shown in Figure 9.11, a specific interrupt must be enabled by asserting its corresponding bit in the ADC interrupt enable registers (ADC12IE0 to ADC12IE2). When an enabled interrupt event occurs, the corresponding flag bit is set in the ADC12_B interrupt flag (ADC12IFG1 to ADC12IFG3) registers. The numerical value of the highest priority active interrupt may be read from the ADC12_B interrupt vector (ADC12IV) register.

All ADC interrupt sources are provided to the ADC interrupt vector (ADC12IV) register. When an ADC interrupt(s) occurs, the interrupt sources are sent to the ADC12IV register where they are prioritized.

Results register. The result(s) of sampled and converted signal(s) are stored in the ADC conversion memory (ADCMEMx) registers. Prior to storage in the result registers, the result is formatted to user specified configuration. Also, if the comparator features are asserted, appropriate interrupts are set.

9.4.3 MSP430FR5994 REGISTER SUMMARY

ADC operation and feature selection are determined by user selected settings in the ADC registers. The ADC register set includes the following registers [SLAU367O, 2017]:

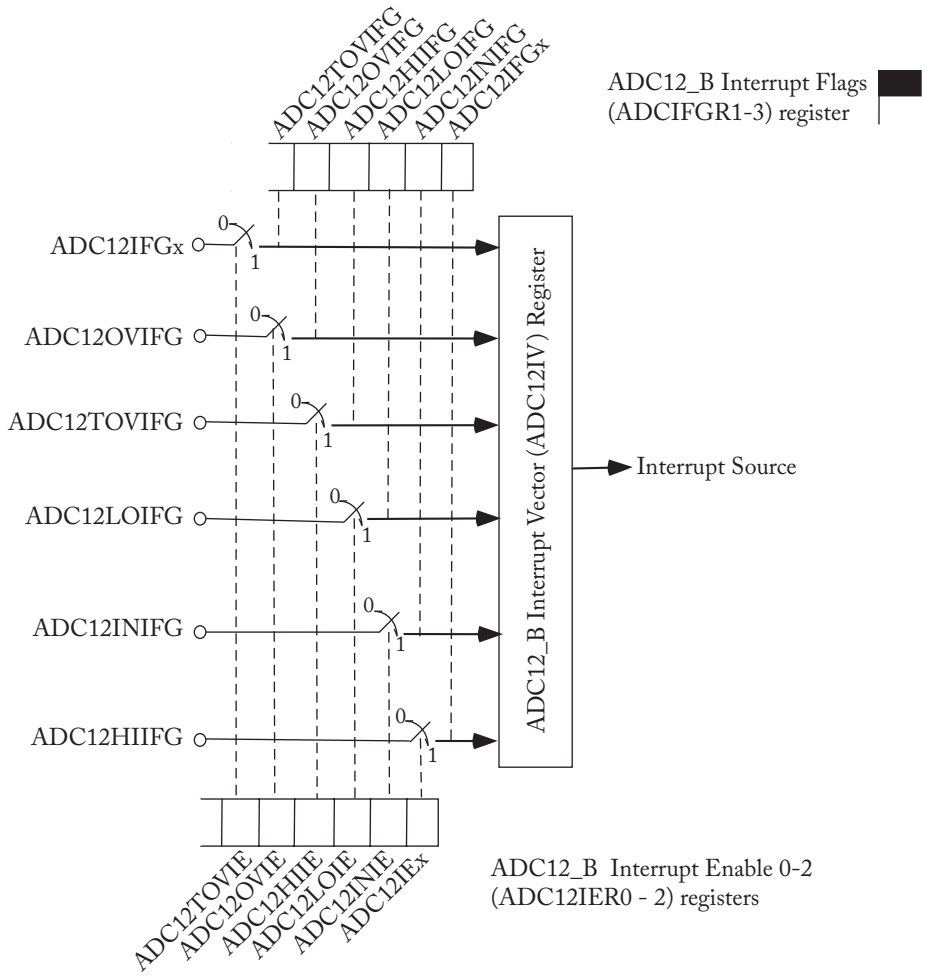


Figure 9.11: MSP430FR5994 ADC12_B interrupts overview.

- ADC12CTL0 to ADC12CTL3, ADC Control registers 0–3
- ADC12MCTL0 to ADC12MCTL31, ADC Memory Control registers 0–31
- ADC12MEM0 to ADC12MEM31, ADC Conversion Memory register. Conversion results are stored here.
- ADC12IER0 to ADC12IER2, ADC Interrupt Enable registers. Used to enable individual interrupts.
- ADC12IFGR0 to ADC12IFGR2, ADC Interrupt Flag registers. Corresponding flags for individual ADC interrupts are provided here.
- ADC12IV, ADC Interrupt Vector register. Provides numerical value of active interrupt.

Details of configuring selected registers is provided in SLAU367O [2017]. We highlight the settings of selected ADC Control Registers in Figure 9.12.

9.4.4 ANALYSIS OF RESULTS

The ADC12_B provides a digital representation of the analog sample in a binary unsigned format. The values range from 0000_h to $3FFF_h$. If the sampled signal is below the low reference voltage, ADC12_B reports 0000_h; whereas, if the sampled signal exceeds the high reference, ADC12_B reports $3FFF_h$. For analog sensed values between the low and high reference voltage, ADC12_B reports a value of N_{ADC} when configured for single-ended operation [SLAU367O, 2017]:

$$N_{ADC} = 2^{12} \times ((V_{in+} + 1/2LSB - V_{R+})/(V_{R+} - V_{R-})),$$

where:

$$LSB = (V_{R+} - V_{R-})/2^{12}.$$

If configured for differential mode, ADC12_B reports a value of:

$$N_{ADC} = 2^{11} \times ((V_{in+} - V_{in-})/(V_{R+} - V_{R-})) + 2^{11}.$$

9.4.5 PROGRAMMING THE MSP430FR5994 ADC12_B SYSTEM

The ADC12_B system may be programmed using Energia, APIs contained in DriverLib, and via register settings in C.

Programming the MSP430FR5994 with Energia

The Energia library contains several functions to support analog conversions including ADC- and DAC-related functions (`www.energia.nu`).

- **analogRead():** The analogRead function performs an ADC conversion on the indicated analog pin. The measured voltage is converted to an integer value between 0 and 1023, where 0 corresponds to 0 VDC and 1023 corresponds to 3.3 VDC.

15	14	13	12	11	10	9	8
ADC12SHT1x				ADC12SHT0x			
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)
7	6	5	4	3	2	1	0
ADC12MSC	Reserved		ADC12ON	Reserved		ADC12ENC	ADC12SC
rw-(0)	r-0	r-0	rw-(0)	r-0	r-0	rw-(0)	rw-(0)
	Can be modified only when ADC12ENC = 0						

(a) ADC12_B Control 0 register (ADC12CTL0)
ADC12CTL0[4]: ADC on: ADC12ON: 0 = ADC off, 1 = ADC on
ADC12CTL0[1]: ADC enable conversion: ADC12ENC: 0 = disabled, 1 = enabled
ADC12CTL0[0]: ADC start conversion: ADC12SC: 0 = no conversion, 1 = start conversion

15	14	13	12	11	10	9	8
Reserved	ADC12PDIV		ADC12SHSx			ADC12SHP	ADC12ISSH
r-0	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)
7	6	5	4	3	2	1	0
ADC12DIVx			ADC12SSELx		ADC12CONSEQx		ADC12BUSY
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	r-(0)
	Can be modified only when ADC12ENC = 0						

(b) ADC12_B Control 1 register (ADC12CTL1)
ADC12CTL1[14-13]: ADC12_B clock predivider: ADC12PDIV: 00 = 1, 01 = 4, 10 = 32, 11 =64
ADC12CTL1[7-5]: ADC clock divider: ADC12DIVx: 000 = 1 to 111 = 8
ADC12CTL1[4-3]: ADC clock source select: ADC12SSELx: MODCLK (00), ACLK (01), MCLK (10), SMCLK (11)
ADC12CTL1[2-1]: ADC conversion sequence: ADC12CONSEQx:
single channel, single conversion (00): sequence of channels, single conversion (01),
single channel, repeat conversion (10): sequence of channels, repeat conversion (11)
ADC12CTL1[0]: ADC12BUSY: 0 = inactive, 1 = active

15	14	13	12	11	10	9	8
Reserved							
r0	r0	r0	r0	r0	r0	r0	r0
7	6	5	4	3	2	1	0
Reserved		ADC12RES		ADC12DF	Reserved		ADC12PWRMD
r0	r0	rw-(1)	rw-(0)	rw-(0)	r0	r0	rw-(0)

(c) ADC12_B Control 2 register (ADC12CTL2)
ADC12CTL2[5-4]: ADC resolution: ADC12RES: 00 = 8 bit, 01 = 10 bit, 10 = 12 bit
ADC12CTL2[3]: ADC read back format: ADC12DF: 0: unsigned binary, 1= 2's complement signed binary

15	14	13	12	11	10	9	8
Reserved	ADC12WINC	ADC12DIF	Reserved	ADC12VRSEL			
r0	rw-(0)	rw-(0)	r0	rw-(0)	rw-(0)	rw-(0)	rw-(0)
7	6	5	4	3	2	1	0
ADC12EOS	Reserved		ADC12INCHx				
rw-(0)	r0	r0	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)
	Can be modified only when ADC12ENC = 0						

(d) ADC12_B Conversion Memory Control Register (ADCMCTLx)
ADCMCTLx [11-8]: ADC12VRSEL: Voltage reference select
ADCMCTLx[7]: ADC12EOS: 0 = not complete, 1 = complete
ADCMCTLx[4-0]: ADC12INCHx : Input channel select 00000 to 11111

Figure 9.12: MSP430FR5994 ADC control registers [SLAU367O, 2017]. (Illustration used with permission of Texas Instruments (www.ti.com).)

- **analogReference():** The analogReference function provides for changing the high-level reference voltage for the ADC. The different settings include the following.
  - DEFAULT: sets ADC high reference level to VCC 3.3 V.
  - INTERNAL1V5: sets ADC high reference level to internal 1.5 VDC reference.
  - INTERNAL2V5: sets ADC high reference level to internal 2.5 VDC reference.
  - EXTERNAL: sets ADC high reference level to the VREF pin value.
- **map:** As its name implies the map function maps a range of integers (fromLow, fromHigh) to a new range of integers (toLow, toHigh).
- **analogWrite:** The analogWrite function generates a pseudo analog output signal using a pulse width modulated signal. The analogWrite function generates a 490 Hz signal on the specified pin with a duty cycle specified from 0–255.

Programming the MSP430FR5994 ADC12_B in C

To successfully configure and program the converter, the following steps should be followed.

1. Select the desired ADC reference voltage via ADC12SREFx bits in the ADC12MCTLx register.
2. Connect analog signal(s) for conversion to appropriate input pins (A0-A31).
3. Turn on the ADC: ADC12ON in ADC12CTL0.
4. Select the clock source and the sampling mode in ADC12CTL1.
5. Configure the converter for proper operation: single channel, single conversion; single channel, multiple conversion; multiple channels, single conversion; or multiple channels, multiple conversions using ADC12CONSEQx bits in ADC12CTL1.
6. Initiate conversion: ADC12ENC bit in ADC12CTL0.
7. Monitor for conversion completion using the ADC12IFG0 flag.
8. Use the results of the conversion located in the corresponding result register (ADC12MEMx).
9. Repeat the process starting at step 6.

Example: In this example a single sample is made on analog input A1 with reference to AVcc. In the mainloop, the MSP430FR5994 waits in LPM0 to save power until the ADC12 conversion is complete. The ADC12 interrupt service routine will force exit from LPM0 in the mainloop

on the return from the interrupt. If the sampled value of A1 is greater than 0.5*AVcc, P1.0 is set, else it is reset. The full, correct handling of the ADC12 interrupt is shown.

Note how the #if, #elif, #else, and #endif directives are used to implement conditional compilation. This allows the same interrupt service routine to be used by the TI, IAR, or the GNU compiler.

```
//******************************************************************
// --COPYRIGHT--,BSD_EX
// Copyright (c) 2015, Texas Instruments Incorporated
// All rights reserved.
//
//                  MSP430 CODE EXAMPLE DISCLAIMER
//
//******************************************************************
//MSP430FR5x9x Demo - ADC12, Sample A1, AVcc Ref,
//Set P1.0 if A1 > 0.5*AVcc
//
//Description: A single sample is made on A1 with reference to AVcc.
//Software sets ADC12SC to start sample and conversion - ADC12SC
//automatically cleared at EOC. ADC12 internal oscillator times
//sample (16x) and conversion. In mainloop MSP430 waits in LPM0 to
//save power until ADC12 conversion complete, ADC12_ISR will force
//exit from LPM0 in mainloop on reti. If A1 > 0.5*AVcc, P1.0 set,
//else reset. The full, correct handling of and ADC12 interrupt is
//shown as well.
//
//               MSP430FR5994
//             -----------------
//         /|\|              XIN|-
//          | |                 |
//          --|RST          XOUT|-
//            |                 |
//        >---|P1.1/A1      P1.0|-->LED
//
//William Goh, Texas Instruments Inc., October 2015
//Built with IAR Embedded Workbench V6.30 & Code Composer
//Studio V6.1
//******************************************************************

#include <msp430.h>
```

```
int main(void)
{
WDTCTL = WDTPW | WDTHOLD;                 //Stop WDT

                                          //GPIO Setup
P1OUT &= ~BIT0;                           //Clear LED to start
P1DIR |= BIT0;                            //Set P1.0/LED to output
P1SEL1 |= BIT1;                           //Configure P1.1 for ADC
P1SEL0 |= BIT1;

//Disable the GPIO power-on default high-impedance mode to activate
//previously configured port settings
PM5CTL0 &= ~LOCKLPM5;

                                          //Configure ADC12
ADC12CTL0 = ADC12SHT0_2 | ADC12ON;        //Sample time,S&H=16, ADC12 on
ADC12CTL1 = ADC12SHP;                     //Use sampling timer
ADC12CTL2 |= ADC12RES_2;                  //12-bit conversion results
ADC12MCTL0 |= ADC12INCH_1;                //A1 ADC input select; Vref=AVCC
ADC12IER0 |= ADC12IE0;                    //Enable ADC conv complete
                                          //interrupt
while(1)
  {
  __delay_cycles(5000);
  ADC12CTL0 |= ADC12ENC | ADC12SC;        //Start sampling/conversion
  __bis_SR_register(LPM0_bits | GIE);     //LPM0, ADC12_ISR will force exit
  __no_operation();                       //For debugger
    }
}

//*****************************************************************
#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector = ADC12_B_VECTOR
__interrupt void ADC12_ISR(void)

#elif defined(__GNUC__)
void __attribute__ ((interrupt(ADC12_B_VECTOR))) ADC12_ISR (void)
```

```
#else
#error Compiler not supported!

#endif
{
switch(__even_in_range(ADC12IV, ADC12IV__ADC12RDYIFG))
  {
  case ADC12IV__NONE:        break;   //Vector  0:  No interrupt
  case ADC12IV__ADC12OVIFG:  break;   //Vector  2:  ADC12MEMx Overflow
  case ADC12IV__ADC12TOVIFG: break;   //Vector  4:  Conv time overflow
  case ADC12IV__ADC12HIIFG:  break;   //Vector  6:  ADC12BHI
  case ADC12IV__ADC12LOIFG:  break;   //Vector  8:  ADC12BLO
  case ADC12IV__ADC12INIFG:  break;   //Vector 10:  ADC12BIN
  case ADC12IV__ADC12IFG0:            //Vector 12:  ADC12MEM0 Interrupt
       if(ADC12MEM0 >= 0x7ff)         //ADC12MEM0 = A1 > 0.5AVcc?
         P1OUT |= BIT0;               // P1.0 = 1
       else
         P1OUT &= ~BIT0;              //P1.0 = 0
       // Exit from LPM0 and continue executing main
       __bic_SR_register_on_exit(LPM0_bits);
       break;

  case ADC12IV__ADC12IFG1:   break;   //Vector 14:  ADC12MEM1
  case ADC12IV__ADC12IFG2:   break;   //Vector 16:  ADC12MEM2
  case ADC12IV__ADC12IFG3:   break;   //Vector 18:  ADC12MEM3
  case ADC12IV__ADC12IFG4:   break;   //Vector 20:  ADC12MEM4
  case ADC12IV__ADC12IFG5:   break;   //Vector 22:  ADC12MEM5
  case ADC12IV__ADC12IFG6:   break;   //Vector 24:  ADC12MEM6
  case ADC12IV__ADC12IFG7:   break;   //Vector 26:  ADC12MEM7
  case ADC12IV__ADC12IFG8:   break;   //Vector 28:  ADC12MEM8
  case ADC12IV__ADC12IFG9:   break;   //Vector 30:  ADC12MEM9
  case ADC12IV__ADC12IFG10:  break;   //Vector 32:  ADC12MEM10
  case ADC12IV__ADC12IFG11:  break;   //Vector 34:  ADC12MEM11
  case ADC12IV__ADC12IFG12:  break;   //Vector 36:  ADC12MEM12
  case ADC12IV__ADC12IFG13:  break;   //Vector 38:  ADC12MEM13
  case ADC12IV__ADC12IFG14:  break;   //Vector 40:  ADC12MEM14
  case ADC12IV__ADC12IFG15:  break;   //Vector 42:  ADC12MEM15
  case ADC12IV__ADC12IFG16:  break;   //Vector 44:  ADC12MEM16
  case ADC12IV__ADC12IFG17:  break;   //Vector 46:  ADC12MEM17
```

```
  case ADC12IV__ADC12IFG18:  break;   //Vector 48:  ADC12MEM18
  case ADC12IV__ADC12IFG19:  break;   //Vector 50:  ADC12MEM19
  case ADC12IV__ADC12IFG20:  break;   //Vector 52:  ADC12MEM20
  case ADC12IV__ADC12IFG21:  break;   //Vector 54:  ADC12MEM21
  case ADC12IV__ADC12IFG22:  break;   //Vector 56:  ADC12MEM22
  case ADC12IV__ADC12IFG23:  break;   //Vector 58:  ADC12MEM23
  case ADC12IV__ADC12IFG24:  break;   //Vector 60:  ADC12MEM24
  case ADC12IV__ADC12IFG25:  break;   //Vector 62:  ADC12MEM25
  case ADC12IV__ADC12IFG26:  break;   //Vector 64:  ADC12MEM26
  case ADC12IV__ADC12IFG27:  break;   //Vector 66:  ADC12MEM27
  case ADC12IV__ADC12IFG28:  break;   //Vector 68:  ADC12MEM28
  case ADC12IV__ADC12IFG29:  break;   //Vector 70:  ADC12MEM29
  case ADC12IV__ADC12IFG30:  break;   //Vector 72:  ADC12MEM30
  case ADC12IV__ADC12IFG31:  break;   //Vector 74:  ADC12MEM31
  case ADC12IV__ADC12RDYIFG: break;   //Vector 76:  ADC12RDY
  default: break;
  }
}

//*****************************************************************
```

9.5 MSP430FR5994 COMPARATOR

The MSP430FR5994 is equipped with a comparator system called COMP_E. COMP_E provides two channels of analog comparators. As its name implies, a comparator compares an analog signal with a known reference. If the analog signal is greater than the reference, the output of the comparator is logic high. On the other hand, if the analog signal is less than the reference, the analog output is low.

The COMP_E system is controlled by a complement of registers. Register settings are provided in *MSP430FR58xx, MSP430FR59xx, and MSP430FR6xx Family User's Guide* [SLAU367O, 2017].

- CECTL0 Comparator_E control register 0
- CECTL1 Comparator_E control register 1
- CECTL2 Comparator_E control register 2
- CECTL3 Comparator_E control register 3
- CEINT Comparator_E interrupt register
- CEIV Comparator_E interrupt vector word

Example: In this example, the comparator CompE is used with an internal reference to determine if the input "Vcompare" is high or low. When "Vcompare" exceeds 2.0 V, COUT goes high and when "Vcompare" is less than 2.0 V, CEOUT goes low.

```
//*******************************************************************
// --COPYRIGHT--,BSD_EX
// Copyright (c) 2015, Texas Instruments Incorporated
// All rights reserved.
//
//                  MSP430 CODE EXAMPLE DISCLAIMER
//
//*******************************************************************
//MSP430FR5x9x Demo - COMPE output Toggle in LPM4; input channel C1;
//Vcompare is compared against internal 2.0V reference
//
//Description: Use CompE and internal reference to determine if
//input'Vcompare' is high or low.  When Vcompare exceeds 2.0V COUT
//goes high and when Vcompare is less than 2.0V then CEOUT goes low.
//
//               MSP430FR5994
//             ------------------
//         /|\|                  |
//          | |                  |
//          --|RST        P1.1/C1|<--Vcompare
//            |                  |
//            |         P1.2/COUT|----> 'high'(Vcompare>2.0V);
//            |                  |      'low'(Vcompare<2.0V)
//            |                  |
//
//William Goh, Texas Instruments Inc., October 2015
//Built with IAR Embedded Workbench V6.30 & Code Composer
//Studio V6.1
//*******************************************************************

#include <msp430.h>

int main(void)
{
WDTCTL = WDTPW | WDTHOLD;               //Stop WDT
```

```
//Configure P1.1 as C1 and P1.2 as COUT
P1SEL0 |= BIT1;
P1SEL0 &= ~(BIT2);
P1SEL1 |= BIT1 | BIT2;
P1DIR |= BIT2;

//Disable the GPIO power-on default high-impedance mode to activate
//previously configured port settings
PM5CTL0 &= ~LOCKLPM5;

//Setup Comparator_E
CECTL0 = CEIPEN | CEIPSEL_1;            //Enable V+, input ch CE1
CECTL1 = CEPWRMD_1;                     //normal power mode
CECTL2 = CEREFL_2 | CERS_3 | CERSEL;    //VREF is applied to -term
                                        //R-ladder off; bandgap ref
                                        //voltage supplied to ref
                                        //amplifier to get Vcref=2.0V
CECTL3 = BIT1;                          //Input Buffer Disable @P1.1/CE1
CECTL1 |= CEON;                         //Turn On Comparator_E
__delay_cycles(75);                     //delay for the ref to settle
__bis_SR_register(LPM4_bits);           //Enter LPM4
__no_operation();                       //For debug
}

//*******************************************************************
```

9.6 ADVANCED ANALOG PERIPHERALS

Although not available on the MSP430FR2433 and the MSP430FR5994 variants, this section provides information on enhanced analog peripheral features.

9.6.1 SMART ANALOG COMBO (SAC)

Available on MSP430FR2355 and MSP430FR2311 microcontrollers, the Smart Analog Combo is an analog peripheral module that removes needs for external Op Amps circuits for interfacing sensors and measurement devices with the MSP430 controller. The module connects the controller to a sensor whose output is a low amplitude analog signal that needs to be amplified and digitized before it can be used by a microcontroller. It can also be configured to generate an analog signal for driving an external device. Thus, it is a programmable submodule of the controller that performs signal conditioning for analog input and output signals, removing the needs for additional circuits and providing one-chip solution. The module is made of gain

amplifiers (gain value of up to 33) integrated with a 12-bit digital-to-analog converter (DAC). For inputs, the module is connected internally to an internal analog-to-digital converter (ADC) for MSP430FR2355, for example, again, allowing built-in, easy analog to digital conversion and signal conditioning capabilities

Three configurations are available for the Smart Analog Combo module. Configuration SAC-L1 allows only the integration of an operational amplifier function for an application. Configuration SAC-L2 allows application of a programmable gain amplifier (PGA) with gain up to 33, and configuration SAC-L3 allows the integration of PGA and DAC. MSP430 family microcontrollers have varying number of SAC units in a SAC module, enabling different configurations for a variety of applications. We refer the reader to obtain device specific data sheet for details.

9.6.2 ENHANCED COMPARATOR (ECOMP)

The enhanced comparator of MSP430 controller compares an analog voltage signal with the output of a built-in 6 bit DAC output, working as a reference signal. The comparator is available to some of MSP430 family controllers (MSP430FR2111, MSP430FR2110, MSP430FR2100, and MSP430FR2000) to allow a programmer to configure the controller for simple analog-to-digital applications, applications not requiring high-resolution ADC capabilities. The enhanced comparator is typically used to implement a simple ADC function.

9.6.3 TRANSIMPEDANCE AMPLIFIER (TIA)

Found in some of MSP430 controllers (MSP430F2274, MSP430FR2311, for example), the transimpedance amplifier converts an electric current signal into a voltage signal. Using the built-in analog-to-digital converter, TIA is used in an application where the input signal is in the form of electric current. The unit contains an integrated operational amplifier, which needs to be programmed according to requirements of an application along with a feedback resistor. A typical application of TIA is to measure the amount of lights in an environment using a photodiode connected to an MSP430 controller. The conversion allows a controller to initiate actions appropriately, based on the light condition of an environment. The objective of designers of the TIA module is to provide users with a tool to simplify an overall system design by taking advantage of the onboard operational amplifier to create an ADC for electrical current signal input.

9.7 LABORATORY EXERCISE: SMART HOME SENSOR

In this section, your task is to program both the ADC12 and DAC12 converters to perform two smart home functions. How many times have you been told by your parents to turn off lights as you were growing up? Or if you are a parent, how many times did you have to remind your children to turn off lights when they leave their rooms? In the smart home of our choice, we

will solve this problem once and for all. We will do so using the MSP430 controller's analog-to-digital converters.

To that end, suppose that we installed infrared sensors at each room of the house to detect human locations. A precise location of each human is measured using multiple infrared sensors, and based on the locations of humans, in the house, lights will automatically be turned on and turned off. An infrared sensor consists of a transmitter and a receiver. The transmitter sends out an infrared signal and waits for any reflection of the transmitted signal. If a human is in the path of the signal, the signal is reflected back to the receiver. By measuring the amount of reflection of the light, we can compute the distance of a human from the particular sensor. By placing these sensors at appropriate locations in each room and combining the sensor information, we can compute the location of each person in the house. The job of the ADC is to convert the received infrared light signal into appropriate distance from the sensor. That is, the received signal strength is converted into a numerical number that represents the distance.

Suppose that the infrared light transmitter and receiver pair is calibrated such that when a person is within 2 ft from the sensor mounted on a wall the receiver generates 2.5 VDC. When a person is at distance 15 ft away from the sensor, the receiver generates a 1.0 VDC signal. The output of the receiver changes linearly as the distance between the sensor and a person changes from 2–15 ft. If no person is in front of the sensor, the receiver outputs 0 V. Suppose also that for each sensor, a MSP430 microcontroller is attached. Your task is to write a program utilizing the ADC module to do the following.

1. Use the 12-bit format of ADC.
2. Set reference voltages to be 2.5 V and 0 V.
3. Insure the receiver output is directly connected to pin A0. Continuously convert ADC12 inputs.
4. Using a four-sample sliding window and the associated interrupt system, continually compute the average of four ADC outputs and send resulting average values to an array.

MSP430 controllers (sensors) around the smart home continuously send their data to a central controller that controls which lights of the house get to be turned on and turned off. Of course, some type of a high-level decision maker controller must also be involved to remove any irritable system behavior such as turning lights off in an area when a person leaves the area for a short time.

9.8 REFERENCES AND FURTHER READING

Analog Input to PWM Output Using the MSP430 MCU Enhanced Comparator, (SLAA833), Texas Instruments, 2018.

MSP430F2273 Transimpedance Amplifier, (TIDU443),

MSP430FR4xx and MSP430FR2xx Family User's Guide, (SLAU445G), Texas Instruments, 2016. 366, 369, 371, 372

MSP430FR58xx, MSP430FR59xx, and MSP430FR6xx Family User's Guide, (SLAU367O), Texas Instruments, 2017. Texas Instruments, 2014. 376, 379, 381, 382, 387

9.9 CHAPTER PROBLEMS

Fundamental

1. Using the Nyquist sampling rate, find the minimum sampling frequency of an ADC if the highest frequency of an input analog signal is 2 kHz.
2. Given a sinusoidal input analog signal, $5\cos(2\pi 10kt)$, and sampling frequency of 1 KHz, find the first three sampled values with starting time 0.
3. Given an 8-bit ADC and input range of 0 V and 5 V, what is the quantization level for sampled value of 2.9 V?
4. What is the quantization error for the sampled signal in Problem 3?
5. What is the encoded value of quantization level from Problem 3?
6. Repeat questions 3–6 for a 3.3 VDC ADC.

Advanced

1. Write a program segment using the ADC to (1) operate with the 12 bit resolution, (2) use internal reference voltages of 2.5 V and 0 V, (3) continuously sample analog signals from pins A0 and A1, (4) use the unsigned binary format, (5) compare the input analog values, (6) turn the logic state on Pz.x pin high if the signal on A0 is higher than the one on A1, otherwise, turn the logic state low, and (7) turn the logic state on Pz.y pin high if the signal on A1 is higher than the one on A1, otherwise, turn the logic state low.
2. A weather station is equipped with a vane to indicate wind direction. The output voltage of the weather vane is linearly related to wind direction. The vane provides 0 V for wind heading North (0 degrees) and 3.3 V for 360°. Write a program to convert the output of the weather vane to wind direction in degrees. Display the result on a serial configured liquid crystal display.
3. The LM34 is a precision Fahrenheit temperature sensor. The LM34 provides a linear output of 10 mV per degree Fahrenheit. Sketch the required circuit for the LM34. Write a program to convert the output of the LM34 to temperature. Display the result on a serial configured liquid crystal display.

Challenging

1. Present your design and write a program to construct a smart home that locate your position in room whose size is 10 ft wide, 10 ft long, and 9 ft high. Assume that you need to use infrared sensors to do the job. You can use as many sensors as you need but want to minimize the number used. Suppose the infrared sensor output is fed to an ADC of a MSP430 and you have means to communicate among MSP430s. Design the sensor positions and write a program to locate a person in the room.

2. Extend the weather vane example above to include eight LEDs to indicate the closest wind direction (e.g., N, NE, E, etc.).

CHAPTER 10

Communication Systems

Objectives: After reading this chapter, the reader should be able to:

- describe the differences between serial and parallel communication methods;
- present the features of the MSP430 microcontroller's eUSCI systems A and B;
- illustrate the operation of the UART mode of the eUSCI;
- program the UART for basic transmission and reception;
- describe the operation of the SPI mode of the eUSCI;
- configure a SPI-based system to extend the features of the MSP430 microcontroller;
- describe the purpose and function of the inter-integrated communication (I^2C) mode of the eUSCI; and
- program the I^2C communication system for read and write to compatible devices.

Microcontrollers must often exchange data with other microcontrollers or peripheral devices. For such applications, data may be exchanged by using parallel or serial techniques. With parallel techniques, an entire byte (or a set of n bits) of data is typically sent simultaneously from a transmitting device to a receiving device or received at the same time from an external device. While this is efficient from a time point of view, it requires eight separate lines (or n separate lines) for the data transfer. Parallel connections are typically limited to short lengths.

In serial transmission, data is sent or received a single bit at a time. For a byte-size data transmission, once eight bits have been received at the receiver, the data byte is reconstructed. While this is inefficient from a time point of view, it only requires a line (or two) to transmit and receive the data. Serial transmission techniques also help minimize the use of precious microcontroller I/O pins.

10.1 BACKGROUND

The MSP430 microcontroller is equipped with the eUSCI, with subsystems shown in Figure 10.1. The eUSCI consists of two different communication subsystems: eUSCI_A type modules and eUSCI_B modules. Each microcontroller in the MSP430 line has a complement of A and B type eUSCI modules. Should a specific MSP430 microcontroller type have more than

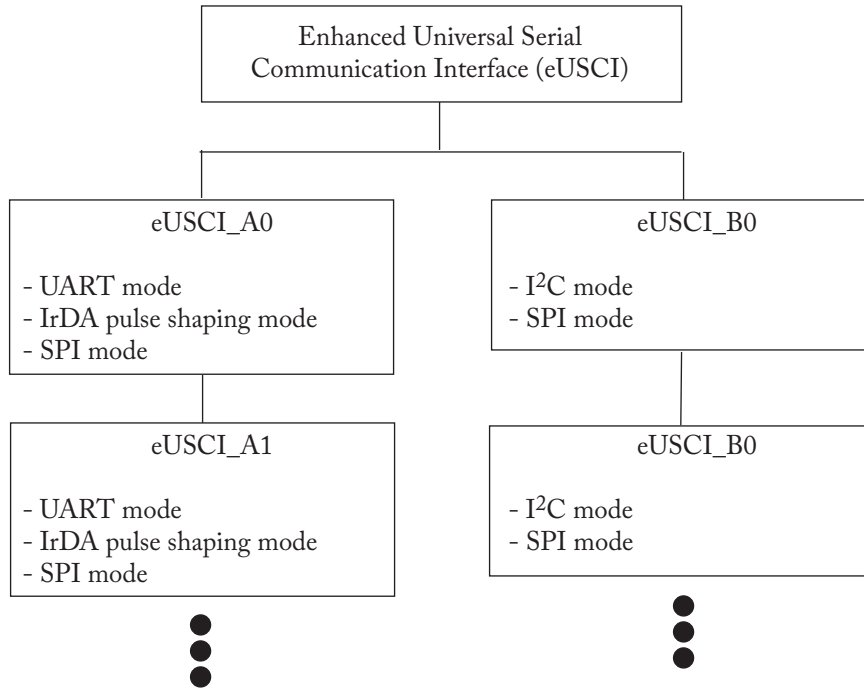


Figure 10.1: MSP430 enhanced universal serial communication interface (eUSCI).

one of the A and/or B type modules, they are numbered sequentially starting with zero (e.g., eUSCI_A0, A1, etc.) [SLAU445G, 2016, SLAU367O, 2017]. The MSP430FR2433 is equipped with two eUSCI_A channels (A0, A1) and one eUSCI_B channel (B0). The MSP430FR5994 is equipped with four eUSCI_A channels (A0 to A3) and four eUSCI_B channels (B0 to B3).

As can be seen in Figure 10.1, eUSCI_A modules provide support for [SLAU445G, 2016, SLAU367O, 2017] the following.

- Universal asynchronous serial receiver and transmitter (UART). The UART provides a serial data link between a transmitter and a receiver. The transmitter and receiver pair maintains synchronization using start and stop bits that are embedded in the data stream.
- Infrared data association (IrDA). The IrDA protocol provides for a short-range data link using an infrared (IR) link. It is a standardized optical protocol for IR linked devices. It is used in various communication devices, personal area networks, and instrumentation.
- The serial peripheral interface (SPI). The SPI provides synchronous communications between a receiver and a transmitter. The SPI system maintains synchronization between the transmitter and receiver pair using a common clock provided by the master designated microcontroller. An SPI serial link has a much faster data rate than UART.

- The I^2C bus is a two-wire bus with a serial data line (SDL) and the serial clock line (SCL). I^2C compatible devices, each with a unique address, are connected to the two-wire bus as either a master device or a slave device. The MSP430 eUSCI device allows its I^2C communication unit to operate either in the standard mode (100 kbps) or in the fast mode (400 kbps) with either a 7- or 10-bit device addressing [SLAU445G, 2016, SLAU367O, 2017].

The eUSCI_B modules also provide support for SPI communications and inter-integrated communication (I^2C) communications. The I^2C bus is a two-wire bus with a serial data line (SDL) and the serial clock line (SCL). By configuring devices connected to the common I^2C bus as either a master device or a slave device, multiple devices can share information. The I^2C system is used to link multiple peripheral devices to a microcontroller or several microcontrollers together in a system that are in close proximity to one another [SLAU445G, 2016, SLAU367O, 2017].

Space does not permit an in-depth discussion of all communication features of the eUSCI system. We concentrate on the basic operation of the UART, SPI and I^2C systems. For each system, we provide a technical overview, a review of system registers, and code examples. We begin with a review of serial communication concepts.

10.2 SERIAL COMMUNICATION CONCEPTS

Before we delve into the serial communication technologies, we first review common serial communication terminology.

Asynchronous vs. synchronous serial transmission: In serial communications, the transmitting and receiving devices must agree on the "rules of engagement" by using a common data rate and protocol. This allows both the transmitter and receiver to properly coordinate data transmission/reception. There are two basic methods of maintaining coordination or "sync" between the transmitter and receiver: asynchronous and synchronous.

In an asynchronous serial communication system, such as the UART aboard the MSP430 microcontroller, framing bits are used at the beginning and end of a data byte. The framing bits alert the receiver that an incoming data byte has arrived and also signal the completion of the data byte reception. The data rate for an asynchronous serial system is typically much slower than the synchronous system, but it only requires a single wire between the transmitter and receiver (and a common ground) for simplex (one way) communication.

A synchronous serial communication system maintains "sync" between the transmitter and receiver by employing a common clock between the two devices. Data bits are sent and received at the time when a clock edge appears. This allows higher data transfer rates than with asynchronous techniques but the communication method requires a minimum of two lines (and a common ground), data and clock, to connect a receiver and a transmitter for simplex communications.

Baud rate: Data transmission rates are typically specified as a Baud or bits per second rate. For example, 9600 Baud indicates the data is being transferred at 9600 bits per second.

Full duplex: Often serial communication systems must both transmit and receive data simultaneously. To do so requires separate hardware for transmission and reception at each end of the communication link. A single duplex system has a single complement of hardware that must be switched from transmission to reception configuration. A full duplex serial communication system has separate hardware for transmission and reception.

Non-return to zero (NRZ) coding format: There are many different coding standards used within serial communications. The important point is a transmitter and a receiver must use a common coding standard so data may be interpreted correctly at the receiving end. The MSP430 microcontroller uses a non-return to zero (NRZ) coding standard. In NRZ coding, a logic one is signaled by a logic high during the entire time slot allocated for a single bit, whereas, a logic zero is signaled by a logic low during the entire time slot allocated for a single bit.

The RS-232 communication protocol: When serial transmission occurs over a long distance, additional techniques may be used to insure data integrity. Over long distances, logic levels degrade and may be corrupted by noise. When this happens at the receiving end, it is difficult to discern a logic high from a logic low. The RS-232 standard has been around for some time. With the RS-232 standard (EIA RS-232), a logic one is represented with a −12 VDC level while a logic zero is represented by a +12 VDC level. Chips are commonly available (e.g., MAX3232) that convert the output levels from a microcontroller to RS-232 compatible levels and convert back to microcontroller compatible levels at the receiver. The RS-232 standard also specifies other features for this communication protocol such as connector type and pinout.

Parity: To enhance data integrity during transmission, parity techniques may be used. A parity bit is an additional bit (or bits) that is transmitted with the data byte. With a single parity bit, a single-bit error may be detected. Parity may use an even or odd parity bit. In even parity, the parity bit is set to one or zero such that the number of ones in the data byte including the parity bit is an even number. In odd parity, the parity bit is set to one or zero such that the number of ones in the data byte including the parity bit is odd. At the receiver, bits within a data byte, including the parity bit, are counted to determine if parity has changed during transmission. A change in parity indicates an error occurred during transmission. For single-bit error correction or multiple-bit error detection, additional parity bits are required.

ASCII: The American Standard Code for Information Interchange (ASCII) is a standardized, seven bit method of encoding alphanumeric data. It has been in use for many decades, so some of the characters and actions listed in the ASCII table are not in common use today. However, ASCII is still the most common method of encoding alphanumeric data code, is shown in Figure 10.2. For example, the capital letter "G" is encoded in ASCII as 0x47. The "0x" symbol indicates the hexadecimal number representation. Unicode is the international counter-

part of ASCII. It provides standardized 16-bit encoding format for the written languages of the world. ASCII is a subset of Unicode. The interested reader is referred to the Unicode home page website at: `www.unicode.org` for additional information on this standardized encoding format.

Least Significant Digit	Most Significant Digit 0x0_	0x1_	0x2_	0x3_	0x4_	0x5_	0x6_	0x7_
0x_0	NUL	DLE	SP	0	@	P	`	p
0x_1	SOH	DC1	!	1	A	Q	a	q
0x_2	STX	DC2	"	2	B	R	b	r
0x_3	ETX	DC3	#	3	C	S	c	s
0x_4	EOT	DC4	$	4	D	T	d	t
0x_5	ENQ	NAK	%	5	E	U	e	u
0x_6	ACK	SYN	&	6	F	V	f	v
0x_7	BEL	ETB	'	7	G	W	g	w
0x_8	BS	CAN	(	8	H	X	h	x
0x_9	HT	EM	)	9	I	Y	i	y
0x_A	LF	SUB	*	:	J	Z	j	z
0x_B	VT	ESC	+	;	K	[	k	{
0x_C	FF	FS	‘	<	L	\	l	|
0x_D	CR	GS	-	=	M	]	m	}
0x_E	SO	RS	.	>	N	^	n	~
0x_F	SI	US	/	?	O	_	o	DEL

Figure 10.2: ASCII code. The ASCII code is used to encode alphanumeric characters. The "0x" indicates the hexadecimal notation in the C programming language.

10.3 MSP430 UART

The UART system is located within the eUSCI module A. In this section, we discuss UART features, provide an overview of the UART hardware operation and character format, discuss how to set the UART Baud rate, provide an overview of UART-related registers, and conclude with several examples.

10.3.1 UART FEATURES

The MSP430 microcontroller is equipped with a powerful and flexible UART system. To select the UART (asynchronous) mode the Synchronous Mode Enable bit (UCSYNC bit) located in the eUCSI_Ax Control Register 0 (part of eUSCI_Ax Control Word 0) must be cleared to 0. This action places the system in the asynchronous mode. When in this mode, serial data is transmitted from the microcontroller via the UCAxTXD pin and received via the UCAxRXD

pin. (Note: The "x" designates which eUSCI_A module is employed (e.g., 0, 1, 2)) [SLAU445G, 2016, SLAU367O, 2017].

The UART system provides features that allow the MSP430 to communicate with a wide variety of peripheral devices or another microcontroller. These features include [SLAU445G, 2016, SLAU367O, 2017]:

- support for serial transmission protocols including the capability to transmit 7- or 8-bit data with odd, even, or no parity;
- independent transmit and receive shift registers equipped with separate transmit and receive buffer registers;
- capability to send or receive data the LSB first or the MSB on both the transmit and receive channels. This feature allows the MSP430 microcontroller to match the protocol of an existing peripheral device;
- capability to operate within a multiprocessor system using the built-in, idle-line, and address-bit communication protocols;
- auto wake-up feature from a low power mode (LPMx) when a start edge is received;
- extensive flexibility in setting programmable baud rates;
- system status flags for error detection, error suppression, and address detection; and
- interrupts for the data receive and transmit.

In the next section, we examine how these features are incorporated into the UART hardware.

10.3.2 UART OVERVIEW

Provided in Figure 10.3 is a block diagram of the eUCSI_Ax module configured for UART (asynchronous) mode (UCSYNC bit = 0). The UART module can be subdivided into the Baud--Rate Generator (center of Figure 10.3), the receiver-related hardware (top of figure), and the transmit hardware (lower portion of figure). We discuss each in turn.

The eUSCI_Ax module communicates asynchronously with another device (e.g., peripheral) when the UCSYNC mode is set to zero. As previously mentioned, in an asynchronous mode, the transmitter and receiver maintain synchronization with one another, using start and stop bits to frame each data byte sent. It is essential that both transmitter and receiver are configured with the same Baud rate, number of start and stop bits, and the type of parity employed (odd, even, or none).

The Baud rate is set using the Baud-Rate Generator shown in the center of Figure 10.3. The clock source for the Baud-Rate Generator may either be the UCAxCLK, ACLK, or SMCLK. The clock source is selected using the eUSCI clock source select bits (UCSSELx) located

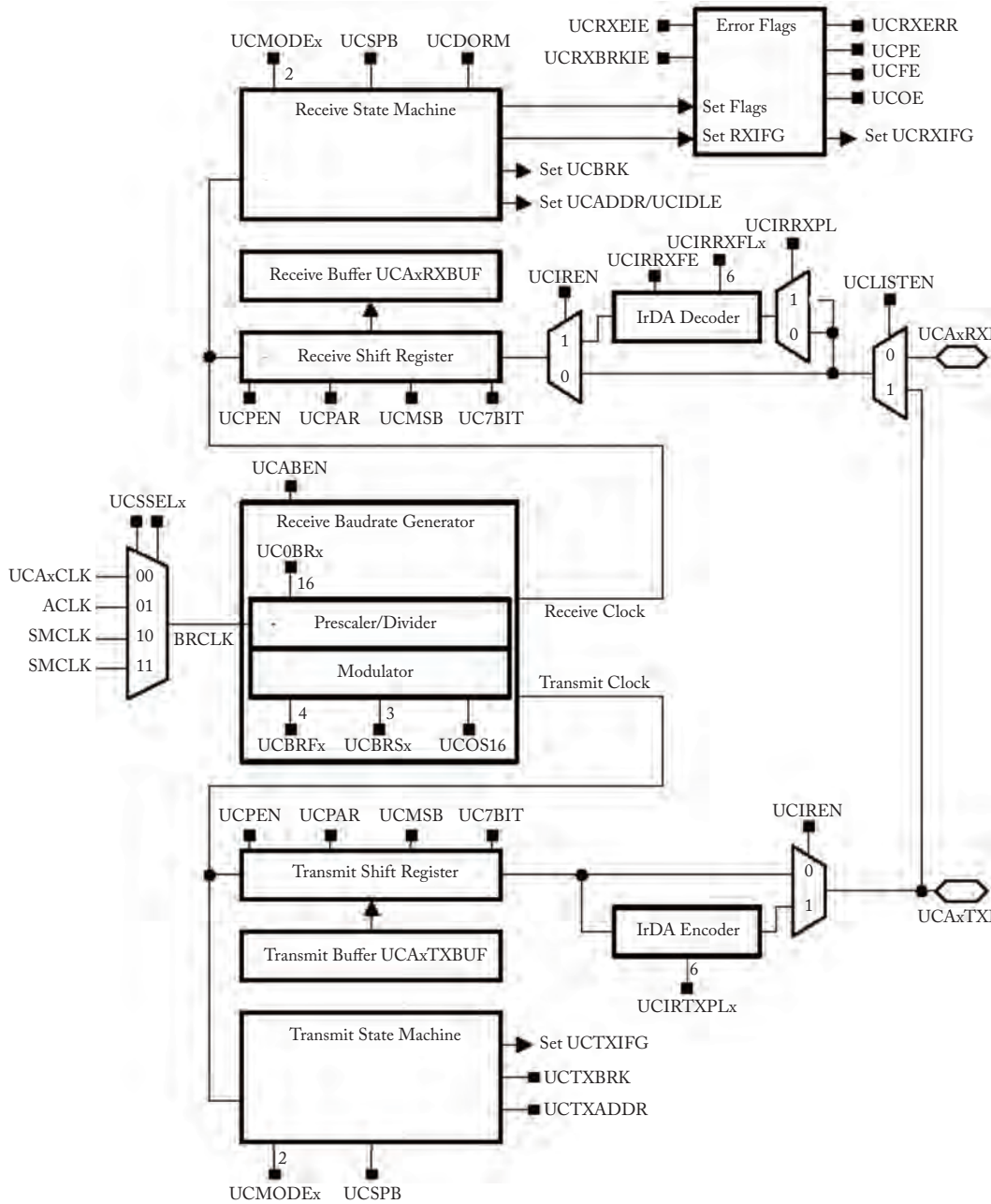


Figure 10.3: Block diagram of the eUCSI_Ax module configured for UART mode (UCSYNC bit = 0) [SLAU445G, 2016, SLAU367O, 2017]. (Illustration used with permission of Texas Instruments (www.ti.com).)

in the eUSCI_Ax Control Register 1 (UCAxCTL1). The source selected becomes the Baud-Rate clock (BRCLK). The Baud-Rate clock may then be prescaled and divided to set the Baud rate for the transmit and receive clock.

The receive portion of the UART system is in the upper part of Figure 10.3. Serial data is received via the UCAxRXD pin. The serial data is routed into the Receive Shift Register when the UCLISTEN bit located within the eUSCI_Ax Status Register (UCAxSTAT) is set to zero. If required by the specific application, the data may first be routed through the IrDA Decoder.

The configuration of the Receive Shift Register is set by several bits located within the eUSCU_Ax Control Register 0 (UCAxCTL0). These include the:

- parity enable bit, UCPEN (0: parity disabled, 1: parity enabled),
- parity select bit, UCPAR (0: odd parity, 1: even parity),
- MSB first select, UCMSB (0: LSB first, 1: MSB first), and
- character length bit, UC7BIT (0: 8-bit data, 1: 7-bit data)

The Receive State Machine controls the operation of the receive associated hardware. It has control bits to:

- select the number of stop bits, UCSPB (0: one stop bit, 1: two stop bits),
- select the eUSCI mode, UCMODEx (00: UART mode), and
- select the synchronous mode, UCSYNC (0: asynchronous mode, 1: synchronous mode).

The hardware associated with serial data transmission is very similar to the receive hardware with the direction of data routed for transmission out of the UCAxTXD pin.

10.3.3 CHARACTER FORMAT

As previously mentioned, the UART system has great flexibility in setting the protocol of the serial data, including the number of bits (7 or 8), parity (even, odd, or none), MSB or LSB first, and selection of transmit/receive operation. A typical serial data word is illustrated in Figure 10.4. To verify the valid functionality of the communication using the MSP430 microcontroller, it is very helpful to write a short program to transmit the same piece of data continuously from the UART and observe the transmission on the UCAxTXD pin with an oscilloscope or a logic analyzer.

10.3.4 BAUD RATE SELECTION

The MSP microcontroller also has considerable flexibility in setting the Baud rate for UART transmission and reception. It has two different modes for Baud rate generation.

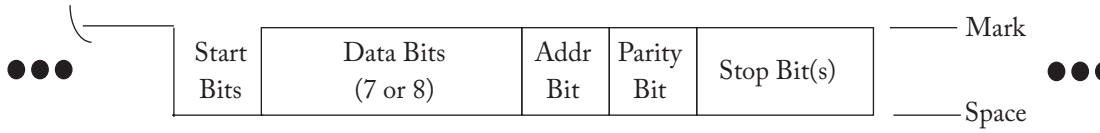


Figure 10.4: UART serial data format [SLAU445G, 2016, SLAU367O, 2017].

- Low-frequency Baud rate generation (UCOS16, Oversampling Mode Enable bit = 0). The mode allows Baud rates to be set when the microcontroller is being clocked by a low-frequency clock. A common choice is a 32,768 Hz crystal source. It is advantageous to do this to reduce power consumption by using a lower frequency time base. In this mode, the Baud-rate generator uses a prescaler and a modulator to generate the desired Baud rate. The maximum selectable Baud rate in this mode is limited to one-third of the Baud rate clock (BRCLK).

- Oversampling Baud rate generation (UCOS16 = 1). This mode employs a prescaler and a modulator to generate higher sampling frequencies.

To set a specific Baud rate, the following parameters must be determined:

- The clock prescaler setting (UCBRx) in the Baud Rate Control Register 0 and 1 (UCAxBR0 and UCAxBR1) must be determined. The 16-bit value of the UCBRx prescaler value is determined by UCAxBR0 + UCAxBR1 × 256.

- First modulation stage setting, UCBRFx bits in the eUSCI_Ax Modulation Control Register (UCAxMCTL).

- Second modulation stage setting, UCBRSx bits in the eUSCI_Ax Modulation Control Register (UCAxMCTL).

The documentation for the MSP430 microcontroller contains extensive tables for determining the UCBRx, UCBRFx, and UCBRSx bit settings for various combinations of the Baud rate clock (BRCLK) and desired Baud rate [SLAU445G, 2016, SLAU367O, 2017].

10.3.5 UART ASSOCIATED INTERRUPTS

The UART system has two associated interrupts. The transmit interrupt flag (UCTXIFG) is set when the UCAxTXBUF is empty, indicating another data byte may be sent. The receive interrupt flag (UCRXIFG) is set when the receive buffer (UCAxRXBUF) has received a complete character. Both of these interrupt flags are contained within the eUSCI_Ax interrupt flag register (UCAxIFG).

10.3.6 UART REGISTERS

As discussed throughout this section, the basic features of the UART system is configured and controlled by the following UART-related registers [SLAU445G, 2016, SLAU367O, 2017]:

- UCAxCTLW0 eUSCI_Ax Control Word 0
- UCAxCTL0 eUSCI_Ax Control 0
- UCAxCTL1 eUSCI_Ax Control 1
- UCAxCTLW1 eUSCI_Ax Control Word 1
- UCAxBRW eUSCI_Ax Baud Rate Control Word
- UCAxBR0 eUSCI_Ax Baud Rate Control 0
- UCAxBR1 eUSCI_Ax Baud Rate Control 1
- UCAxMCTLW eUSCI_Ax Modulation Control Word
- UCAxSTATW eUSCI_Ax Status
- UCAxRXBUF eUSCI_Ax Receive Buffer
- UCAxTXBUF eUSCI_Ax Transmit Buffer
- UCAxABCTL eUSCI_Ax Auto Baud Rate Control
- UCAxIRCTL eUSCI_Ax IrDA Control
- UCAxIRTCTL eUSCI_Ax IrDA Transmit Control
- UCAxIRRCTL eUSCI_Ax IrDA Receive Control
- UCAxIE eUSCI_Ax Interrupt Enable
- UCAxIFG eUSCI_Ax Interrupt Flag
- UCAxIV eUSCI_Ax Interrupt Vector

Details of specific register and bits settings are contained in SLAU367O [2017] and SLAU445G [2016]. Figure 10.5 provides the bit settings of eUSCI_Ax Control Word 0 for basic UART operation using eUSCI_Ax.

10.4 CODE EXAMPLES

The MSP430 UART features may be programmed using Energia, DriverLib APIs, or in C.

15	14	13	12	11	10	9	8
UCPEN	UCPAR	UCMSB	UC7BIT	UCSPB	UCMODEx		UCSYNC
rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0
7	6	5	4	3	2	1	0
UCSSELx		UCRXEIE	UCBRKIE	UCDORM	UCTXADDR	UCTXBRK	UCSWRST
rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-1

Can be modified only when UCSWRST = 1

UCAxCTLW0 Register
- UCAxCTLW0[15]: Parity enable: UCPEN: 0 = off, 1 = on
- UCAxCTLW0[14]: Parity select: UCPAR: 0 = odd, 1 = even
- UCAxCTLW0[13]: MSB first select: UCMSB: 0 = LSB first, 1 = MSB first
- UCAxCTLW0[12]: Character length: UC7BIT: 0 = 8-bit, 1 = 7-bit
- UCAxCTLW0[11]: Stop bit select: UCSPB: 0 = 1 stop bit, 1 = two stop bits
- UCAxCTLW0[10-9]: eUSCI_A mode: UCMODEx: 00 = UART mode
- UCAxCTLW0[8]: Synchronous mode enable: UCSYNC: 0 = Asynchronous, 1 = Synchronous
- UCAxCTLW0[7-6]: eUSCI_A clock source select: UCSSELx: 00 = UCLK, 01 = ACLK, 10 or 11 = SMCLK

Figure 10.5: eUSCI_Ax control word 0 [SLAU445G, 2016, SLAU367O, 2017]. (Illustration used with permission of Texas Instruments (www.ti.com).)

10.4.1 ENERGIA

The Energia Integrated Development Environment has many built-in functions to support UART operations (energia.nu). In the next several examples, we use:

- Serial.begin(baud_rate): sets the Baud rate for data transmission,
- Serial.print: prints data to the serial port as ASCII text, and
- Serial.println: prints data to the serial port as ASCII text followed by a carriage return.

Example: LCD. In this example a Sparkfun LCD-09067, 3.3 VDC, serial, 16 by 2 character, black on white LCD display is connected to the MSP430. Communication between the MSP430 and the LCD is accomplished by a single 9600 bits per second (BAUD) connection using the onboard universal asynchronous receiver transmitter (UART). The UART is configured for 8 bits, no parity, and one stop bit (8-N-1). The MSP-EXP430FR2433 LaunchPad is equipped with two UART channels. One is the back channel UART connection to the PC. The other is accessible by pin 3 (RX, P1.5) and pin 4 (TX, P1.4). Provided below is the sample Energia code to print a test message to the LCD. Note the UART LCD channel is designated "Serial1" in the program. The back channel UART for the Energia serial monitor display is designated "Serial."

```
//*****************************************************************
//Serial_LCD_energia
//Serial 1 accessible at:
```

```
// - RX: P1.5, pin 3
// - TX: P1.4, pin 4
//*****************************************************************

void setup()
  {
  //Initialize serial channel 1 to 9600 BAUD and wait for port to open
  Serial1.begin(9600);
  }

void loop()
  {
  Serial1.print("Hello World");
  delay(500);
  Serial1.println("...Hello World");
  delay(500);
  }

//*****************************************************************
```

Example: Voice chip. For speech synthesis, we use the SP0-512 text to speech chip (www.speechchips.com). The SP0-512 accepts UART compatible serial text stream. The text stream should be terminated with the carriage return control sequence (backslash r). The text stream is converted to phoneme codes used to generate an audio output. The chip requires a 9600 Baud bit stream with no parity, 8 data bits and a stop bit. The associated circuit is provided in Figure 10.6. Additional information on the chip and its features are available at www.speechchips.com.

```
//*****************************************************************
//SP0512
//Serial 1 accessible at:
// - RX: P1.5, pin 3
// - TX: P1.4, pin 4
//*****************************************************************

void setup()
  {
  //Initialize serial channel 1 to 9600 BAUD and wait for port to open
  Serial1.begin(9600);
  }
```

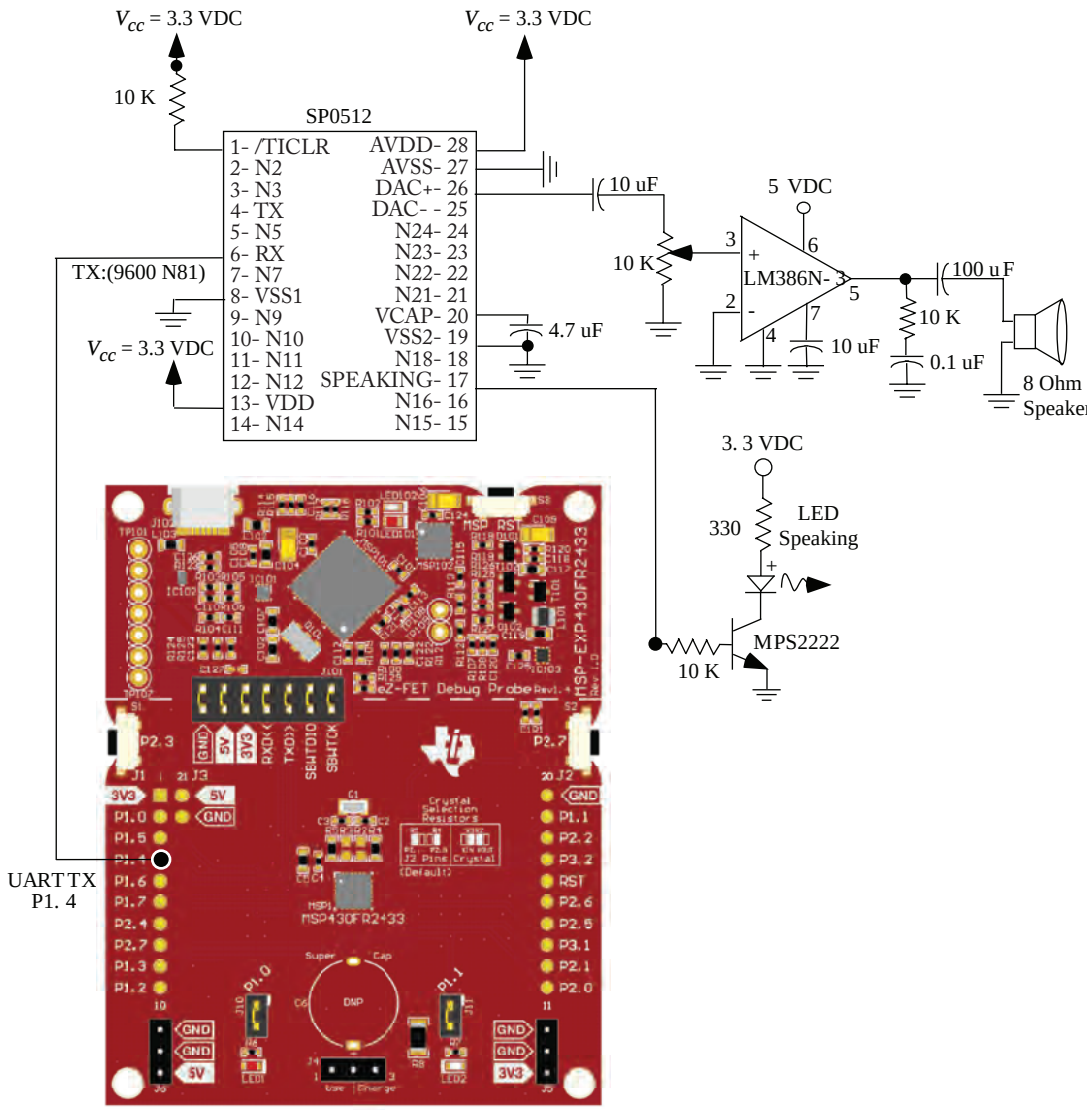


Figure 10.6: Speech synthesis support circuit (www.speechchips.com). (Illustration used with permission of Texas Instruments (www.ti.com).)

```
void loop()
  {
  Serial1.print("[BD]This [BD]is [BD]a [BD]test \r");
  delay(3000);
  }

//******************************************************************
```

10.4.2 UART C EXAMPLE

Example: In this example the MSP430 UART's TX pin is connected to the RX pin. The example shows proper initialization of registers and interrupts to receive and transmit data. If data is incorrect P1.0 LED is turned ON.

```
//******************************************************************
// --COPYRIGHT--,BSD_EX
// Copyright (c) 2015, Texas Instruments Incorporated
// All rights reserved.
//
//                  MSP430 CODE EXAMPLE DISCLAIMER
//
//******************************************************************
//MSP430FR24xx Demo - USCI_A0 External Loopback test @ 115200 baud
//
//Description: This demo connects TX to RX of the MSP430 UART
//The example code shows proper initialization of registers
//and interrupts to receive and transmit data. If data is incorrect,
//P1.0 LED is turned ON.
//  ACLK = n/a, MCLK = SMCLK = BRCLK = DCODIV ~1MHz.
//
//                MSP430FR2433
//             -----------------
//         /|\|                 |
//          | |                 |
//          --|RST              |
//            |                 |
//            |                 |
//            |     P1.4/UCA0TXD|----
//            |                 |   |
//            |     P1.5/UCA0RXD|----
//            |                 |
```

```
//            |            P1.0 |--> LED
//            |                  |
//
//Ling Zhu, Texas Instruments Inc., July 2015
//Built with:
//  IAR Embedded Workbench v6.20 & Code Composer Studio v6.0.1
//*******************************************************************

#include <msp430.h>

unsigned char RXData = 0;
unsigned char TXData = 1;

int main(void)
{
WDTCTL = WDTPW | WDTHOLD;               //Stop watchdog timer
PM5CTL0 &= ~LOCKLPM5;                   //Disable the GPIO power-on
                                        //default high-impedance mode
                                        //to activate previously
                                        //configured port settings

P1DIR |= BIT0;
P1OUT &= ~BIT0;                         //P1.0 out low
                                        //Configure UART pins
P1SEL0 |= BIT4 | BIT5;                  //set 2-UART pin as second
                                        //function
                                        //Configure UART
UCA0CTLW0 |= UCSWRST;                   //Put eUSCI in reset
UCA0CTLW0 |= UCSSEL__SMCLK;

                                        //Baud Rate calculation
UCA0BR0 = 8;                            //1000000/115200 = 8.68
UCA0MCTLW = 0xD600;                     //1000000/115200
                                        //INT(1000000/115200)=0.68
                                        //UCBRSx value = 0xD6 (See UG)

UCA0BR1 = 0;
UCA0CTLW0 &= ~UCSWRST;                  //Initialize eUSCI
UCA0IE |= UCRXIE;                       //Enable USCI_A0 RX interrupt

while (1)
  {
```

```
  while(!(UCA0IFG & UCTXIFG));
  UCA0TXBUF = TXData;                       //Load data onto buffer
  __bis_SR_register(LPM0_bits|GIE);         //Enter LPM0
  __no_operation();                         //For debugger
    }
  }

//*********************************************************************
#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector=USCI_A0_VECTOR
__interrupt void USCI_A0_ISR(void)
#elif defined(__GNUC__)
void __attribute__ ((interrupt(USCI_A0_VECTOR))) USCI_A0_ISR (void)
#else
#error Compiler not supported!
#endif
{
switch(__even_in_range(UCA0IV,USCI_UART_UCTXCPTIFG))
  {
  case USCI_NONE: break;
  case USCI_UART_UCRXIFG:
       UCA0IFG &=~ UCRXIFG;                 //Clear interrupt
       RXData = UCA0RXBUF;                  //Clear buffer
       if(RXData != TXData)                 //Check value
         {
         P1OUT |= BIT0;                     //If incorrect turn on P1.0
         while(1);                          //trap CPU
         }
       TXData++;                            //increment data byte
       __bic_SR_register_on_exit(LPM0_bits); // Exit LPM0 on reti
       break;
   case USCI_UART_UCTXIFG: break;
   case USCI_UART_UCSTTIFG: break;
   case USCI_UART_UCTXCPTIFG: break;
   }
}
//*********************************************************************
```

10.5 SERIAL PERIPHERAL INTERFACE-SPI

The SPI is also used for two-way serial communication between a transmitter and a receiver. In the SPI system, the transmitter and receiver pair shares a common clock source (UCxCLK). This requires an additional clock line between the transmitter and the receiver but allows for higher data transmission rates as compared to the UART. The SPI system allows for fast and efficient data exchange between microcontrollers or peripheral devices. There are many SPI compatible external systems available to extend the features of the microcontroller. For example, a liquid crystal display or a multi-channel DAC could be added to the microcontroller using the SPI system.

10.5.1 SPI OPERATION

The SPI may be viewed as a synchronous 16-bit shift register with an 8-bit, half residing in the transmitter and the other 8-bit half residing in the receiver, as shown in Figure 10.7. The transmitter is designated as the master since it provides the synchronizing clock source between the transmitter and the receiver. The receiver is designated as the slave. A slave is chosen for reception by taking its slave select ($\overline{SS}$) line low. When the $\overline{SS}$ line is taken low, the slave's register shifting capability is enabled.

SPI transmission is initiated by loading a data byte into the master-configured transmit buffer (UCxTXBUF). At that time, the UCSI SPI mode Bit Clock Generator provides clock pulses to the master and also to the slave via the UCxCLK pin. A single bit is shifted out of the master designated shift register on the slave in master out (UCxSIMO) microcontroller pin on every SCK pulse. The data is received at the UCxSIMO pin of the slave designated device. In some peripheral devices, this is referred to as master out slave in (MOSI). At the same time, a single bit is shifted out of the slave out master in (UCxSOMI) pin of the slave device and into the UCxSOMI pin of the master device. After eight master UCxCLK clock pulses, a byte of data has been exchanged between the master and slave designated SPI devices. Completion of data transmission in the master and data reception in the slave is signaled by SPI-related interrupts in both devices. At that time, another data byte may be transmitted.

10.5.2 MSP430 SPI FEATURES

As previously mentioned, the MSP430 SPI system has many features that allow the system to be interfaced to a wide variety of SPI configured peripheral devices. These features include [SLAU445G, 2016, SLAU367O, 2017]:

- 7- or 8-bit data length,
- LSB-first or MSB-first data transmit and receive capability,
- 3- or 4-wire SPI operation,
- master or slave modes,

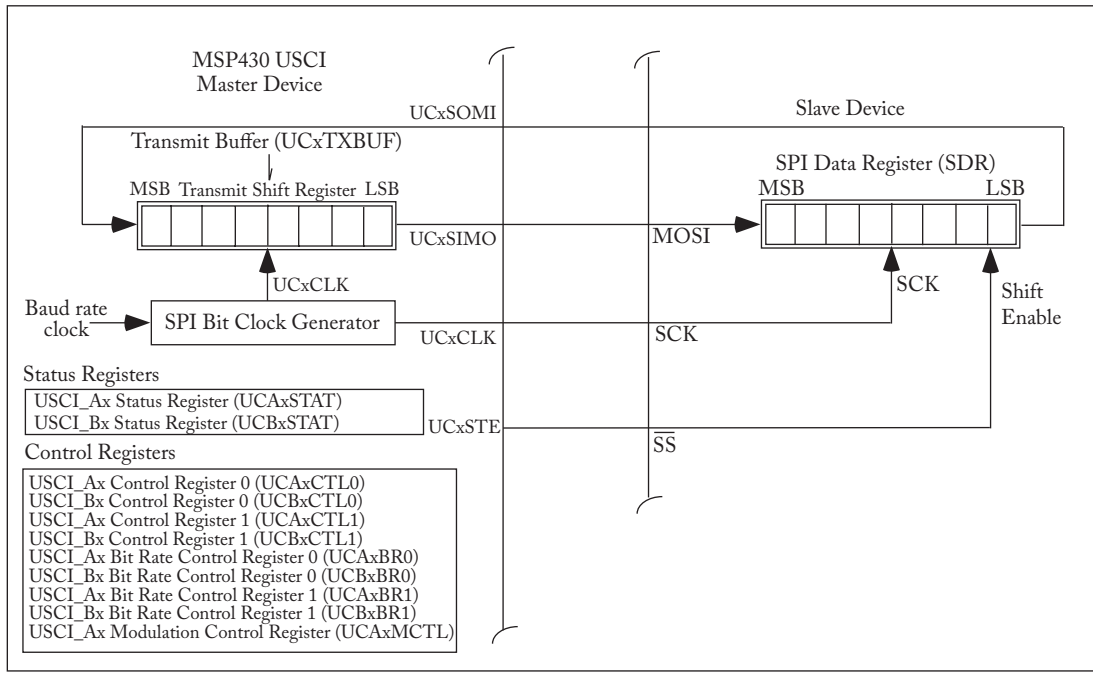


Figure 10.7: SPI overview.

- independent transmit and receive shift registers which provide continuous transmit and receive operation,
- selectable clock polarity and phase control,
- programmable clock frequency in master mode, and
- independent interrupt capability for receive and transmit.

10.5.3 MSP430 SPI HARDWARE CONFIGURATION

The MSP430 provides support for SPI communication in both of the eUSCI_A and eUSCI_B modules. A block diagram of an UCSI module configured for SPI operation is shown in Figure 10.8. SPI operation is selected by setting the UCSYNC (Synchronous mode enable) bit to logic one in the module's eUSCI_Ax or USCI_Bx Control Register 0 (UCAxCTL0 or UCBxCTL0).

Located in the center of the diagram, the clock source for the SPI Baud rate clock (BRCLK) is either provided by the ACLK or the SMCLK. The clock source is chosen using the eUSCI clock source select (UCSSELx) bits in eUSCI_Ax (or B) control register 1 (UCAxCTL1 or UCBxCTL1).

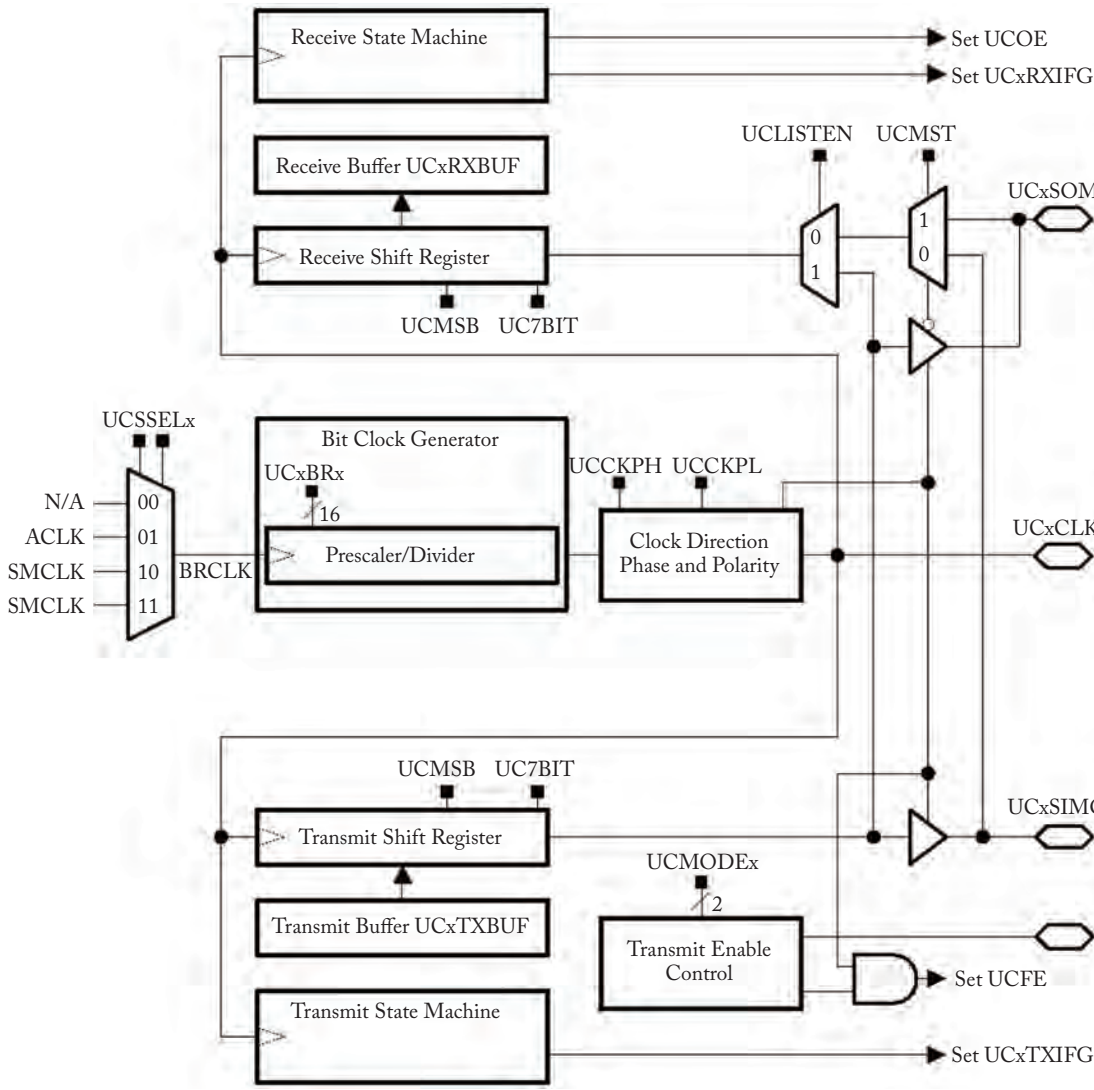


Figure 10.8: SPI hardware overview [SLAU445G, 2016, SLAU367O, 2017]. (Illustration used with permission of Texas Instruments (www.ti.com).)

The Baud rate clock is fed to the Bit Clock Generator. The 16-bit clock prescaler is formed using (UCxxBR0 + UCxxBR1 × 256). The values for UCxxBR0 and UCxxBR1 are contained in the eUSCI_xx Bit Rate Control Registers 0 and 1 (UCxxBR0 and UCxxBR1).

The MSP430 eUSCI provides the flexibility to configure the SPI data transmission format to match that of many different peripheral devices. Either a 7- or 8-bit data format may be selected using the UC7BIT. Also, the phase and polarity of the data stream may be adjusted to match peripheral devices. The polarity setting determines active high or low transmission while the polarity bit determines if the signal is asserted in the first half of the bit frame or in the second half. Furthermore, the data may be transmitted with the LSB first or the MSB first. In summary, the serial data stream format is configured using the following bits in the eUSCI_Ax (or Bx) control register 0 (UCAxCTL0) [SLAU445G, 2016, SLAU367O, 2017]:

- UCCKPH: clock phase select bit - 0: data changed on the first UCLK edge and captured on the following edge; 1: data captured on the first edge and changed on the second edge
- UCCKPL: clock polarity select bit - 0: inactive state low; 1: inactive state high
- UCMSB: MSB first select bit - 0: LSB transmitted first; 1: MSB transmitted first
- UC7BIT: character length select bit - 0: 8-bit data; 1: 7-bit data

The clock signal is routed from the Bit Clock Generator to both the receive state machine and the transmit state machine. To transmit data, the data is loaded to the transmit buffer (UCxTXBUF). Writing to the UCxTXBUF activates the Bit Clock Generator. The data begins to transmit. Also, the SPI-system receives data when the transmission is active. The transmit and receive operations occur simultaneously [SLAU445G, 2016, SLAU367O, 2017].

The SPI system is also equipped with interrupts. The UXTXIFG interrupt flag in the eUSCI_Ax (or Bx) interrupt flag register (UCAxIFG, UCBxIFG) is set when the UCxxTXBUF is empty indicating another character may be transmitted. The UCRXIFG interrupt flag is set when a complete character has been received.

10.5.4 SPI REGISTERS

As discussed throughout this section, the basic features of the SPI system is configured and controlled by the following SPI-related registers [SLAU445G, 2016, SLAU367O, 2017]:

eUSCI_A SPI Registers

- UCAxCTLW0 eUSCI_Ax Control Word 0
- UCAxCTL1 eUSCI_Ax Control 1
- UCAxCTL0 eUSCI_Ax Control 0
- UCAxBRW eUSCI_Ax Bit Rate Control Word

- UCAxBR0 eUSCI_Ax Bit Rate Control 0
- UCAxBR1 eUSCI_Ax Bit Rate Control 1
- UCAxSTATW eUSCI_Ax Status
- UCAxRXBUF eUSCI_Ax Receive Buffer
- UCAxTXBUF eUSCI_Ax Transmit Buffer
- UCAxIE eUSCI_Ax Interrupt Enable
- UCAxIFG eUSCI_Ax Interrupt Flag
- UCAxIV eUSCI_Ax Interrupt Vector

eUSCI_B SPI Registers

- UCBxCTLW0 eUSCI_Bx Control Word 0
- UCBxCTL1 eUSCI_Bx Control 1
- UCBxCTL0 eUSCI_Bx Control 0
- UCBxBRW eUSCI_Bx Bit Rate Control Word
- UCBxBR0 eUSCI_Bx Bit Rate Control 0
- UCBxBR1 eUSCI_Bx Bit Rate Control 1
- UCBxSTATW eUSCI_Bx Status
- UCBxRXBUF eUSCI_Bx Receive Buffer
- UCBxTXBUF eUSCI_Bx Transmit Buffer
- UCBxIE eUSCI_Bx Interrupt Enable
- UCBxIFG eUSCI_Bx Interrupt Flag
- UCBxIV eUSCI_Bx Interrupt Vector

Details of specific register and bits settings are contained in SLAU367O and SLAU445G. Figure 10.9 provides the bit settings of eUSCI_Ax Control Word 0 for basic SPI operation using eUSCI_Ax.

15	14	13	12	11	10	9	8
UCCKPH	UCCKPL	UCMSB	UC7BIT	UCMST	UCMODEx		UCSYNC
rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0
7	6	5	4	3	2	1	0
UCSSELx		Reserved				UCSTEM	UCSWRST
rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-1

Can be modified only when UCSWRST = 1

UCAxCTLW0 Register
UCAxCTLW0[15]: Clock phase select: UCCKPH: 0 = data changed on first edge, captured on second
1 = data is captured on first edge, changed on second
UCAxCTLW0[14]: Clock polarity select: UCCKPL: 0 = inactive state is low, 1 = inactive state high
UCAxCTLW0[13]: MSB first select: UCMSB: 0 = LSB first, 1 = MSB first
UCAxCTLW0[12]: Character length: UC7BIT: 0 = 8-bit, 1 = 7-bit
UCAxCTLW0[11]: Master mode select: UCMST: 0 = slave mode, 1 = master mode
UCAxCTLW0[10-9]: eUSCI_A mode: UCMODEx: 00 = 3-pin SPI, 01 = 4-pin SPI with UCxSTE active high,
10 = 4-pin SPI with UCxSTE active low
UCAxCTLW0[8]: Synchronous mode enable: UCSYNC: 0 = Asynchronous, 1 = Synchronous
UCAxCTLW0[7-6]: eUSCI_A clock source select: UCSSELx: 00 = UCLK, 01 = ACLK, 10 or 11 = SMCLK

Figure 10.9: eUSCI_Ax control word 0 [SLAU445G, 2016, SLAU367O, 2017]. (Illustration used with permission of Texas Instruments (www.ti.com).)

10.5.5 SPI CODE EXAMPLES

Energia

The Energia Integrated Development Environment has several built-in functions to support SPI operations (energia.nu). These include:

- SPI.begin(): initializes SPI bus
- SPI.setBitOrder(direction): may be set for MSBFIRST or LSBFIRST
- SPI.setDataMode(mode): may be set for SPI modes 0 to 3 for compatibility to different peripheral devices. The four different modes include:
  - SPI_MODE0: clock polarity = 0, clock phase = 0
  - SPI_MODE1: clock polarity = 0, clock phase = 1
  - SPI_MODE2: clock polarity = 1, clock phase = 0
  - SPI_MODE3: clock polarity = 1, clock phase = 1
- SPI.setClockDivider: use to divide (slow) the SPI clock. May be set to SPI_CLOCK_DIV2, 4, 8, 16, 32, 64, 128.
- SPI.transfer(data): transmits one byte of data on the SPI bus

Also, the slave select line of the desired device must be taken low for SPI transfer. It is then taken high when the data transfer is complete. In the following examples we demonstrate

how to interface a number of SPI compatible devices to the MSP430FR2433 LaunchPad using the SPI system. The SPI connections are:

- MOSI: LaunchPad pin 15, P2.6
- MISO: LaunchPad pin 14, P2.5
- SCK: LaunchPad pin 7, P2.4
- SS: LaunchPad pin 17, P3.2

Example. In Chapter 2 we used the Energia SPI features to control a one meter, 32 RGB LED strip available from Adafruit (#306) (www.adafruit.com). Recall the red, blue, and green component of each RGB LED was independently set using an eight-bit code. The MSB was logic one followed by seven bits to set the LED intensity (0–127). The component values were sequentially shifted out of the MSP430FR2433 LaunchPad using the serial peripheral interface (SPI) features.

Example. In this example we interface the MSP430FR2433 to Sparkfun's SerLCD display (LCD-14072). The display may be used with UART, SPI, or an I^2C interface. The connections between the microcontroller and display are shown in Figure 10.10. The sample code example illustrates how to display numerals and characters to the display.

```
//*********************************************************************
//LCD_SPI:  Demonstrates use of the MSP430FR2433's SPI system
//with the Sparkfun 16x2 SerLCD (LCD-14072).
//
//LCD connections:
// - RAW: 3.3 VDC
// -   -: Ground
// - Power supply grounds should be connected to common ground
// - Serial Data Out  - LaunchPad pin 15 (MOSI pin), P2.6 - to LCD SDI
// - CLK - LaunchPad pin 7 (SCK pin), P2.4 - to LCD SCK
// - Chip Select - LaunchPad pin 18, P2.2 - to LCD /CS
// - Ground - black
//
//
//Notes:
// - SPI must be configured for least significant bit (LSB) first,
//   Mode 0, SPI clock divide 128
//
//This example code is in the public domain.
```

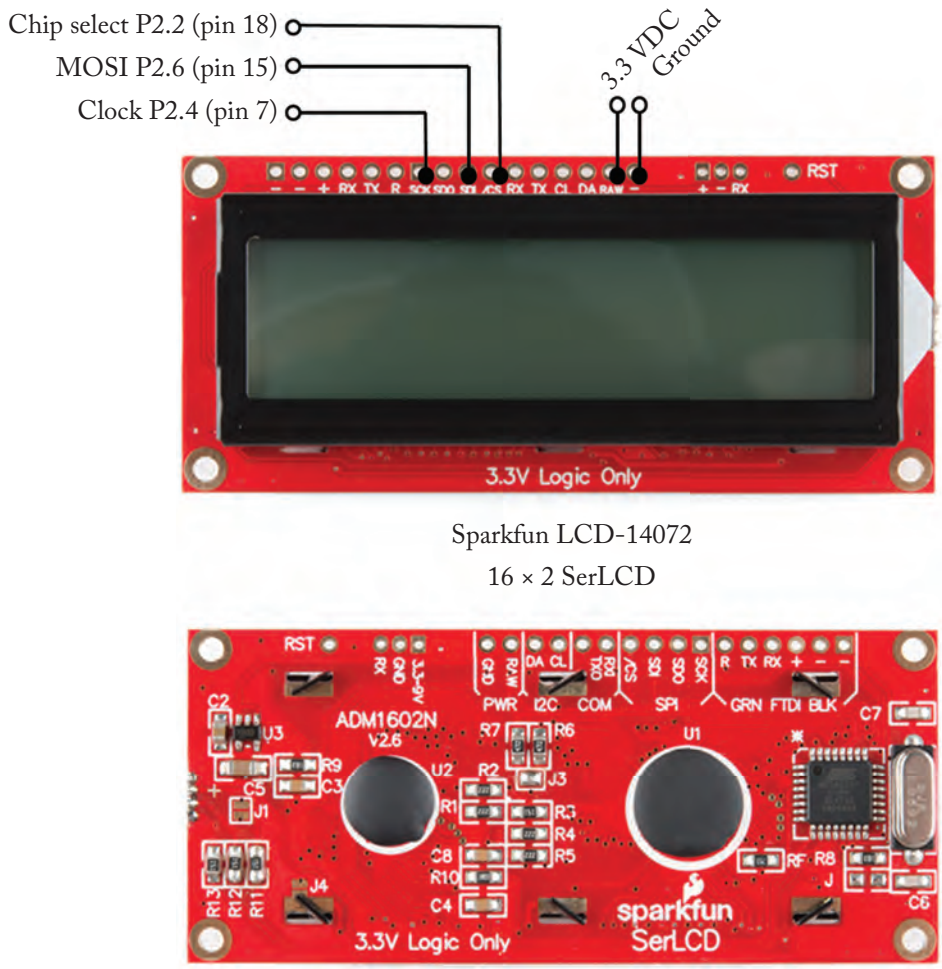


Figure 10.10: MSP430FR2433 interface to Sparkfun's LCD-14072 display. (Illustration used with permission of Sparkfun Electronics (`www.sparkfun.com`).)

```
//*********************************************************************

#include <SPI.h>

#define  chip_select    18          //LaunchPad pin P2.2

unsigned int  numeral, number, i;
unsigned char printstring[16] = "Hello World";

void setup()
  {
  pinMode(chip_select, OUTPUT);
  SPI.begin();                           //SPI support functions
  SPI.setBitOrder(MSBFIRST);             //SPI bit order - MSB first
  SPI.setDataMode(SPI_MODE0);            //SPI mode
  SPI.setClockDivider(SPI_CLOCK_DIV128);//SPI data clock rate
  }

void loop()
{
digitalWrite(chip_select, HIGH);        //chip select high

digitalWrite(chip_select, LOW);         //chip select assert
SPI.transfer(0x7C);                     //enter LCD settings mode
SPI.transfer(0x2D);                     //clear display, cursor home
digitalWrite(chip_select, HIGH);        //initialize chip select

for(numeral = 0; numeral<=9; numeral++)
  {
  number = numeral + 48;                //convert to ASCII
  digitalWrite(chip_select, LOW);       //chip select assert
  SPI.transfer(number);                 //transmit data via SPI
  digitalWrite(chip_select, HIGH);      //chip select high
  delay(1000);                          //1s delay
  } //end for

digitalWrite(chip_select, HIGH);        //chip select high

for(i=0; i<=5; i++)                     //advance to line 2
```

```
  {
  digitalWrite(chip_select, LOW);       //chip select assert
  SPI.transfer(' ');
  digitalWrite(chip_select, HIGH);      //initialize chip select
  }
                                        //write characters to line 2
digitalWrite(chip_select, LOW);         //chip select assert
for(i=0; printstring[i] != '\0'; i++)
  {
  SPI.transfer(printstring[i]);         //transmit data via SPI
  delay(1000);                          //1s delay
  } //end for
digitalWrite(chip_select, HIGH);        //chip select high

}  //end void

//*****************************************************************
```

Example. In this example the MSP430 SPI system is used to send numerical data to Sparkfun's 6.5" 7-segment displays (COM-08530). Two of the large digit displays are serially linked together via Sparkfun's Large Digit Driver (WIG-13279). The Large Digit Driver contains a Texas Instruments TPIC6C696 Power Logic 8-bit Shift Register [TPIC6C596, 2015]. The Large Digit Drivers are soldered to the back of the 6.5" 7-segment displays. The hardware configuration is shown in Figure 10.11.

Numerical data is shifted out of the MSP430FR2433 to the TPIC6C696 shift register within the Large Digit Driver (WIG-13279). In the first code example, LED_big_digit1, a single display is sent an incrementing value from 0–9. The numerals are coded to match the requirements of the Large Digit Driver.

In the second code example, LED_big_digit2, two displays are sent an incrementing value from 00–99.

```
//*******************************************************************
//LED_big_digit1:  Demonstrates use of the MSP430FR2433's SPI system
//to illuminate different numbers on Sparkfun's 6.5" 7-segment display
//(COM-08530).  Numerals are sent from the MSP430 to Sparkfun's Large
//Digit Driver (WIG-13279).
//
//WIG-13279 pin connections:
// - External 12 VDC supply - red
// - External 5 VDC supply - orange
```

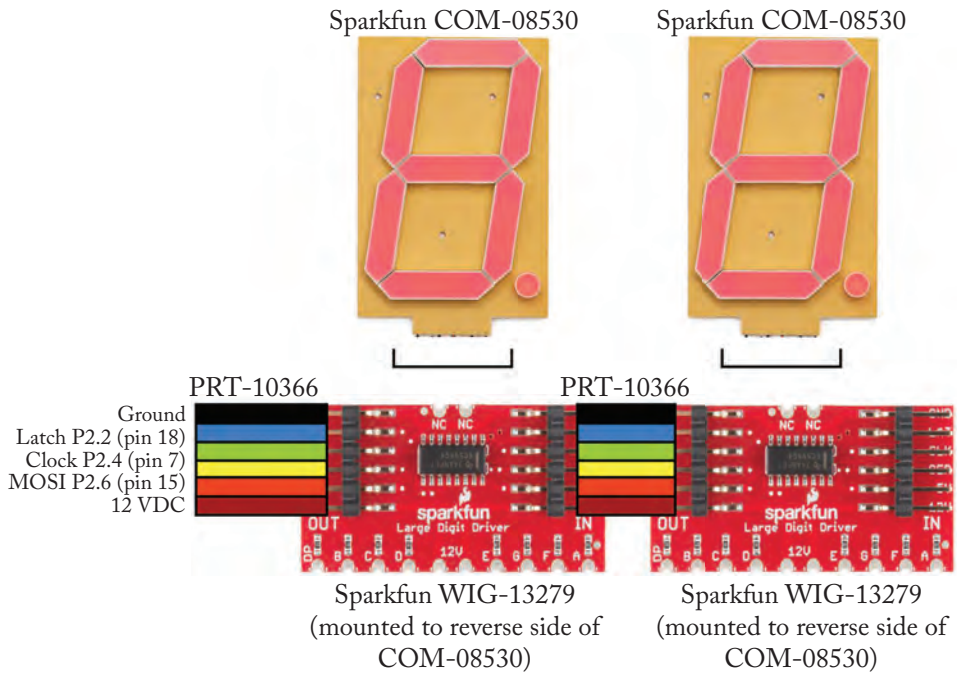


Figure 10.11: MSP430FR2433 interface to Sparkfun's 6.5" 7-segment displays (COM-08530). (Illustration used with permission of Sparkfun Electronics (www.sparkfun.com).)

```
// - Power supply grounds should be connected to common ground
// - Serial Data Out  - LaunchPad pin 15 (MOSI pin), P2.6 - yellow
// - CLK - LaunchPad pin 7 (SCK pin), P2.4 - green
// - Latch - LaunchPad pin 18, P2.2 - blue
// - Ground - black
//
//
//Notes:
// - SPI must be configured for least significant bit (LSB) first
// - The numerals 0 to 9 require the following data words as required
//   by the interface between the Spakfun Large Digit Driver (WIG-13279)
//   and the Sparkfun 6.5" 7-segment display (COM-08530).
//
//   Numeral          Data representation of numeral
//     0                        0xDE
//     1                        0x06
```

```
//     2                          0xBA
//     3                          0xAE
//     4                          0x66
//     5                          0xEC
//     6                          0xFC
//     7                          0x86
//     8                          0xFE
//     9                          0xE6
//
//This example code is in the public domain.
//************************************************************************

#include <SPI.h>

//Seven-segment numeral code
#define  seven_seg_zero         0xDE
#define  seven_seg_one          0x06
#define  seven_seg_two          0xBA
#define  seven_seg_three        0xAE
#define  seven_seg_four         0x66
#define  seven_seg_five         0xEC
#define  seven_seg_six          0xFC
#define  seven_seg_seven        0x86
#define  seven_seg_eight        0xFE
#define  seven_seg_nine         0xE6

#define  LATCH    18               //LaunchPad pin P2.2

const byte    strip_length = 1;     //number of 7-segment LEDs
unsigned char troubleshooting = 0;  //allows printouts to serial
unsigned int  numeral;
unsigned char segment_data;

void setup()
  {
  pinMode(LATCH, OUTPUT);
  SPI.begin();                          //SPI support functions
  SPI.setBitOrder(LSBFIRST);            //SPI bit order - LSB first
  SPI.setDataMode(SPI_MODE3);           //SPI mode
```

```
  SPI.setClockDivider(SPI_CLOCK_DIV32);//SPI data clock rate
  Serial.begin(9600);                   //serial comm at 9600 bps
  }

void loop()
{
digitalWrite(LATCH, LOW);             //initialize LATCH signal
SPI.transfer(seven_seg_zero);         //reset to zero
assert_latch();

for(numeral = 0; numeral<=9; numeral++)
  {
  switch(numeral)                     //convert numeral to
    {                                 //7-segment code
    case 0: segment_data = seven_seg_zero;  break;
    case 1: segment_data = seven_seg_one;   break;
    case 2: segment_data = seven_seg_two;   break;
    case 3: segment_data = seven_seg_three; break;
    case 4: segment_data = seven_seg_four;  break;
    case 5: segment_data = seven_seg_five;  break;
    case 6: segment_data = seven_seg_six;   break;
    case 7: segment_data = seven_seg_seven; break;
    case 8: segment_data = seven_seg_eight; break;
    case 9: segment_data = seven_seg_nine;  break;
    default: break;
    }

  SPI.transfer(segment_data);            //transmit data via SPI
  assert_latch();

  delay(1000);                        //1s delay

  if(troubleshooting)
    {
    Serial.println(numeral, DEC);
    Serial.println(" ");
    }
  }
}
```

```
//*****************************************************************

void assert_latch()
{
digitalWrite(LATCH, HIGH);            //transmit latch pulse
delay(50);
digitalWrite(LATCH, LOW);             //initialize LATCH signal
}

//*****************************************************************
```

The two-digit example follows.

```
//*********************************************************************
//LED_big_digit2:  Demonstrates use of the MSP430FR2433's SPI system
//to illuminate different numbers on Sparkfun's 6.5" 7-segment display
//(COM-08530).  Numerals are sent from the MSP430 to Sparkfun's Large
//Digit Driver (WIG-13279).
//
//WIG-13279 pin connections:
// - External 12 VDC supply - red
// - External 5 VDC supply - orange
// - Power supply grounds should be connected to common ground
// - Serial Data Out  - LaunchPad pin 15 (MOSI pin), P2.6 - yellow
// - CLK - LaunchPad pin 7 (SCK pin), P2.4 - green
// - Latch - LaunchPad pin 18, P2.2 - blue
// - Ground - black
//
//
//Notes:
// - SPI must be configured for least significant bit (LSB) first
// - The numerals 0 to 9 require the following data words as required
//   by the interface between the Spakfun Large Digit Driver (WIG-13279)
//   and the Sparkfun 6.5" 7-segment display (COM-08530).
//
//   Numeral          Data representation of numeral
//     0                        0xDE
//     1                        0x06
//     2                        0xBA
```

```
//     3                          0xAE
//     4                          0x66
//     5                          0xEC
//     6                          0xFC
//     7                          0x86
//     8                          0xFE
//     9                          0xE6
//
//This example code is in the public domain.
//*****************************************************************

#include <SPI.h>

//Seven-segment numeral code
#define  seven_seg_zero         0xDE
#define  seven_seg_one          0x06
#define  seven_seg_two          0xBA
#define  seven_seg_three        0xAE
#define  seven_seg_four         0x66
#define  seven_seg_five         0xEC
#define  seven_seg_six          0xFC
#define  seven_seg_seven        0x86
#define  seven_seg_eight        0xFE
#define  seven_seg_nine         0xE6

#define  LATCH    18               //LaunchPad pin P2.2

const byte    strip_length = 1;     //number of 7-segment LEDs
unsigned char troubleshooting = 0;  //allows printouts to serial
unsigned int  numeral, first_digit, second_digit;
unsigned char segment_data_return;

void setup()
  {
  pinMode(LATCH, OUTPUT);
  SPI.begin();                          //SPI support functions
  SPI.setBitOrder(LSBFIRST);            //SPI bit order - LSB first
  SPI.setDataMode(SPI_MODE3);           //SPI mode
  SPI.setClockDivider(SPI_CLOCK_DIV32);//SPI data clock rate
```

```
  Serial.begin(9600);                  //serial comm at 9600 bps
  }

void loop()
{
digitalWrite(LATCH, LOW);             //initialize LATCH signal
SPI.transfer(seven_seg_zero);         //reset to zero
assert_latch();

for(numeral = 0; numeral<=99; numeral++)
  {
  if(numeral <= 9)
    {
    segment_data_return = determine_segments(numeral);
    SPI.transfer(segment_data_return);     //transmit data via SPI
    SPI.transfer(seven_seg_zero);
    assert_latch();
    delay(1000);                           //1s delay
    } //end if
  else //numeral >=10 - two digit analysis
    {
    first_digit = numeral
    second_digit = (int)((numeral-first_digit)/10);
    segment_data_return = determine_segments(first_digit);
    SPI.transfer(segment_data_return);     //transmit data via SPI
    segment_data_return = determine_segments(second_digit);
    SPI.transfer(segment_data_return);     //transmit data via SPI
    assert_latch();
    delay(1000);                           //1s delay
    }//end else
  }  //end for
}  //end void

//*****************************************************************

void assert_latch()
{
digitalWrite(LATCH, HIGH);            //transmit latch pulse
delay(50);
```

```
digitalWrite(LATCH, LOW);              //initialize LATCH signal
}

//*************************************************************

unsigned char determine_segments(unsigned int segment_number)
{

unsigned char segment_data;

switch(segment_number)                 //convert numeral to
      {                                //7-segment code
      case 0: segment_data = seven_seg_zero;  break;
      case 1: segment_data = seven_seg_one;   break;
      case 2: segment_data = seven_seg_two;   break;
      case 3: segment_data = seven_seg_three; break;
      case 4: segment_data = seven_seg_four;  break;
      case 5: segment_data = seven_seg_five;  break;
      case 6: segment_data = seven_seg_six;   break;
      case 7: segment_data = seven_seg_seven; break;
      case 8: segment_data = seven_seg_eight; break;
      case 9: segment_data = seven_seg_nine;  break;
      default: break;
      }

return segment_data;

}

//*************************************************************
```

Example. In this example we interface the MSP430FR2433 LaunchPad to the Sparkfun Real Time Clock (BOB-10160) and the Sparkfun 16x2 SerLCD (LCD-14072). Both devices are interfaced to the MSP430FR2433 LaunchPad via the same SPI channel. The SPI channel is time shared between the RTC and the LCD by using different device select lines as shown in Figure 10.12. The two different devices use different SPI modes.

The Sparkfun RTC (BOB-10160) breakout board hosts the Maxim Integrated DS3234 RTC [DS3234, 2015]. Once set the RTC tracks date and time. The RTC has two sets of registers to read and write the time values. The read registers span RTC address space from 0x00 to 0x0D while the write registers span from 0x80 to 0x8D. DS3234 register details are available

in the DS3234 data sheet. Date and time information is stored in the DS3234 in BCD format. The code example demonstrates how to covert from decimal representation to BCD format for storage within the DS3234. The DS3234 is configured for operation via a control register at 0x8E (Maxim DS3234 [2015]).

In the example the RTC is initialized with the current date and time. This is only accomplished once. The RTC continues to monitor time while on battery power. To use the software to read the RTC at a later time simply comment out the "set_RTC" function.

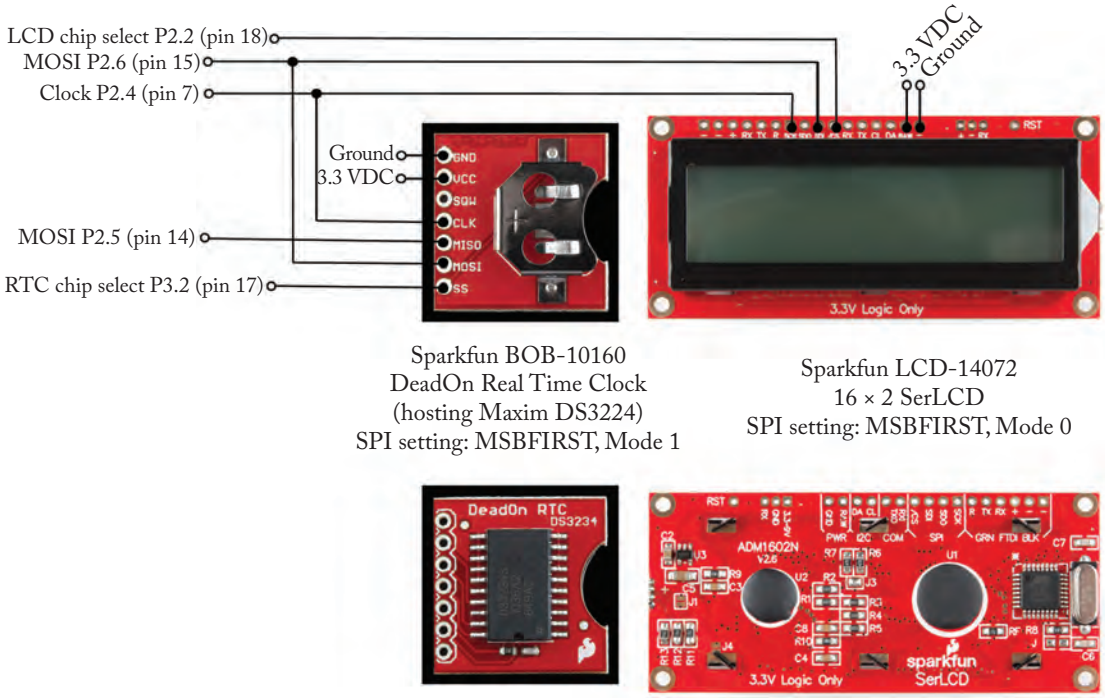


Figure 10.12: MSP430FR2433 interface to Sparkfun's real time clock (BOB-10160). (Illustration used with permission of Sparkfun Electronics (www.sparkfun.com).)

```
//*****************************************************************************
//SPI_RTC_LCD:  Demonstrates use of the MSP430FR2433's LaunchPad
//SPI system with Sparkfun's Real Time Clock (BOB-10160) hosting
//Maxim's DS3224 (Extremely Accurate SPI Bus RTC with Integrated
//Crystal and SRAM). RTC time data is displayed on Sparkfun's
//16x2 SerLCD (LCD-14072).
//
//Once the RTC time is initially set, the RTC will maintain
```

```
//clock time while on battery power.
//
//RTC SPI settings: MSBFIRST, Mode 1
//RTC connections:
// - VCC: 3.3 VDC
// - GND: Ground
// - Power supply ground should be connected to common ground
// - MOSI: LaunchPad pin 15 (MOSI pin), P2.6
// - MISO: LaunchPad pin 14 (MISO pin), P2.5
// -  CLK: LaunchPad pin 7 (SCK pin), P2.4
// -   SS: RTC Chip Select - LaunchPad pin 17, P3.2
//
//LCD SPI settings: MSBFIRST, Mode 0
//LCD connections:
// - RAW: 3.3 VDC
// -   -: Ground
// - Power supply ground should be connected to common ground
// - SDI: LaunchPad pin 15 (MOSI pin), P2.6
// - SCK: LaunchPad pin 7 (SCK pin), P2.4
// - /SS: LCD Chip Select - LaunchPad pin 18, P2.2
//
//Note: RTC and LCD requires different SPI mode settings.
//
//RTC code adapted from code provided by Jim Lindblom of
//SparkFun Electronics (www.sparkfun.com)
//
//This example code is in the public domain.
//*****************************************************************

#include <SPI.h>

#define  RTC_chip_select    17          //LaunchPad pin P3.2
#define  LCD_chip_select    18          //LaunchPad pin P2.2

unsigned int  i;
unsigned char printstring[17] = "   DS3234 RTC   ";
int TimeDateGroup[7];
String TDG;
```

```
void setup()
{
pinMode(LCD_chip_select, OUTPUT);
pinMode(RTC_chip_select, OUTPUT);
Serial.begin(9600);

initialize_RTC();                     //Initialize RTC
                                      //int day    (1 to 31)
                                      //int month  (1 to 12)
                                      //int year   (0 to 99)
                                      //int hour   (0 to 23)
                                      //int minute (0 to 59)
                                      //int second (0 to 59)
set_RTC(12, 10, 18, 5, 0, 0);         //set current time in RTC
}

void loop()
{
digitalWrite(LCD_chip_select, HIGH);  //LCD chip select de-assert
digitalWrite(RTC_chip_select, HIGH);  //RTC chip select de-assert
TDG = ReadTimeDate();                 //get current time
Serial.println(TDG);                  //display time to serial mon
clear_LCD();                          //clear LCD, cursor to home
                                      //Display RTC banner on LCD
SPI.setDataMode(SPI_MODE0);           //SPI mode
digitalWrite(LCD_chip_select, LOW);   //LCD chip select assert
for(i=0; printstring[i] != '\0'; i++)
  {
  SPI.transfer(printstring[i]);       //transmit data via SPI
  delay(100);                         //100 ms delay
  } //end for
digitalWrite(LCD_chip_select, HIGH);  //chip select high

display_time_LCD();
delay(3000);
}  //end void

//*************************************************************
```

```
void clear_LCD(void)
{
SPI.begin();                            //SPI support functions
SPI.setBitOrder(MSBFIRST);              //SPI bit order - MSB first
SPI.setDataMode(SPI_MODE0);             //SPI mode
SPI.setClockDivider(SPI_CLOCK_DIV128);//SPI data clock rate
delay(10);
digitalWrite(LCD_chip_select, LOW);   //LCD chip select assert
SPI.transfer(0x7C);                     //enter LCD settings mode
SPI.transfer(0x2D);                     //clear display, cursor home
digitalWrite(LCD_chip_select, HIGH);  //LCD chip select de-assert
}

//*****************************************************************

void initialize_RTC(void)
{
SPI.begin();                            //SPI support functions
SPI.setBitOrder(MSBFIRST);              //SPI bit order - MSB first
SPI.setDataMode(SPI_MODE1);             //SPI mode
SPI.setClockDivider(SPI_CLOCK_DIV128);//SPI data clock rate
delay(10);

digitalWrite(RTC_chip_select, LOW);   //RTC chip select assert
SPI.transfer(0x8E);                     //0x8E: RTC control register
SPI.transfer(0x60);                     //turn on RTC clock
digitalWrite(RTC_chip_select, HIGH);  //RTC chip select de-assert
}

//*****************************************************************
//RTC code adapted from RTC code provided by Jim Lindblom of
//SparkFun Electronics (www.sparkfun.com)
//*****************************************************************

void set_RTC(int day, int mon, int yr, int hr, int mn, int sec)
{
                                        //assemble TimeDateGroup
int TimeDateGroup[7] = {sec, mn, hr, 0, day, mon, yr};
```

```
SPI.begin();                            //SPI support functions
SPI.setBitOrder(MSBFIRST);              //SPI bit order - MSB first
SPI.setDataMode(SPI_MODE1);             //SPI mode
SPI.setClockDivider(SPI_CLOCK_DIV128);//SPI data clock rate
delay(10);

//Parse out portions of TimeDateGroup.  Convert digits of each portion
//of TimeDateGroup to BCD.  Transmit BCD charara ers to DS3234
//Timekeeping registers. Reference Table 1 of DS3234 data sheet.
//The DS3234 write registers begin at address location 0x80.

for(int i=0; i<=6; i++)
  {
  if(i==3) i++;                         //skip over position three
                                        //isolate bit positions
  int b= TimeDateGroup[i]/10;           //10's place
  int a= TimeDateGroup[i]-b*10;         //1's place

  if(i==2)
    {
    if(b==2)
      b=B00000010;
    else if (b==1)
      b=B00000001;
    }
                                        //assemble BCD digits to
                                        //required storage
                                        //configuration
  TimeDateGroup[i]= a+(b<<4);

  digitalWrite(RTC_chip_select, LOW); //RTC chip select assert
  SPI.transfer(i+0x80);
  SPI.transfer(TimeDateGroup[i]);
  digitalWrite(RTC_chip_select, HIGH);//RTC chip select assert
  }
}

//*************************************************************
//RTC code adapted from RTC code provided by Jim Lindblom of
```

```
//SparkFun Electronics (www.sparkfun.com)
//*************************************************************

String ReadTimeDate()
{
String temp;

//Assemble portions of TimeDateGroup from DS3234 memory.
//Read BCD chararaters from DS3234 Timekeeping registers.
//Reference Table 1 of DS3234 data sheet. The DS3234 read
//registers begin at address location 0x00.
SPI.begin();                          //SPI support functions
SPI.setBitOrder(MSBFIRST);            //SPI bit order - MSB first
SPI.setDataMode(SPI_MODE1);           //SPI mode
SPI.setClockDivider(SPI_CLOCK_DIV128);//SPI data clock rate
delay(10);

for(int i=0; i<=6;i++)
  {
  if(i==3) i++;                       //skip position 3

  digitalWrite(RTC_chip_select, LOW); //RTC chip select assert
  SPI.transfer(i+0x00);               //read DS3234 register
  unsigned int n = SPI.transfer(0x00);
  digitalWrite(RTC_chip_select, HIGH);//RTC chip select assert

  int a=n & B00001111;                //assemble BCD digits
  if(i==2)                            //null space
    {
    int b=(n & B00110000)>>4;         //24 hour mode
    if(b==B00000010)
      b=20;
    else if(b==B00000001)
      b=10;
    TimeDateGroup[i]=a+b;
    }
  else if(i==4)                       //days
    {
    int b=(n & B00110000)>>4;
```

```
    TimeDateGroup[i]=a+b*10;
    }
  else if(i==5)                         //month
    {
    int b=(n & B00010000)>>4;
    TimeDateGroup[i]=a+b*10;
    }
  else if(i==6)                         //year
    {
    int b=(n & B11110000)>>4;
    TimeDateGroup[i]=a+b*10;
    }
  else
    {
    int b=(n & B01110000)>>4;
    TimeDateGroup[i]=a+b*10;
    }
  }
  temp.concat(TimeDateGroup[4]);        //day
  temp.concat("/") ;
  temp.concat(TimeDateGroup[5]);        //month
  temp.concat("/") ;
  temp.concat(TimeDateGroup[6]);        //year
  temp.concat("     ") ;
  temp.concat(TimeDateGroup[2]);        //hour
  temp.concat(":") ;
  temp.concat(TimeDateGroup[1]);        //minute
  temp.concat(":") ;
  temp.concat(TimeDateGroup[0]);        //seconds
  return(temp);
}

//*****************************************************************

void display_time_LCD(void)
{

for(int i=4; i<=6; i++)
  {
```

```
  int b= TimeDateGroup[i]/10;          //10's place
  int a= TimeDateGroup[i]-b*10;        //1's place
                                       //convert to ASCII
  b = b + 48;
  a = a + 48;

  digitalWrite(LCD_chip_select, LOW); //LCD chip select assert
  SPI.transfer(b);
  digitalWrite(LCD_chip_select, HIGH);//LCD chip select high

  digitalWrite(LCD_chip_select, LOW); //LCD chip select assert
  SPI.transfer(a);
  digitalWrite(LCD_chip_select, HIGH);//LCD chip select high
  }

  digitalWrite(LCD_chip_select, LOW); //LCD chip select assert
  SPI.transfer(0x20);
  digitalWrite(LCD_chip_select, HIGH);//LCD chip select high

for(int i=2; i>=0; i--)
  {
  int b= TimeDateGroup[i]/10;          //10's place
  int a= TimeDateGroup[i]-b*10;        //1's place
                                       //convert to ASCII
  b = b + 48;
  a = a + 48;

  digitalWrite(LCD_chip_select, LOW); //LCD chip select assert
  SPI.transfer(b);
  digitalWrite(LCD_chip_select, HIGH);//LCD chip select high

  digitalWrite(LCD_chip_select, LOW); //LCD chip select assert
  SPI.transfer(a);
  digitalWrite(LCD_chip_select, HIGH);//LCD chip select high

  if (i !=0)
    {
    digitalWrite(LCD_chip_select, LOW); //LCD chip select assert
    SPI.transfer(0x3A);
```

```
    digitalWrite(LCD_chip_select, HIGH);//LCD chip select high
    }
  }
}

//*****************************************************************
```

SPI C Example

In this example, code is provided for both the SPI master and the SPI slave configured processor using the SPI 3-wire mode. Incrementing data is sent by the master configured processor starting at 0x01. The slave configured processor received data is expected to be same as the previous transmission $TXData = RXData - 1$. The eUSCI RX interrupt service routine is used to handle communication with the processor (www.ti.com).

Master configured SPI processor code:

```
//*****************************************************************
// --COPYRIGHT--,BSD_EX
// Copyright (c) 2015, Texas Instruments Incorporated
// All rights reserved.
//
//                  MSP430 CODE EXAMPLE DISCLAIMER
//
//*****************************************************************
//MSP430FR243x Demo - eUSCI_A0, SPI 3-Wire Master Incremented Data
//
//Description: SPI master talks to SPI slave using 3-wire mode.
//Incrementing data is sent by the master starting at 0x01. Received
//data is expected to be same as the previous transmission
//TXData = RXData-1.  USCI RX ISR is used to handle communication
//with the CPU, normally in LPM0.
//
//   ACLK = ~32.768kHz, MCLK = SMCLK = DCO ~ 1MHz.  BRCLK = SMCLK/2.
//
//                   MSP430FR2433
//                -----------------
//             /|\|                 |
//              | |                 |
//              --|RST              |
//                |                 |
```

```
//                |             P1.4|-> Data In (UCA0SIMO)
//                |                  |
//                |             P1.5|<- Data OUT (UCA0SOMI)
//                |                  |
//                |             P1.6|-> Serial Clock Out (UCA0CLK)
//
//
//Ling Zhu, Texas Instruments Inc., Sept 2015
//Built with IAR Embedded Workbench v6.20 & Code Composer Studio v6.0.1
//******************************************************************

#include <msp430.h>

unsigned char RXData = 0;
unsigned char TXData;

int main(void)
{
WDTCTL = WDTPW | WDTHOLD;               //Stop watchdog timer
P1SEL0 |= BIT4 | BIT5 | BIT6;           //set 3-SPI pin as second func

UCA0CTLW0 |= UCSWRST;                   //**Put state machine in reset**
UCA0CTLW0 |= UCMST|UCSYNC|UCCKPL|UCMSB; //3-pin, 8-bit SPI master
                                        //Clock polarity high, MSB
UCA0CTLW0 |= UCSSEL__SMCLK;               //SMCLK
UCA0BR0 = 0x01;                         // /2,fBitClock=fBRCLK/(UCBRx+1).
UCA0BR1 = 0;                              //
UCA0MCTLW = 0;                            //No modulation
UCA0CTLW0 &= ~UCSWRST;                    //**Init USCI state machine**
UCA0IE |= UCRXIE;                         //Enable USCI_A0 RX interrupt
TXData = 0x01;                            //Holds TX data

PM5CTL0 &= ~LOCKLPM5;                     //Disable the GPIO power-on
                                          //default high-impedance mode
                                          //to activate previously

//configured port settings
while(1)
  {
  UCA0IE |= UCTXIE;                       //Enable TX interrupt
```

```
  __bis_SR_register(LPM0_bits | GIE);     //enable global interrupts,
                                          //enter LPM0
  __no_operation();                       //For debug, remain in LPM0
  __delay_cycles(2000);                   //Delay before next transmis
  TXData++;                               //Increment transmit data
  }
}

//******************************************************************************
#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector=USCI_A0_VECTOR
__interrupt void USCI_A0_ISR(void)
#elif defined(__GNUC__)

void __attribute__ ((interrupt(USCI_A0_VECTOR))) USCI_A0_ISR (void)
#else
#error Compiler not supported!
#endif
{
   switch(__even_in_range(UCA0IV,USCI_SPI_UCTXIFG))
     {
     case USCI_NONE: break;               //Vector 0 - no interrupt
     case USCI_SPI_UCRXIFG:  RXData = UCA0RXBUF;
                             UCA0IFG &= ~UCRXIFG;
                                          //Wake up to setup next TX
                             __bic_SR_register_on_exit(LPM0_bits);

                             break;

                                          //Transmit characters
     case USCI_SPI_UCTXIFG:  UCA0TXBUF = TXData;
                             UCA0IE &= ~UCTXIE;
                             break;

     default:                break;
    }
}

//******************************************************************************
```

Slave configured SPI processor code:

```
//******************************************************************************
// --COPYRIGHT--,BSD_EX
// Copyright (c) 2015, Texas Instruments Incorporated
// All rights reserved.
//
//                  MSP430 CODE EXAMPLE DISCLAIMER
//
//******************************************************************************
//MSP430FR243x Demo - eUSCI_A0, SPI 3-Wire Slave Data Echo
//
//Description: SPI slave talks to SPI master using 3-wire mode.
//Data received from master is echoed back.
//ACLK = 32.768kHz, MCLK = SMCLK = DCO ~ 1MHz
//
//Note: Ensure slave is powered up before master to prevent delays due to
//      slave initialization
//
//                   MSP430FR2433
//                 -----------------
//             /|\|                 |
//              | |                 |
//              --|RST              |
//                |                 |
//                |             P1.4|<- Data In (UCA0SIMO)
//                |                 |
//                |             P1.5|-> Data OUT (UCA0SOMI)
//                |                 |
//                |             P1.6|<- Serial Clock In (UCA0CLK)
//
//
//Ling Zhu, Texas Instruments Inc., Nov 2015
//Built with IAR Embedded Workbench v6.20 & Code Composer Studio v6.0.1
//******************************************************************************

#include <msp430.h>

int main(void)
{
```

```
WDTCTL = WDTPW|WDTHOLD;                   //Stop watchdog timer

P1SEL0 |= BIT4 | BIT5 | BIT6;             //3-SPI pin as second function
UCA0CTLW0 |= UCSWRST;                     //**State machine in reset**
UCA0CTLW0 |= UCSYNC|UCCKPL|UCMSB;         //3-pin, 8-bit SPI slave
                                          //Clock polarity high, MSB
UCA0CTLW0 |= UCSSEL__SMCLK;               //SMCLK
UCA0BR0 = 0x01;                           // /2,fBitClock =
                                          //fBRCLK/(UCBRx+1).
UCA0BR1 = 0;
UCA0MCTLW = 0;                            //No modulation
UCA0CTLW0 &= ~UCSWRST;                    //**Initialize USCI state
                                          //machine**
UCA0IE |= UCRXIE;                         //Enable USCI_A0 RX interrupt

PM5CTL0 &= ~LOCKLPM5;                     //Disable the GPIO power-on
                                          //default high-impedance mode
                                          //to activate previously
                                          //configured port settings
__bis_SR_register(LPM0_bits | GIE);       //Enter LPM0, enable interrupts
}


//**************************************************************************
#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector=USCI_A0_VECTOR
__interrupt void USCI_A0_ISR(void)
#elif defined(__GNUC__)

void __attribute__ ((interrupt(USCI_A0_VECTOR))) USCI_A0_ISR (void)
#else
#error Compiler not supported!
#endif
{
    while (!(UCA0IFG&UCTXIFG));               // USCI_A0 TX buffer ready?
    UCA0TXBUF = UCA0RXBUF;                    // Echo received data
}

//**************************************************************************
```

10.6 INTER-INTEGRATED COMMUNICATION – I²C MODULE

The MSP430's USCI can be programmed to operate in the I²C communication mode. As discussed earlier in this chapter, the eUSCI_Ax ports are programmed to operate in the UART, IrDA, and SPI communication mode while the eUSCI_Bx ports are used for the I²C and SPI serial communication modes.

The I²C module is a communication system used when multiple serial devices are interconnected using a serial bus. Devices on the bus are designated either as a master or slave device. One of the reasons the I²C serial communication became popular is its flexibility to allow multiple master devices to co-exist on the same bus.

The I²C bus is a two-wire bus with a SDL and the SCL. I²C compatible devices, each with a unique address, are connected to the two-wire bus as either a master device or a slave device. The master device initiates a communication transaction by means of either requesting data from another device or sending data to a designated device. The master device must also provide a clock signal (SCL) to the two-wire bus as shown in Figure 10.13a. Note how each bus line requires a pull-up resistor. The MSP430 eUSCI device allows its I²C communication unit to operate either in the standard mode (100 kbps) or in the fast mode (400 kbps) with either a 7- or 10-bit device addressing. Addressing capability is provided for up to four hardware designated slave devices [SLAU445G, 2016, SLAU367O, 2017].

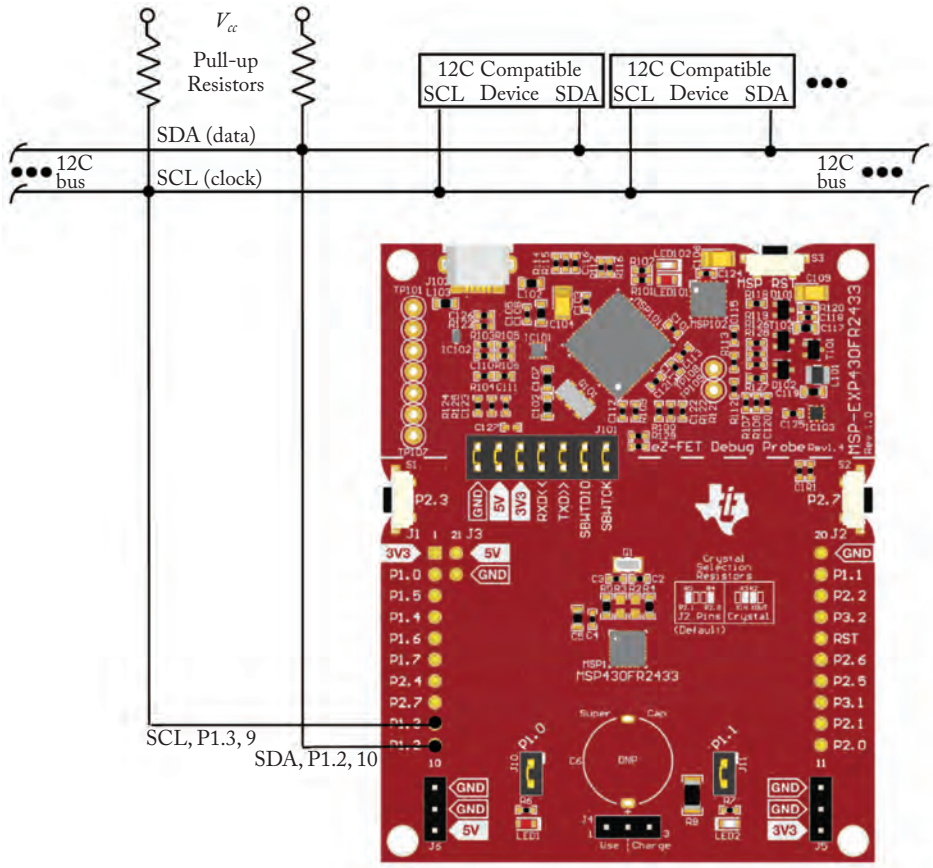
An I²C communication transaction is initiated by the master device. The master device pulls the SDA line low while the SCL line is idling high. The falling edge of the SDA line triggers the transfer. The master then sends a single byte of data containing the desired slave address. The LSB of the address is set for read (1) or write (0). The address is transmitted MSB first. A single bit of the data byte is sent on each SCL clock pulse. A stop condition occurs when the SCL line idles high and at the rising edge of the SDA line. The slave device with the matching address responds appropriately with either a read or write response as shown in Figure 10.13b.

10.6.1 I²C INITIALIZATION

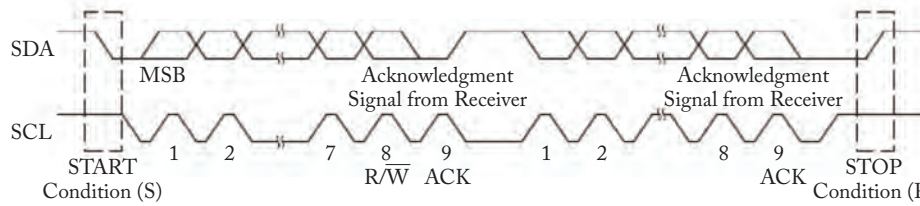
To initialize a eUSCI_Bx port as an I²C communication port, you must: (1) set the UCSWRST bit in the UCxCTL1 register to one, (2) configure the I²C mode of operation by setting UCMODEx bits to 11 and initialize the eUSCI registers, and (3) set up an actual port with a pull-up resistor. As soon as the UCSWRST bit is cleared, the I²C communication of MSP430 can commence.

10.6.2 I²C PROTOCOL

The communication performed on the I²C bus must follow a set of agreed rules, including the data format used on the bus. Data is transferred between devices connected on the bus in 8



(a) 12C bus connection



(b) 12C transaction waveform (SLAU445G)

Figure 10.13: I²C system overview. (Illustration used with permission of Texas Instruments (www.ti.com).)

bits per segment, followed by control bits. Each communication "session" is started by a master device with a start condition, which is defined as the signal changing from logic high to low on the SDA line while the logic state on the SCL line is high. Following the start condition, the master device must send either the 7- or 10-bit unique address of a destination device on the SDA line [SLAU445G, 2016, SLAU367O, 2017].

Following the address, the master device sends a Read/Write bit describing its intent and listens on the bus to hear an acknowledge bit from the receiver on the 9th SCL clock for the 7-bit addressing mode or on the both 9th and 18th clocks for the 10-bit addressing mode.

For the 10-bit addressing mode, the 10-bit address is split into two segments: two MSBs and eight LSBs. The MSBs are sent along with pre-designated bits (11110), and the LSBs are sent separately. After the first part of the address is sent, a Read/Write bit, followed by an acknowledgment bit, must appear on the bus before the second part of the address is sent. After the second part of the address, an acknowledgment bit must appear before data is sent over the bus. Figure 10.14 shows the format of data transfer between two devices, using both the 7-bit and 10-bit addressing modes. For each communication session, it must end with a stop condition (P in the figure), which is defined as the signal state on the SDA line changing from logic low to logic high while the clock signal on the SCL line is high [SLAU445G, 2016, SLAU367O, 2017].

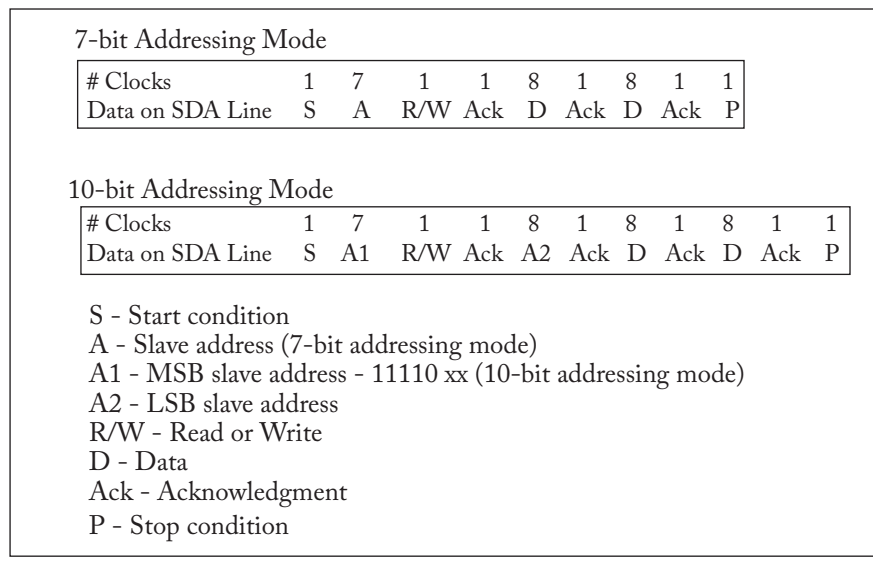


Figure 10.14: Data format for both 7-bit and 10-bit addressing modes.

10.6.3 MSP430 AS A SLAVE DEVICE

The MSP430 microcontroller can also be configured to be either as a slave device or as a master device. To configure the controller as a slave device, the eUSCI_Bx ports must first be programmed to operate in the I^2C slave mode (UCMODEx = 11, UCSYNC = 1, UCMST = 0). The slave address of MSP430 is defined using UCBxI2COA register. The UCA10 bit in the UCBx control register 0 (UCBxCTL0) determines whether the controller is using a 7-bit address or a 10-bit address [SLAU445G, 2016, SLAU367O, 2017].

You can program the MSP430 microcontroller to respond to a general call by setting the general call response enable bit (UCGCEN) in the UCBxI2COA register. To receive device addresses sent by masters, the eUSCI_Bx ports must also be configured in the receiver mode (UCTR = 0). When the start condition is detected on the bus, the address bits are compared, and if there is a match, the UCSTTIFG flag is set [SLAU445G, 2016, SLAU367O, 2017].

After testing that the Read/Write bit is high, MSP430 uses the clock signal on the SCL line to send data on the SDA line. To do so, the UCTR and UCTXIFG bits are set while holding the SCL line logic low. While the logic state on the SCL line is low, the transmit buffer register (UCBxTXBUF) is loaded with data. Once the buffer is loaded, the UCSTTIFG flag is cleared, which sends the data out to the SDA line, and the UCTXIFG flag is automatically set again for the next data to be transmitted, which occurs after an acknowledge bit is detected on the bus. If the not-acknowledge (NACK) bit is detected, followed by a stop condition, instead, the UCSTPIFG flag is set. If the NACK bit is detected followed by a start condition, MSP430 starts to monitor this device address, again, on the SDA line [SLAU445G, 2016, SLAU367O, 2017].

If the MSP430 controller should receive data from a slave device (the Read/Write bit is low), the UCTR bit is cleared, the receive buffer (UCBxRXBUF) is loaded with the data from the bus, and the UCRXIFG flag is set, acknowledging the receipt of the data. Once the data in the bus is read, the flag is cleared, and the controller is ready to receive the next 8-bit data. The controller has an option to send the UCTXNACK bit to a master to release the bus. When a stop condition is detected on the bus, the UCSTPIFG flag is set. If two repeated start conditions are detected or the UCSTPIFG flag is set, the MSP430 terminates its current session and starts monitoring its address on the bus [SLAU445G, 2016, SLAU367O, 2017].

10.6.4 MSP430 AS A MASTER DEVICE

To configure the MSP430 controller to function as a master device, the eUSCI_Bx ports must be programmed to operate in the I^2C mode (UCMODEx = 11, UCSYNC = 1), and one must configure the MSP430 to operate in the master mode by setting the UCMST bit. Since the I^2C bus can handle more than one master device and if there are multiple master devices, the MSP430 needs to be programmed as one of many master devices on the bus by setting the UCMM bit and storing the address (either 7 or 10 bits) of MSP430 in the UCBxI2COA register. As in the case of the slave mode, the address size is determined by the UCA10 bit, and the

general call response is programmed using the UCGCEN bit [SLAU445G, 2016, SLAU367O, 2017].

To initiate a session to transmit data, the UCTR bit and the UCTxSTT bit are set, the UCSLA10 bit is configured to match the slave address size, and the address of a slave device is loaded to the UCBxI2CSA. When the start condition is generated by setting the UCTxSTT bit, the data can be loaded to the UCBxTXBUF, and the UCTxIFG bit is set. Once a slave address acknowledges its address, the UCTxSTT and UCTxIFG bits are cleared. Once the data is sent, the UCTxIFG flag bit is set, again, for the next set of data transfer. To generate a stop condition, set UCTxSTP bit while UCTxIFG and UCTxSTP bits are set. If a repeated start conditions are necessary, set UCTxSTT bit. During a data transfer session, if a slave does not respond (i.e., send acknowledge bits), the MSP430 must either send a stop condition or a repeated start conditions [SLAU445G, 2016, SLAU367O, 2017].

When the MSP430 controller needs to receive data from a slave, the UCTR bit must be cleared, and the UCTxSTT bit must be set to generate a start condition. When a slave device sends an acknowledgment, the UCTxSTT bit is cleared, and the data is received. Upon receiving an 8-bit data set, the UCRxIFG flag is set. Once the data is read from the buffer, the UCRxIFG flag is cleared, and the next data can be received. If only a single 8-bit byte should be received, the controller must set the UCTxSTP bit while the byte is received [SLAU445G, 2016, SLAU367O, 2017].

10.6.5 I²C REGISTERS

I²C associated registers include [SLAU445G, 2016, SLAU367O, 2017]:

- UCBxCTLW0 eUSCI_Bx Control Word 0
- UCBxCTL1 eUSCI_Bx Control 1
- UCBxCTL0 eUSCI_Bx Control 0
- UCBxCTLW1 eUSCI_Bx Control Word 1
- UCBxBRW eUSCI_Bx Bit Rate Control Word
- UCBxBR0 eUSCI_Bx Bit Rate Control 0
- UCBxBR1 eUSCI_Bx Bit Rate Control 1
- UCBxSTATW eUSCI_Bx Status Word
- UCBxSTAT eUSCI_Bx Status
- UCBxBCNT eUSCI_Bx Byte Counter Register
- UCBxTBCNT eUSCI_Bx Byte Counter Threshold Register

- UCBxRXBUF eUSCI_Bx Receive Buffer
- UCBxTXBUF eUSCI_Bx Transmit Buffer
- UCBxI2COA0 eUSCI_Bx I2C Own Address 0
- UCBxI2COA1 eUSCI_Bx I2C Own Address 1
- UCBxI2COA2 eUSCI_Bx I2C Own Address 2
- UCBxI2COA3 eUSCI_Bx I2C Own Address 3
- UCBxADDRX eUSCI_Bx Received Address Register
- UCBxADDMASK eUSCI_Bx Address Mask Register
- UCBxI2CSA eUSCI_Bx I2C Slave Address
- UCBxIE eUSCI_Bx Interrupt Enable
- UCBxIFG eUSCI_Bx Interrupt Flag
- UCBxIV eUSCI_Bx Interrupt Vector

10.6.6 PROGRAMMING THE I^2C

Programming the I^2C in Energia

The Energia IDE supports the following I^2C functions:

- Wire.available(): returns the numbers of bytes available to be read with the Wire.read() function.
- Wire.begin(addr): used by a device to join the I^2C bus. For a slave device, the unique slave address is provided. If no address is provided, the device joins the bus as the master.
- Wire.beginTransmission(addr): used to begin transmission to a slave device.
- Wire.endTransmission(stop): ends transmission to a slave device.
- Wire.read(): reads a byte that was transmitted from a slave device.
- Wire.requestFrom(addr, qty, stop): used by a master device to request data bytes from a slave device.
- Wire.write(val), Wire.write(str), Wire.write(data, length): variety of functions used to write data from a slave designated device.

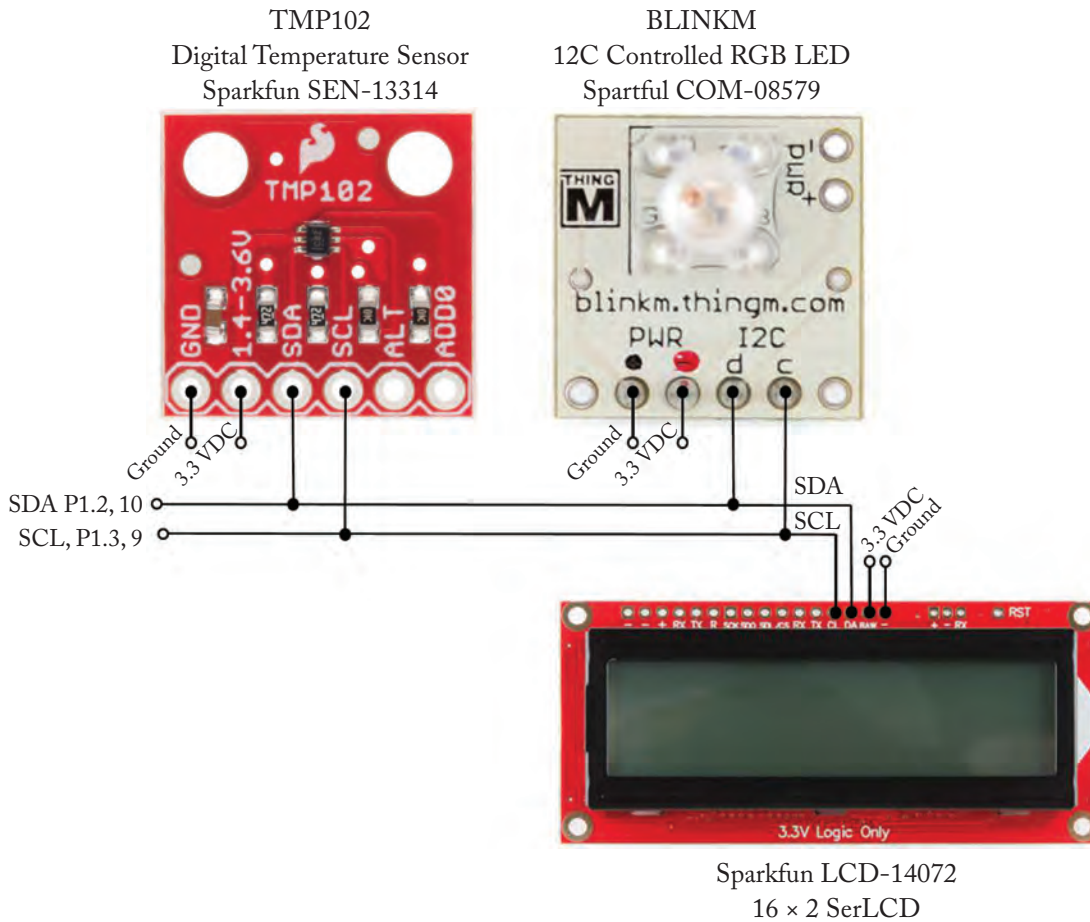


Figure 10.15: MSP430FR2433 configured with peripheral devices via I²C bus. (Figures used courtesy of Texas Instruments (www.ti.com) and Sparkfun Electronics (www.sparkfun.com).)

In the next several examples we illustrate writing and reading data to peripheral devices on an I²C bus. Specifically, we connect a Sparkfun I²C compatible LCD (LCD-14072), the BLINKM I²C controlled RGB LED (Sparkfun COM-08579), and a TMP102 digital temperature sensor (Sparkfun SEN-13314), as shown in Figure 10.15.

I²C LCD. In this example we configure the Sparkfun 16x2 SerLCD (LCD-14072) as an I²C slave device at address 0x72.

```
//*********************************************************************
//I2C_LCD: demonstrates initialization and operation of I2C compatible
// LCD (Sparkfun 16x2 SerLCD (LCD-14072).
//
//I2C code adapted from I2C code provided by Jim Lindblom of
//Sparkfun Electronics (www.sparkfun.com)
//*********************************************************************

#include <Wire.h>

#define lcd_addr 0x72         //OpenLCD address

int cycles = 0;

void setup()
{

Wire.begin();                          //Join I2C bus as master
                                       //bus speed: 100 kHz

Wire.beginTransmission(lcd_addr);      //write mode LCD
Wire.write(0x7c);                      //LCD in setting mode
Wire.write(0x2d);                      //clear LCD display
Wire.endTransmission();
}

void loop()
{
cycles++;                             //cycle count
i2cSendValue(cycles);                 //send the count to LCD -
                                      //integer sent as four values
delay(50);                            //Maximum update rate of OpenLCD
                                      //approximately 100Hz (10ms).
```

```
                                        //Shorter delay causes flicker
}

//*****************************************************************
//void i2cSendValue(int value): the function divides the integer into
//four bytes nteger into four values and sends them out over I2C
//*****************************************************************

void i2cSendValue(int value)
{
Wire.beginTransmission(lcd_addr);     //write to LCD
Wire.write(0x7c);                     //LCD in setting mode
Wire.write(0x2d);                     //clear LCD display
Wire.print("Cycles: ");
Wire.print(value);
Wire.endTransmission();               //Stop I2C transmission
}

//*****************************************************************
```

I²C BLINKM LED. The BLINKM LED is a tri-color LED manufactured by THINGM (www.thingm.com). An I²C compatible breakout board is available from Sparkfun Electronics (COM-08579). The BLINKM I²C address is 0x09. The LED color is set by sending a triplet of color values (R,G,B) to the BLINKM via the I²C bus.

```
//*****************************************************************
//BlinkMFlash---simple demonstration of flashing a BlinkM
//Adapted from BlinkM code provided by THINKM at http://thingm.com
//*****************************************************************

#include "Wire.h"

int blinkm_addr = 9;          //default address of a blinkm

void setup()
{
Wire.begin();
}

void loop()
```

```
{
BlinkM_setRGB(blinkm_addr, 0xff, 0x00, 0x00); //red
delay(500);
BlinkM_setRGB(blinkm_addr, 0x00, 0xff, 0x00); //green
delay(500);
BlinkM_setRGB(blinkm_addr, 0x00, 0x00, 0xff); //blue
delay(500);
}

//*********************************************************************
//void BlinkM_setRGB - sets an RGB color immediately
//*********************************************************************

static void BlinkM_setRGB(byte addr, byte red, byte grn, byte blu)
{
Wire.beginTransmission(addr);
Wire.write('n');
Wire.write(red);
Wire.write(grn);
Wire.write(blu);
Wire.endTransmission();
}

//*********************************************************************
```

TMP102 I²C Temperature Sensor. The TMP102 is a low-power digital temperature sensor compatible with the I²C bus. A breakout board is available from Sparkfun (SEN-13314). The TMP102 has the unique address of 0x4B. Temperature data is stored as two bytes. The two bytes are read from the TMP102 and then assembled as a single value. It is important to note that the least significant nibble of the second byte does not contain temperature data. The TMP102 has a resolution of 0.0625 degrees Centigrade per bit. In this example, temperature data is gathered from the TMP102 converted to Centigrade and Fahrenheit and displayed on the serial monitor [SBOS397B].

```
//*********************************************************************
//TMP102_Example
//Adapted from TMP102_Example provided by Texas Instruments
//*********************************************************************
//TMP102: low power digital temperature sensor with a two-wire
//        serial interface [SOBS397B].
```

```
//*********************************************************************

#include <Wire.h>


void setup()
{
Serial.begin(9600);                  //initialize for serial monitor
Wire.begin();                        //initialize I2C communication
}

void loop()
{
                                     //call the sensorRead function
double temperature = sensorRead();   //to retrieve the temperature
Serial.println(temperature, DEC);    //displap temp serial monitor
delay(500);                          //wait 500 ms
}

//*********************************************************************
//sensorRead: reads two bytes of temperature data from TMP102.
//Converts result to Centigrade and Fahrenheit.
//*********************************************************************

double sensorRead(void)
{
uint8_t temp[2];                     //holds two bytes of data
                                     //read from TMP102
int16_t tempc;                       //holds modified data bytes
double tempf;                        //holds conversion from tempc

Wire.beginTransmission(0x48);        //point to device 0x48
Wire.write(0x00);                    //point to temp register
Wire.endTransmission();              //relinquish control of I2C line
delay(10);                           //delay for conversion time
Wire.requestFrom(0x48, 2);           //request temperature data

if(2 <= Wire.available())            //if two bytes returned
  {
```

```
  temp[0]  = Wire.read();            //read out the data
  temp[1] = Wire.read();
  temp[1] = temp[1] >> 4;            //ignore lower 4 bits of byte 2
  tempc = ((temp[0] << 4) | temp[1]);//combine for 12 bit binary number
  tempc = tempc*0.0625;              //convert to celcius given
                                     //0.0625C resolution
  tempf = tempc * 9/5 + 32;          //convert to fahrenheit
  return tempf;
  }
}

//*********************************************************************
```

Programming the I²C in C

In this example two MSP430FR2433 LaunchPads are connected via the I²C bus. The master configured device reads five bytes from the slave configured device.

Master configured processor:

```
//*********************************************************************
// --COPYRIGHT--,BSD_EX
// Copyright (c) 2015, Texas Instruments Incorporated
// All rights reserved.
//
//                  MSP430 CODE EXAMPLE DISCLAIMER
//
//*********************************************************************
//MSP430FR243x Demo  - eUSCI_B0 I2C Master RX multiple bytes
//from MSP430 Slave
//
//Description: This demo connects two MSP430's via the I2C bus.
//The master reads 5 bytes from the slave. This is the MASTER CODE.
//The data from the slave transmitter begins at 0 and increments
//with each transfer.  The USCI_B0 RX interrupt is used to know
//when new data has been received.
//
//  ACLK = default REFO ~32768Hz, MCLK = SMCLK = BRCLK = DCODIV ~1MHz.
//
//    *****used with "msp430fr243x_euscib0_i2c_11.c"****
//
//                                /|\  /|\
```

```
//                MSP430FR2433      10k  10k     MSP430FR2433
//                    slave         |    |        master
//             -----------------    |    |   -----------------
//            |     P1.2/UCB0SDA|<-|----|->|P1.2/UCB0SDA     |
//            |                 |  |       |                 |
//            |                 |  |       |                 |
//            |     P1.3/UCB0SCL|<-|------>|P1.3/UCB0SCL     |
//            |                 |          |             P1.0|--> LED
//
//Cen Fang, Texas Instruments Inc., June 2013
//Built with IAR Embedded Workbench v6.20 & Code Composer Studio v6.0.1
//******************************************************************

#include <msp430.h>

volatile unsigned char RXData;

int main(void)
{
WDTCTL = WDTPW | WDTHOLD;
                                    //Configure GPIO
P1OUT &= ~BIT0;                     //Clear P1.0 output latch
P1DIR |= BIT0;                      //For LED
P1SEL0 |= BIT2 | BIT3;              //I2C pins

//Disable the GPIO power-on default high-impedance mode to activate
//previously configured port settings
PM5CTL0 &= ~LOCKLPM5;
                                    //Configure USCI_B0 for I2C mode
UCB0CTLW0 |= UCSWRST;               //Software reset enabled
UCB0CTLW0 |=UCMODE_3|UCMST|UCSYNC; //I2C mode, Master mode, sync
UCB0CTLW1 |= UCASTP_2;              //Automatic stop generated
                                    // after UCB0TBCNT is reached
UCB0BRW = 0x0008;                   //baudrate = SMCLK / 8
UCB0TBCNT = 0x0005;                 //number of bytes to be received
UCB0I2CSA = 0x0048;                 //Slave address
UCB0CTL1 &= ~UCSWRST;
UCB0IE |= UCRXIE | UCNACKIE | UCBCNTIE;
```

```
while (1)
  {
  __delay_cycles(2000);
  while (UCB0CTL1 & UCTXSTP);      //Ensure stop condition got sent
  UCB0CTL1 |= UCTXSTT;             //I2C start condition
  __bis_SR_register(LPM0_bits|GIE);//Enter LPM0 w/ interrupt
  }
}

//******************************************************************

#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector = USCI_B0_VECTOR
__interrupt void USCIB0_ISR(void)
#elif defined(__GNUC__)

void __attribute__ ((interrupt(USCI_B0_VECTOR))) USCIB0_ISR (void)
#else
#error Compiler not supported!
#endif
{
  switch(__even_in_range(UCB0IV, USCI_I2C_UCBIT9IFG))
  {
  case USCI_NONE: break;            //Vector 0: No interrupts
  case USCI_I2C_UCALIFG: break;     //Vector 2: ALIFG
  case USCI_I2C_UCNACKIFG:          //Vector 4: NACKIFG
       UCB0CTL1 |= UCTXSTT;         //I2C start condition
       break;
  case USCI_I2C_UCSTTIFG: break;    //Vector 6: STTIFG
  case USCI_I2C_UCSTPIFG: break;     //Vector 8: STPIFG
  case USCI_I2C_UCRXIFG3: break;     //Vector 10: RXIFG3
  case USCI_I2C_UCTXIFG3: break;     //Vector 14: TXIFG3
  case USCI_I2C_UCRXIFG2: break;     //Vector 16: RXIFG2
  case USCI_I2C_UCTXIFG2: break;     //Vector 18: TXIFG2
  case USCI_I2C_UCRXIFG1: break;     //Vector 20: RXIFG1
  case USCI_I2C_UCTXIFG1: break;     //Vector 22: TXIFG1
  case USCI_I2C_UCRXIFG0:            //Vector 24: RXIFG0
       RXData = UCB0RXBUF;           //Get RX data
       __bic_SR_register_on_exit(LPM0_bits); //Exit LPM0
```

```
        break;
  case USCI_I2C_UCTXIFG0: break;      //Vector 26: TXIFG0
  case USCI_I2C_UCBCNTIFG:            //Vector 28: BCNTIFG
        P1OUT ^= BIT0;                //Toggle LED on P1.0
        break;
  case USCI_I2C_UCCLTOIFG: break;     //Vector 30: clock low timeout
  case USCI_I2C_UCBIT9IFG: break;     //Vector 32: 9th bit
  default: break;
  }
}
//******************************************************************************
```

Slave configured processor:

```
//******************************************************************************
// --COPYRIGHT--,BSD_EX
// Copyright (c) 2015, Texas Instruments Incorporated
// All rights reserved.
//
//                  MSP430 CODE EXAMPLE DISCLAIMER
//
//******************************************************************************
//MSP430FR243x Demo - eUSCI_B0 I2C Slave TX multiple bytes to
//MSP430 Master
//
//Description: This demo connects two MSP430's via the I2C bus. The
//master reads from the slave. This is the SLAVE code. The TX data
//begins at 0 and is incremented each time it is sent. A stop condition
//is used as a trigger to initialize the outgoing data. The USCI_B0 TX
//interrupt is used to know when to TX.
//
//  ACLK = default REFO ~32768Hz, MCLK = SMCLK = default DCODIV ~1MHz.
//
//    *****used with "msp430fr243x_euscib0_i2c_10.c"****
//
//                                /|\  /|\
//               MSP430FR2433      10k  10k     MSP430FR2433
//                   slave         |    |        master
//             -----------------   |    |   -----------------
//            |     P1.2/UCB0SDA|<-|----|->|P1.2/UCB0SDA     |
```

```
//            |                 |  |       |                 |
//            |                 |  |       |                 |
//            |     P1.3/UCB0SCL|<-|------>|P1.3/UCB0SCL     |
//            |                 |          |                 |
//
//Cen Fang, Texas Instruments Inc., June 2013
//Built with IAR Embedded Workbench v6.20 & Code Composer Studio v6.0.1
//******************************************************************

#include <msp430.h>

volatile unsigned char TXData;

int main(void)
{
WDTCTL = WDTPW | WDTHOLD;
P1SEL0 |= BIT2 | BIT3;                //Configure GPIO I2C pins

//Disable the GPIO power-on default high-impedance mode to activate
//previously configured port settings
PM5CTL0 &= ~LOCKLPM5;
                                      //Configure USCI_B0 for I2C mode
UCB0CTLW0 = UCSWRST;                  //Software reset enabled
UCB0CTLW0 |= UCMODE_3 | UCSYNC;       //I2C mode, sync mode
UCB0I2COA0 = 0x48 | UCOAEN;           //own address is 0x48 + enable
UCB0CTLW0 &= ~UCSWRST;                //clear reset register
UCB0IE |= UCTXIE0 | UCSTPIE;          //transmit,stop interrupt enable
__bis_SR_register(LPM0_bits | GIE);  //Enter LPM0 w/ interrupts
__no_operation();
}

//******************************************************************

#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector = USCI_B0_VECTOR
__interrupt void USCIB0_ISR(void)
#elif defined(__GNUC__)
void __attribute__ ((interrupt(USCI_B0_VECTOR))) USCIB0_ISR (void)
#else
```

```
#error Compiler not supported!
#endif
{
  switch(__even_in_range(UCB0IV, USCI_I2C_UCBIT9IFG))
  {
  case USCI_NONE: break;              //Vector 0: No interrupts
  case USCI_I2C_UCALIFG: break;       //Vector 2: ALIFG
  case USCI_I2C_UCNACKIFG: break;     //Vector 4: NACKIFG
  case USCI_I2C_UCSTTIFG: break;      //Vector 6: STTIFG
  case USCI_I2C_UCSTPIFG:             //Vector 8: STPIFG
       TXData = 0;
       UCB0IFG &= ~UCSTPIFG;          //Clear stop condition int flag
       break;
  case USCI_I2C_UCRXIFG3: break;      //Vector 10: RXIFG3
  case USCI_I2C_UCTXIFG3: break;      //Vector 14: TXIFG3
  case USCI_I2C_UCRXIFG2: break;      //Vector 16: RXIFG2
  case USCI_I2C_UCTXIFG2: break;      //Vector 18: TXIFG2
  case USCI_I2C_UCRXIFG1: break;      //Vector 20: RXIFG1
  case USCI_I2C_UCTXIFG1: break;      //Vector 22: TXIFG1
  case USCI_I2C_UCRXIFG0: break;      //Vector 24: RXIFG0
  case USCI_I2C_UCTXIFG0:
       UCB0TXBUF = TXData++;
       break;                         //Vector 26: TXIFG0
  case USCI_I2C_UCBCNTIFG: break;     //Vector 28: BCNTIFG
  case USCI_I2C_UCCLTOIFG: break;    //Vector 30: clock low timeout
  case USCI_I2C_UCBIT9IFG: break;    //Vector 32: 9th bit
  default: break;
  }
}

//******************************************************************
```

10.7 LABORATORY EXERCISE: UART AND SPI COMMUNICATIONS

Configure two MSP430 LaunchPads to communicate using the UART and SPI.

10.8 SUMMARY

In this chapter we have discussed the complement of serial communication features aboard the MSP430 microcontroller. The system is equipped with a host of different serial communication subsystems including eUSCI_A type modules and eUSCI_B modules. Each microcontroller in the MSP430 line has a complement of A and B type eUSCI modules.

10.9 REFERENCES AND FURTHER READING

DS3234 Extremely Accurate SPI Bus RTC with Integrated Crystal and SRAM, Maxim Integrated, 2015. www.maximintegrated.com 427, 428

MSP430FR2433 LaunchPad Development Kit (MSP-EXP430FR2433), (SLAU739), Texas Instruments, 2017.

MSP430FR2433 Mixed-Signal Microcontroller, (SLASE59D), Texas Instruments, 2018.

MSP430FR4xx and MSP430FR2xx Family User's Guide, (SLAU445G), Texas Instruments, 2016. 396, 397, 400, 401, 403, 404, 405, 411, 413, 414, 416, 441, 443, 444, 445

MSP430FR58xx, MSP430FR59xx, and MSP430FR6xx Family User's Guide, (SLAU367O), Texas Instruments, 2017. 396, 397, 400, 401, 403, 404, 405, 411, 413, 414, 416, 441, 443, 444, 445

MSP430FR5994 LaunchPad Development Kit (MSP-EXP430FR5994), (SLAU678A), Texas Instruments, 2016.

MSP430FR599x, MSP430FR596x Mixed-Signal Microcontrollers, (SLASE54C), Texas Instruments, 2018.

TPIC6C596 Power Logic 8-Bit Shift Register, (SLIS093D), Texas Instruments, 2015. 420

Unicode Consortium. www.unicode.org

10.10 CHAPTER PROBLEMS

Fundamental

1. Describe the difference between parallel and serial communications.
2. If the communication cost is the primary issue, which communication methods (parallel, series) should be used? Why?
3. What is the difference between synchronous and asynchronous communications?

4. The eUSCI in the UART mode supports LIN and IrDA. For each, identify the protocol used: serial/parallel and synchronous/asynchronous.
5. In the I^2C communication protocol, how does one configure the MSP430 to become a master device? What must be done to configure it as a slave device?
6. Give a brief description of a communication protocol.

Advanced

1. Write a function that properly initialize the SPI unit. Specify the configuration parameter values used for the external device.
2. Describe interrupts associated with the I^2C unit.
3. There are multiple I^2C interrupts but a single interrupt vector. After detecting an interrupt, the I^2C interrupt system must identify the source of the interrupt. How is this accomplished?

Challenging

1. Design and program three MSP430 controller systems to measure temperatures surrounding the three controllers. Create a wireless communication network using the three controllers along with the CC2530-ZNP radio transceivers. The controllers should constantly share the temperature sensor data among the members. Select a central controller and display the three temperature values on a LCD display once every 5 s.
2. Design and program an I^2C based system to measure and display temperature. The system should also contain a tri-color LED to display a corresponding color based on measured temperature.
3. Design and program an I^2C based system to provide for a peripheral EEPROM (Sparkfun COM-00525).
4. Design and program an I^2C based system to provide for a peripheral DAC based on the MPC47725 (Sparkfun BOB-12918).

CHAPTER 11

MSP430 System Integrity

Objectives: After reading this chapter, the reader should be able to:

- describe the concept of electromagnetic interference (EMI);
- describe the possible sources of EMI noise in a microcontroller system;
- differentiate between conducted and radiated EMI;
- list different sources of EMI;
- list design techniques to minimize EMI;
- describe how a cyclic redundancy check (CRC) may be used to insure the integrity of data;
- describe the features of the CRC32 system onboard the MSP430FR5994;
- sketch a linear feedback shift register for a given generator polynomial;
- list common generator polynomials used within CRC systems;
- program the MSP430FR5994 CRC32 system to generate a data checksum;
- describe how the MSP430FR5994 advanced encryption standard module, the AES256, may be used to provide for data transmission integrity;
- describe the steps used to encrypt/decrypt data using the AES256 standard;
- sketch a block diagram of the MSP430FR5994 AES256 module; and
- program the MSP430 AES256 module to encrypt and decrypt data.

11.1 OVERVIEW

This chapter may be the most important chapter in the book. It contains essential information about how to maintain the integrity of a microcontroller-based system.[1] The chapter begins with a discussion on EMI, also known as noise. Design practices to minimize EMI are then discussed. The second section of the chapter discusses the concept of the CRC. This is a hardware-based

[1]This chapter was adapted with permission from *Embedded Systems Design with the Texas Instruments MSP432 32-bit Processor*, Morgan & Claypool Publishers, 2017.

subsystem used to generate a checksum of a block of data. The checksum may be used to test the integrity of data once it has been transmitted or loaded to a new location. The final section covers the MSP430FR5994 advanced encryption standard module, the AES256. This module is used to insure the integrity of data transmission using a key-based encryption and decryption technique.

11.2 ELECTROMAGNETIC INTERFERENCE

EMI, commonly referred to as noise, may come from a number of sources as shown in Figure 11.1. Noise causes program malfunction and data corruption, making it impossible to complete the intended task of the controller. It is important to understand the sources of noise and coupling mechanisms to a microcontroller-based project, so proper preventive techniques may be employed during the design process. As shown in the figure, noise may be coupled to a victim receptor system via radiated or conducted mechanisms. Radiated sources include radio frequency sources such as radio stations and cell phones. Naturally occurring lightning is also a source of noise. A nearby lightning strike generates a tremendous amount of noise at a variety of frequencies. Noise may also be generated by motors and motor based appliances such as drills, mixers, and blenders. Often microcontrollers are used to control a motor. The motor, although part of the designed system, may be a source of noise for the microcontroller controlling its operation. Electrostatic discharge (ESD), e.g., static electricity, may inject noise or damage a microcontroller-based system. Conducted sources of noise in a microcontroller-based system include other system components or the power supply serving the system. It is interesting to note the microcontroller itself may also serve as a noise source for other system components or nearby systems [AN1705, 2004, COP888, 1996].

11.2.1 EMI REDUCTION STRATEGIES

There are several strategies to minimize EMI interference. These include [AN1705, 2004, COP888, 1996]:

- implementing EMI suppression techniques early in the design process. It is very challenging to provide EMI suppression after a system has been implemented; and
- implementing noise suppression techniques at the source of the noise, disrupting the source to receptor transmission path, protecting the receptor system from noise, and a combination of all three techniques.

Provided below are specific techniques to suppress EMI noise in a microcontroller-based system [Barrett and Pack, 2004, AN1705, 2004, COP888, 1996].

- If possible, incoming signal lines to a microcontroller-based system should be twisted. This will minimize the chance of parallel conductors inducing noise in an adjacent conductor via crosstalk. If signals are being transmitted by a multiple conductor ribbon cable, consider

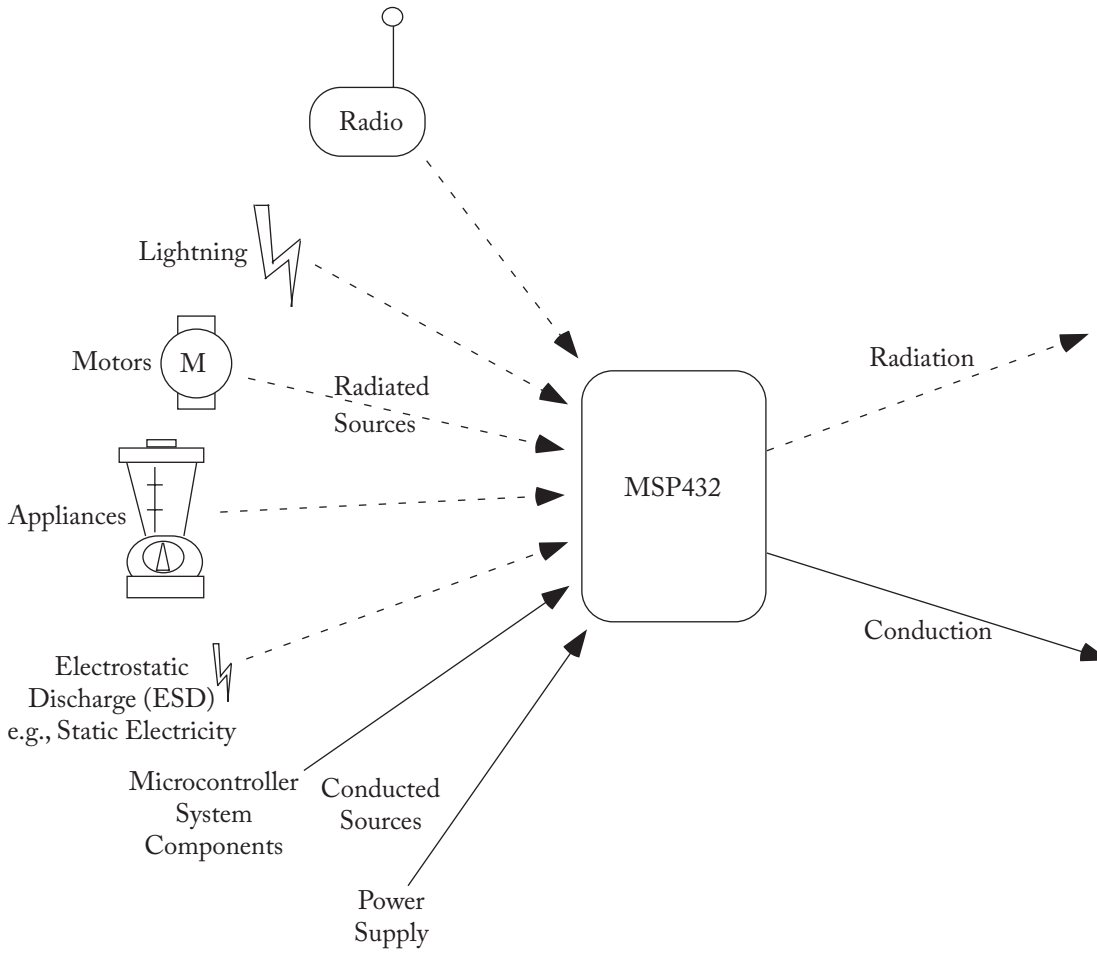


Figure 11.1: Noise sources in a microcontroller-based system. (Adapted from AN1705 [2004], COP888 [1996].)

gently twisting the cable and also providing a ground conductor alternating with signal-carrying conductors.

- Use shielded cable for signal conductors coming into the microcontroller-based system.

- If the microcontroller is being used to control a motor, use an opto-isolator between the microcontroller and the motor interface circuit. Also, the motor and the microcontroller should not share a common power supply.

- Use filters for signals coming into a microcontroller-based system. Filters are commonly available in the form of ferrite beads.
- Filter the power supply lines to the circuit. Typically, a 10–470 μF capacitor is employed for this purpose. Also, a 0.1 μF capacitor should be used between the power and ground pins on each integrated circuit.
- Ground the metal crystal time base case to insure it does not radiate a noise signal.
- Mount the microcontroller-based project in a metal chassis.
- There are several defensive programming techniques to help combat noise. One easy to implement technique is to declare unused microcontroller pins as output.

11.3 CYCLIC REDUNDANCY CHECK

In a previous professional life, one of the authors (sfb) served as a missileer in the United States Air Force. On a routine basis, the guidance set aboard an assigned missile was updated with critical data to insure the missile would serve its intended mission. Maintenance crews from a nearby support base would transport information tapes out to the missile site where the onboard missile guidance set was updated. A CRC checksum was generated on the information tapes before they left the support base. After the information was loaded from the tapes to the missile guidance set, a CRC checksum was performed. If the checksum generated by the missile guidance set matched the checksum generated at the support base, the missile was designated as properly updated.

This scenario illustrates the application and importance of using a CRC to maintain data integrity. This technique is often performed to ensure the integrity of transmitted or stored data.

The basic concept behind generating a CRC checksum is binary division. The basic operation of division can be defined as [AN370]:

$$Dividend/divisor = quotient + remainder.$$

The block of data to be protected via the checksum is considered the dividend. The dividend is divided by a pre-selected CRC polynomial which serves as the divisor. At the completion of the division operation, a quotient and a remainder result. The remainder of the operation serves as the CRC checksum.

Generation of a checksum is based on the concept that when a given block of data is divided by a specific polynomial with the division hardware initialized with the same value (seed), the same checksum will result every time the operation is performed. Similarly, if the input data is different or in a different order, the polynomial is changed, or the division hardware is seeded with a different initial value, a different checksum will result. A number of common

polynomials have been developed to support CRC checksum generation. Two common ones include [SLAU367O, 2017]:

- CRC16-CCITT defined as $f(x) = X^{15} + X^{12} + X^5 + 1$
- CRC32-IS3309 defined as

$$f(x) = X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^{10} + X^8 + X^7 + X^5 + X^4 + X^2 + X + 1. \quad (11.1)$$

A linear feedback shift register (LFSR) is used to generate the checksum. The polynomial divisor chosen to generate the checksum specifies the hardware connection for the LFSR. For example, the LFSR configuration for the CRC16-CCITT polynomial is shown in Figure 11.2. Note how the polynomial terms specify the output connections of certain flip-flops within the LFSR. To generate the checksum, the LFSR is initially configured to the seed value. The data block is fed in as a serial data stream. The resulting remainder is used as the checksum and appended to the original data block for transmission [SLAU367O, 2017].

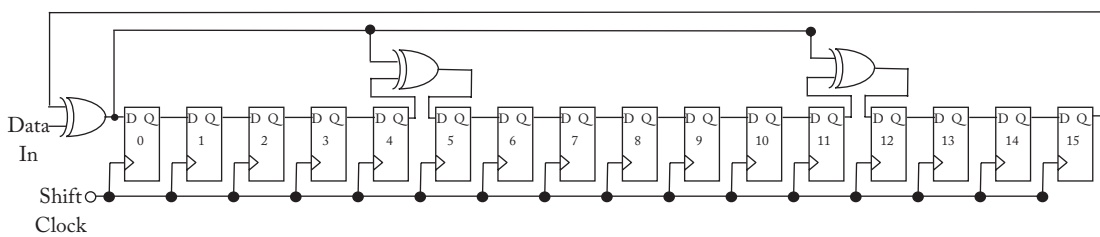
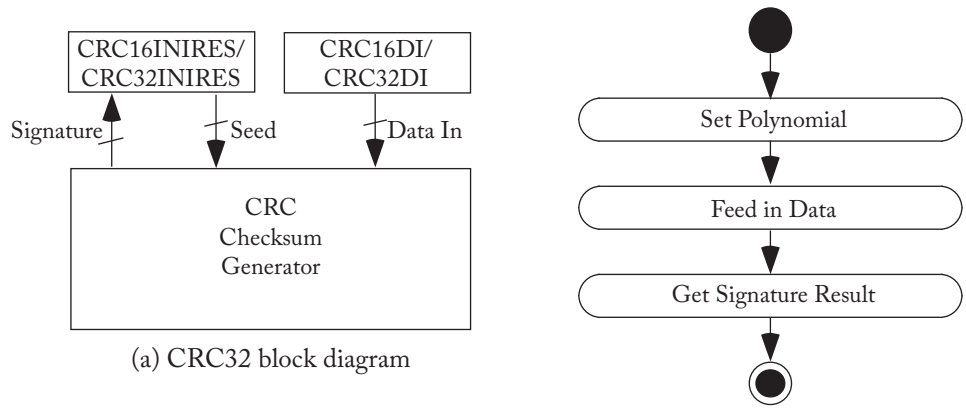


Figure 11.2: CRC16-CCITT polynomial and LFSR configuration [SLAU367O, 2017].

11.3.1 MSP430FR5994 CRC32 MODULE

A block diagram for the MSP430FR5994 CRC32 module is provided in Figure 11.3a. The MSP430FR5994 CRC32 module is quite flexible. It allows for 16- or 32-bit CRC generation. It also provides for data bit 0 being the MSB or LSB. This allows for compatibility with both modern and legacy hardware. Also, to speed up the calculation of the CRC checksum, the linear feedback shift register operation is implemented with an equivalent XOR gate combinational logic tree [SLAU367O, 2017].

The UML activity diagram for the CRC operation is provided in Figure 11.3b. The operation is quite straight forward. The CRC polynomial is provided to the CRC32 system along with the seed via the CRC16INIRES (CRC32INIRES) register. The data block to perform the checksum operation is fed into the CRC16DI (CRC32DI) register. The CRC checksum operation is performed and the checksum is available at the CRC16INIRES (CRC32INIRES) register [SLAU367O, 2017].



(a) CRC32 block diagram

(b) CRC32 UML diagram

Figure 11.3: MSP430 CRC32 module.

11.3.2 CRC16 REGISTERS

The CRC16 system is supported by a set of the following registers [SLAU367O, 2017]:

- CRC16DI CRC16 Data Input Register
- CRCDIRB CRC16 Data In Reverse Register
- CRCINIRES CRC16 Initialization and Result Register
- CRCRESR CRC16 Result Reverse Register

Details of specific register and bit settings are contained in *MSP430P4xx Family Technical Reference Manual* [SLAU367O, 2017].

11.3.3 CRC32 REGISTERS

The CRC32 system is supported by a set of the following registers [SLAU367O, 2017]:

- CRC32DIW0 CRC32 Data Input Register
- CRC32DIRBW0 CRC32 Data In Reverse Register
- CRC32INIRESW0 CRC32 Initialization and Result Register 0
- CRC32INIRESW1 CRC32 Initialization and Result Register 1
- CRC32RESRW0 CRC32 Result Reverse Register 0
- CRC32RESRW1 CRC32 Result Reverse Register 1

- CRC16DIW0 CRC16 Data Input Register
- CRC16DIRBW0 CRC16 Data In Reverse Register
- CRC16INIRESW0 CRC16 Initialization and Result Register
- CRC16RESRW0 CRC16 Result Reverse Register

Details of specific register and bits settings are contained in *MSP430P4xx Family Technical Reference Manual* [SLAU367O, 2017].

Example: In this example the hardware and software techniques of generating a CRC16 checksum is compared.

```
//*****************************************************************************
// --COPYRIGHT--,BSD_EX
// Copyright (c) 2015, Texas Instruments Incorporated
// All rights reserved.
//
//                  MSP430 CODE EXAMPLE DISCLAIMER
//
//*****************************************************************************
//MSP430FR5x9x Demo - CRC16, Compare CRC output with software-based
//algorithm
//
//Description: An array of 16 random 16-bit values are moved through
//the CRC module, as well as a software-based CRC-CCIT-BR algorithm.
//The software-based algorithm handles 8-bit inputs only, the 16-bit
//words are broken into 2 8-bit words before being run through
//(lower byte first). The outputs of both methods are then compared to
//ensure that the operation of the CRC module is consistent with the
//expected outcome. If the values of each output are equal, set P1.0,
//else reset.
//
//   MCLK = SMCLK = default DCO~1MHz
//
//                MSP430FR5994
//             -----------------
//         /|\|                 |
//          | |                 |
//          --|RST              |
//            |                 |
```

```
//            |             P1.0|--> LED
//
//William Goh, Texas Instruments Inc., October 2015
//Built with IAR Embedded Workbench V6.30 & Code Composer Studio V6.1
//******************************************************************

#include <msp430.h>

const unsigned int CRC_Init = 0xFFFF;

const unsigned int CRC_Input[] =
{
0x0fc0, 0x1096, 0x5042, 0x0010,         //16 random 16-bit numbers
0x7ff7, 0xf86a, 0xb58e, 0x7651,         //these numbers can be
0x8b88, 0x0679, 0x0123, 0x9599,         //modified if desired
0xc58c, 0xd1e2, 0xe144, 0xb691
};

unsigned int CRC_Result;                //Holds results obtained through
                                        //the CRC16 module
unsigned int SW_Results;                //Holds results obtained through
                                        //SW

//Software Algorithm Function Declaration
unsigned int CCITT_Update(unsigned int, unsigned int);

int main(void)
{
unsigned int i;

WDTCTL = WDTPW | WDTHOLD;               //Stop WDT
                                        //Configure GPIO
P1OUT &= ~BIT0;                         //Clear LED to start
P1DIR |= BIT0;                          // P1.0 Output

//Disable the GPIO power-on default high-impedance mode to activate
//previously configured port settings
PM5CTL0 &= ~LOCKLPM5;
```

```
//First, use the CRC16 hardware module to calculate the CRC...
CRC16INIRESW0 = CRC_Init;                 //Init CRC16 HW module

for(i = 0; i < (sizeof(CRC_Input) >> 1); i++)
  {
                                          //Input random values into CRC
                                          //Hardware
  CRC16DIRBW0 = CRC_Input[i];             //Input data in CRC
  __no_operation();
  }

CRC_Result = CRC16INIRESW0;               //Save results (per CRC-CCITT
                                          //standard)

//Now use a software routine to calculate the CRC...
SW_Results = CRC_Init;                    //Init SW CRC
for(i = 0; i < (sizeof(CRC_Input) >> 1); i++)
  {
  //First input upper byte
  SW_Results = CCITT_Update(SW_Results, (CRC_Input[i] >> 8) & 0xFF);

  //Then input lower byte
  SW_Results = CCITT_Update(SW_Results, CRC_Input[i] & 0xFF);
  }

//Compare data output results
if(CRC_Result == SW_Results)              //if data is equal
  P1OUT |= BIT0;                          //set the LED
else
  P1OUT &= ~BIT0;                         //if not, clear LED

while(1);                                 //infinite loop
}

//*******************************************************************
// Software algorithm - CCITT CRC16 code
//*******************************************************************

unsigned int CCITT_Update(unsigned int init, unsigned int input)
```

```
{
unsigned int CCITT = (unsigned char) (init >> 8) | (init << 8);
CCITT ^= input;
CCITT ^= (unsigned char) (CCITT & 0xFF) >> 4;
CCITT ^= (CCITT << 8) << 4;
CCITT ^= ((CCITT & 0xFF) << 4) << 1;

return CCITT;
}

//*******************************************************************
```

Example: In this example the hardware and software techniques of generating a CRC32 checksum is compared.

```
//*******************************************************************
// --COPYRIGHT--,BSD_EX
// Copyright (c) 2016, Texas Instruments Incorporated
// All rights reserved.
//
//                  MSP430 CODE EXAMPLE DISCLAIMER
//
//*******************************************************************

//*******************************************************************
//MSP430FR5x9x Demo - CRC32, Compare CRC32 output with software-based
//                    algorithm
//
//Description: An array of 16 random 16-bit values are moved through
//the CRC32 module, as well as a software-based CRC32-ISO3309
//algorithm. The software-based algorithm handles 8-bit inputs only,
//the 16-bit words are broken into 2 8-bit words before being run
//through (lower byte first). The outputs of both methods are then
//compared to ensure that the operation of the CRC module is
//consistent with the expected outcome. If the values of each output
//are equal, set P1.0, else reset.
//
//   MCLK = SMCLK = default DCO~1MHz
//
//                MSP430FR5994
//             -----------------
```

```
//         /|\|                 |
//          | |                 |
//          --|RST              |
//            |                 |
//            |             P1.0|--> LED
//
//William Goh, Texas Instruments Inc., April 2016
//Built with IAR Embedded Workbench V6.40 & Code Composer Studio V6.1
//******************************************************************

#include <msp430.h>

#define  POLYNOMIAL_32    0xEDB88320

//Holds a CRC32 lookup table
unsigned long crc32Table[256];

//Global flag indicating that the CRC32 lookup table has been initialized
unsigned int crc32TableInit = 0;

const unsigned long CRC_Init = 0xFFFFFFFF;

const unsigned short CRC_Input[] =
{
0xc00f, 0x9610, 0x5042, 0x0010,         // 16 random 16-bit numbers
0x7ff7, 0xf86a, 0xb58e, 0x7651,         // these numbers can be
0x8b88, 0x0679, 0x0123, 0x9599,         // modified if desired
0xc58c, 0xd1e2, 0xe144, 0xb691
};

//Holds results obtained through the CRC32 hardware module
unsigned long CRC32_Result;

//Holds results obtained through software algorithm
unsigned long SW_CRC32_Results;

// Software CRC32 algorithm function declaration
void initSwCrc32Table(void);
unsigned long updateSwCrc32(unsigned long crc, char c );
```

```
int main(void)
{
unsigned int i;

WDTCTL = WDTPW | WDTHOLD;               //Stop WDT
                                        //Configure GPIO
P1OUT &= ~BIT0;                         //Clear LED to start
P1DIR |= BIT0;                          //P1.0 Output

//Disable the GPIO power-on default high-impedance mode to activate
//previously configured port settings
PM5CTL0 &= ~LOCKLPM5;

//First, use the CRC32 hardware module to calculate the CRC...
CRC32INIRESW0 = CRC_Init;               //Init CRC32 HW module
CRC32INIRESW1 = CRC_Init;               //Init CRC32 HW module

for(i = 0; i < (sizeof(CRC_Input) >> 1); i=i+2)
  {
  //Input values into CRC32 Hardware
  CRC32DIW0 = (unsigned int) CRC_Input[i + 0];
  CRC32DIW1 = (unsigned int) CRC_Input[i + 1];
  }

//Save the CRC32 result
CRC32_Result = ((unsigned long) CRC32RESRW0 << 16);
CRC32_Result = ((unsigned long) CRC32RESRW1&0x0000FFFF)|CRC32_Result;

//Now use a software routine to calculate the CRC32...
//Init SW CRC32
SW_CRC32_Results = CRC_Init;

for(i = 0; i < (sizeof(CRC_Input) >> 1); i++)
  {
  //Calculate the CRC32 on the low-byte first
  SW_CRC32_Results=updateSwCrc32(SW_CRC32_Results,(CRC_Input[i]&0xFF));

  //Calculate the CRC on the high-byte
```

```
  SW_CRC32_Results=updateSwCrc32(SW_CRC32_Results,(CRC_Input[i] >> 8));
  }

//Compare data output results
if(CRC32_Result == SW_CRC32_Results)   //if data is equal
  P1OUT |= BIT0;                       //set the LED
else
  P1OUT &= ~BIT0;                      //if not, clear LED

while(1);                              //infinite loop
}

//*********************************************************************
// Calculate the SW CRC32 byte-by-byte
//*********************************************************************

unsigned long updateSwCrc32( unsigned long crc, char c )
{
unsigned long tmp, long_c;

long_c = 0x000000ffL & (unsigned long) c;

if(!crc32TableInit)
  {
  initSwCrc32Table();
  }

tmp = crc ^ long_c;
crc = (crc >> 8) ^ crc32Table[ tmp & 0xff ];

return crc;
}

//*********************************************************************
// Initializes the CRC32 table
//*********************************************************************

void initSwCrc32Table(void)
{
```

```
int i, j;
unsigned long crc;

for(i = 0; i < 256; i++)
  {
  crc = (unsigned long) i;

  for(j = 0; j < 8; j++)
    {
    if(crc & 0x00000001L)
      {
      crc = ( crc >> 1 ) ^ POLYNOMIAL_32;
      }
    else
      {
      crc =   crc >> 1;
      }
    }
    crc32Table[i] = crc;
  }
//Set flag that the CRC32 table is initialized
crc32TableInit = 1;

}
//*****************************************************************
```

11.4 AES256 ACCELERATOR MODULE

The MSP430FR5994 is equipped with the AES256 Accelerator Module that allows encryption and decryption of data using the Rijndael cryptographic algorithm. The algorithm allows the encryption of a 128-bit plain text data block into a corresponding size cipher text block. The data may then be transmitted in an encrypted format and decrypted using a similar algorithm at the receiving end [FIPS-197, 2001, SLAU367O, 2017].

The data algorithm uses a 128-, 192-, or 256-bit cipher key to encrypt the plain text data block. The length of the cipher key determines the number of rounds (10, 12, or 14, respectively) of encryption performed on the plain text data to transform it into the cipher text block. The basic encryption process is shown in Figure 11.4a,b. The plain text 128-bit block is formatted into a state block. The state block then goes through a series of transformation rounds including an initial round, the sub-byte round, the shift rows round, the mix columns round, the add key round, and the final round to encrypt the data. As shown in Figure 11.4b, a specific round key is

derived from the original cipher key and used in a given round [FIPS-197, 2001, SLAU367O, 2017].

A block diagram of the MSP430FR5994 AES256 Accelerator Module is provided in Figure 11.4c. Input plain text data for encryption may be stored in register AESADIN or AESAXDIN. Data input to AESAXDIN is XORed with the current state value. The operation of the AES256 Accelerator module is controlled by AES Control Registers 0 and 1. The AES key is provided to the AESAKEY register. The encrypted cipher text is output to the AESADOUT Register [SLAU367O, 2017].

11.4.1 REGISTERS

The AES256 system is supported by a complement of registers including [SLAU367O, 2017]:

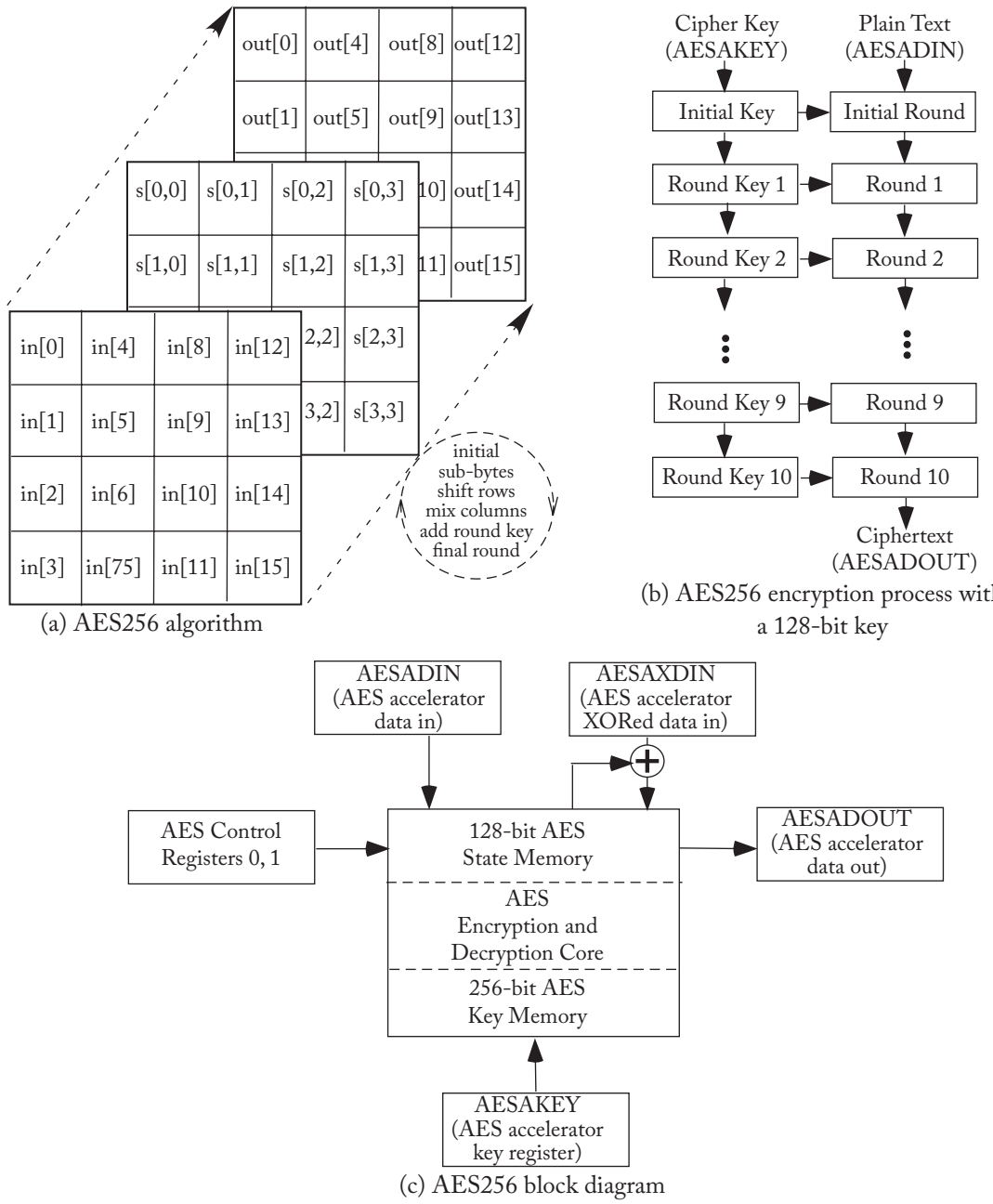
- AESACTL0 AES accelerator control register 0
- AESACTL1 AES accelerator control register 1
- AESASTAT AES accelerator status register
- AESAKEY AES accelerator key register
- AESADIN AES accelerator data in register
- AESADOUT AES accelerator data out register
- AESAXDIN AES accelerator XORed data in register
- AESAXIN AES accelerator XORed data in register (no trigger)

Details of specific register and bits settings are contained in *MSP430FR58xx, MSP430FR59xx, and MSP430FR6xx Family User's Guide* [SLAU367O, 2017] and will not be repeated here.

11.4.2 API SUPPORT

Texas Instruments provides extensive support for many MSP430 subsystems through a series of application program interfaces (APIs). Basically, they are a library of prewritten functions that allow for the rapid prototyping of programs. Provided below is a list of APIs supporting the AES256 Accelerator Modules. Details on API settings are provided in *MSP430 Peripheral Driver Library User's Guide* [Driver Library, 2015] and will not be repeated here.

- AES256_clearErrorFlag
- AES256_clearInterrupt
- AES256_decryptData



(a) AES256 algorithm

(b) AES256 encryption process with a 128-bit key

(c) AES256 block diagram

Figure 11.4: AES256 encryption process: (a) AES256 algorithm, (b) AES256 encryption process with 128-bit key, and (c) MSP430FR5994 AES256 block diagram [SLAU367O, 2017].

- AES256_disableInterrupt
- AES256_enableInterrupt
- AES256_encryptData
- AES256_getDataOut
- AES256_getErrorFlagStatus
- AES256_getInterruptStatus
- AES256_isBusy
- AES256_reset
- AES256_setCipherKey
- AES256_setDecipherKey
- AES256_startDecryptData
- AES256_startEncryptData
- AES256_startSetDecipherKey

Example: In this example the AES256 module is used to encrypt/decrypt data using a cipher key and APIs from DriverLib.

```
//*****************************************************************************
// --COPYRIGHT--,BSD_EX
// Copyright (c) 2015, Texas Instruments Incorporated
// All rights reserved.
//
//                  MSP430 CODE EXAMPLE DISCLAIMER
//
//*****************************************************************************
//*****************************************************************************
//Below is a simple code example of how to encrypt/decrypt data using a
//cipher key with the AES256 module
//*****************************************************************************

int main(void)
{
//Load a cipher key to module
```

```
MAP_AES256_setCipherKey(AES256_MODULE, CipherKey,
                        AES256_KEYLENGTH_256BIT);

//Encrypt data with preloaded cipher key
MAP_AES256_encryptData(AES256_MODULE, Data, DataAESencrypted);

//Load a decipher key to module
MAP_AES256_setDecipherKey(AES256_MODULE, CipherKey,
                          AES256_KEYLENGTH_256BIT);

//Decrypt data with keys that were generated during encryption takes
//214 MCLK cyles. This function will generate all round keys needed for
//
decryption first and then the encryption process starts
MAP_AES256_decryptData(AES256_MODULE, DataAESencrypted,
                       DataAESdecrypted);
}

//****************************************************************************
```

Example: In this example the AES256 module is used for encryption and decryption. The original plain text data is encrypted and then decrypted. The results are compared and if they agree an LED on P1.0 is illuminated. A UML activity diagram for the example is provided in Figure 11.5.

```
//****************************************************************************
//MSP430P401 Demo - AES256 Encryption & Decryption
//
//Description: This example shows a simple example of encryption and
//decryption using the AES256 module.
//
//                MSP430FR5994
//             -----------------
//         /|\|                 |
//          | |                 |
//          --|RST              |
//            |                 |
//            |             P1.0|-->LED
//
//Key: 000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
```

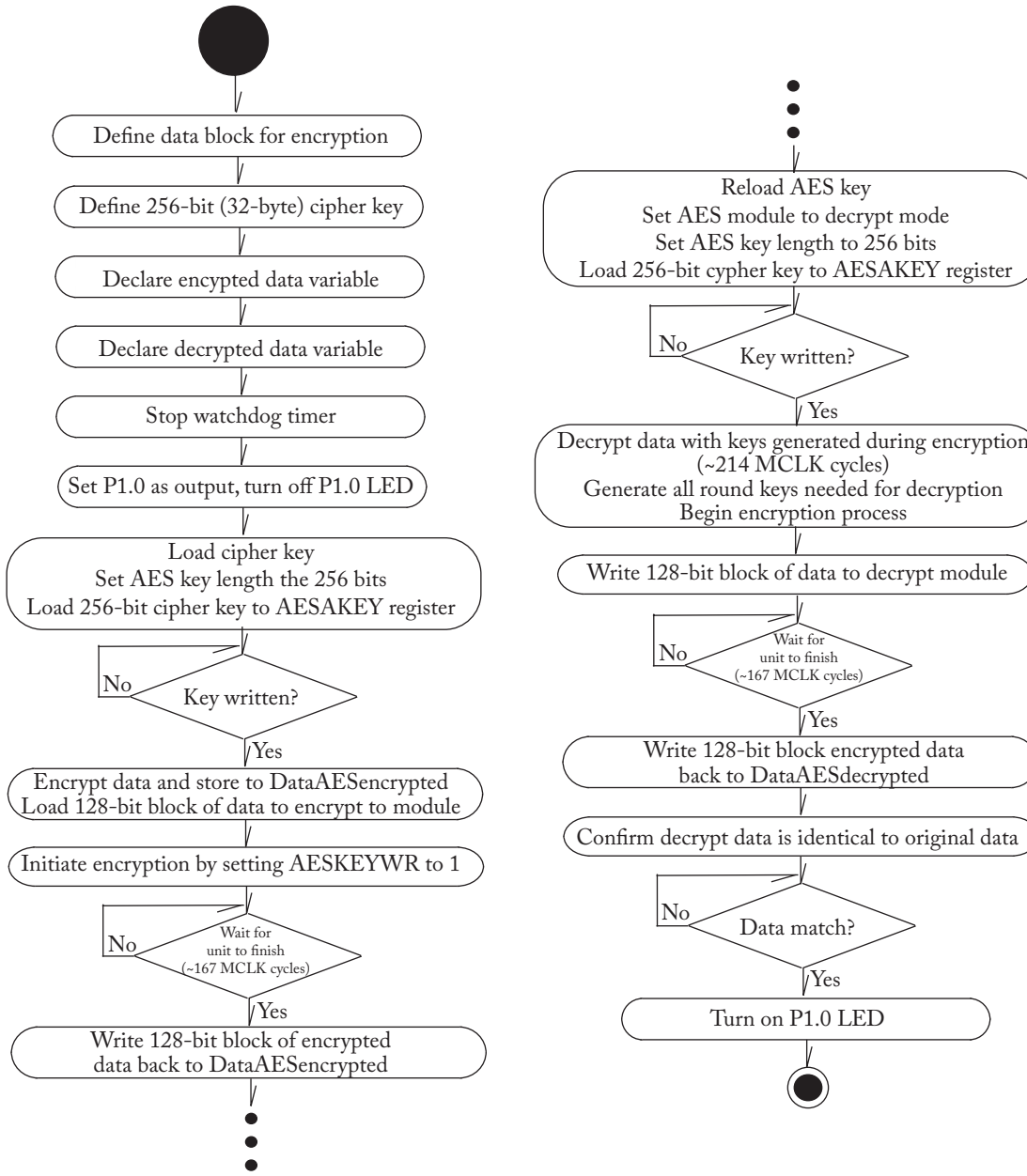


Figure 11.5: AES256 encryption/decryption process [SLAU367O, 2017].

```
//Plaintext: 00112233445566778899aabbccddeeff
//Ciphertext: 8ea2b7ca516745bfeafc49904b496089
//
//Dung Dang
//Texas Instruments Inc.
//Nov 2013
//Built with Code Composer Studio V6.0
//******************************************************************************

#include "msp.h"
#include <stdint.h>

uint8_t Data[16] =
{0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
 0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff };

uint8_t CipherKey[32] =
{0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
 0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f };

uint8_t DataAESencrypted[16];       //Encrypted data
uint8_t DataAESdecrypted[16];       //Decrypted data


int main(void)
{
volatile uint32_t i;
uint16_t sCipherKey, tempVariable;

WDTCTL = WDTPW | WDTHOLD;           //Stop WDT

P1DIR |= BIT0;                      //P1.0 set as output
P1OUT &= ~BIT0;                     //Turn off P1.0 LED

//Step 1: Load cipher key
AESACTL0 &= ~AESOP_3;               //Set AES module to encrypt mode
```

```
//Set AES key length to 256 bits
AESACTL0 = AESACTL0 & (~(AESKL_1 + AESKL_2))  | AESKL__256BIT;

//Load 256-bit cipher key to the AESAKEY register
for(i = 0; i < 256/8; i = i + 2)
  {
  //Concatenate 2 8-bit blocks into one 16-bit block
  sCipherKey =(uint16_t)(CipherKey[i]);
  sCipherKey = sCipherKey |((uint16_t)(CipherKey[i + 1]) << 8);

  //Load 16-bit key block to AESAKEY register
  AESAKEY = sCipherKey;
   }

 //Wait until key is written
 while((AESASTAT & AESKEYWR ) == 0);


 //Step 2: Encrypt data and store to DataAESencrypted
 //Load 128-bit block of data to encrypt to module
 for(i = 0; i < 16; i = i + 2)
   {
   //Concatenate 2 8-bit blocks into one 16-bit block
   tempVariable =(uint16_t)(Data[i]);
   tempVariable = tempVariable |((uint16_t)(Data[i + 1]) << 8);

   //Load 16-bit key block to AESADIn register
   AESADIN = tempVariable;
   }

 //Initiate encryption by setting AESKEYWR to 1
 AESASTAT |= AESKEYWR;

 //Wait unit finished ~167 MCLK
 while( AESASTAT & AESBUSY );

 //Write 128-bit block of encrypted data back to DataAESencrypted
 for(i = 0; i < 16; i = i + 2)
   {
```

```
 tempVariable = AESADOUT;
 DataAESencrypted[i] = (uint8_t)tempVariable;
 DataAESencrypted[i+1] = (uint8_t)(tempVariable >> 8);
 }

//Step 3: Reload AES key
//Set AES module to decrypt mode
AESACTL0 |= AESOP_1;

//Set AES key length to 256 bits
AESACTL0 = AESACTL0 & (~(AESKL_1 + AESKL_2))  | AESKL__256BIT;

//Load 256-bit cipher key to the AESAKEY register
for(i = 0; i < 256/8; i = i + 2)
  {
  //Concatenate 2 8-bit blocks into one 16-bit block
  sCipherKey =(uint16_t)(CipherKey[i]);
  sCipherKey = sCipherKey |((uint16_t)(CipherKey[i + 1]) << 8);

  //Load 16-bit key block to AESAKEY register
  AESAKEY = sCipherKey;
  }

//Wait until key is written
while((AESASTAT & AESKEYWR ) == 0);

//Step 4: Decrypt data with keys that were generated during
//encryption takes 214 MCLK.  This function will generate all round
//keys needed for decryption first and then the encryption process
//starts.

//Write 128-bit block of data to decrypt to module
for(i = 0; i < 16; i = i + 2)
  {
  tempVariable = (uint16_t) (DataAESencrypted[i + 1] << 8);
  tempVariable = tempVariable | ((uint16_t) (DataAESencrypted[i]));
  AESADIN = tempVariable;
  }
```

```
    //Wait until finished ~167 MCLK
    while(AESASTAT & AESBUSY);

    //Write 128-bit block of encrypted data back to DataAESdecrypted
    for(i = 0; i < 16; i = i + 2)
      {
      tempVariable = AESADOUT;
      DataAESdecrypted[i] = (uint8_t)tempVariable;
      DataAESdecrypted[i+1] =(uint8_t)(tempVariable >> 8);
      }

    //Step 4: Confirm decrypted data is identical to original data
    for(i = 0; i < 16; i ++)
       if(DataAESdecrypted[i]!= Data[i])
           while(1);                  //Set breakpoint here for error

    P1DIR |= BIT0;
    P1OUT |= BIT0;                    //Turn on P1.0 LED = success
    while(1);

}

//*********************************************************************
```

11.5 LABORATORY EXERCISE: AES256

Develop an algorithm to encode a plain text block of data with the AES256 system, transmit the cipher text to another microcontroller, and then decrypt back to plain text at the receiving microcontroller.

11.6 SUMMARY

This chapter contained essential information about how to maintain the integrity of a microcontroller-based system. The chapter began with a discussion on EMI, also known as noise. Design practices to minimize EMI were then discussed. The second section of the chapter discussed the concept of the CRC. The final section covered the MSP430 advanced encryption standard module, the AES256.

11.7 REFERENCES AND FURTHER READING

Barrett, S. F. and Pack, D. J. *Embedded Systems: Design and Applications with the 68HC12 and HCS12*, Prentice Hall, Upper Saddle River, NJ, 2004. 462

Federal Information Processing Standards Publication 197 (FIPS-197), November 26, 2001. 474, 475

MSP430 Peripheral Driver Library User's Guide, Texas Instruments, 2015. 475

MSP430FR58xx, MSP430FR59xx, and MSP430FR6xx Family User's, (SLAU367O), Texas Instruments, 2017. 465, 466, 467, 474, 475, 476, 479

Noise Reduction Techniques for Microcontroller-Based Systems, (AN1705/D), Freescale Semiconductor, 2004. 462, 463

Understanding and Eliminating EMI in Microcontroller Applications, (COP888), Texas Instruments, 1996. 462, 463

11.8 CHAPTER PROBLEMS

Fundamental

1. Describe sources of EMI.
2. Describe EMI coupling mechanisms.
3. Describe three strategies to combat EMI.
4. Describe specific techniques to combat EMI.
5. Describe defensive programming techniques.

Advanced

1. Sketch a UML activity diagram for the CRC algorithm.
2. What is the purpose of generating a CRC checksum?
3. What does a correct checksum indicate? An incorrect one?
4. Research common CRC polynomials. Sketch the corresponding LFSR for each polynomial.
5. What is the purpose of the AES256 subsystem?

Challenging

1. What are the advantages and disadvantages of using different encryption key lengths with the AES256?
2. Sketch a UML activity diagram for the AES256 encryption algorithm.

CHAPTER 12

System-Level Design

Objectives: After reading this chapter, the reader should be able to do the following:

- define an embedded system;
- list multiple aspects related to the design of an embedded system;
- provide a step-by-step approach to design an embedded system;
- discuss design tools and practices related to embedded systems design;
- discuss the importance of system testing;
- apply embedded system design practices in the prototype of a MSP430 based system with several subsystems;
- provide a detailed design for a weather station including hardware layout and interface, structure chart, UML activity diagrams, and an algorithm coded in Energia;
- provide a detailed design for a submersible remotely operated vehicle (ROV) including hardware layout and interface, structure chart, UML activity diagrams, and an algorithm coded in Energia; and
- provide a detailed design for a four-wheel drive (4WD) mountain maze navigating robot including hardware layout and interface, structure chart, UML activity diagrams, and an algorithm coded in Energia.

12.1 OVERVIEW

This chapter provides a step-by-step, methodical approach toward designing advanced embedded systems. We begin with a definition of an embedded system. We then explore the process of how to successfully (and with low stress) develop an embedded system prototype that meets established requirements. The overview of embedded system design techniques was adapted with permission from earlier Morgan & Claypool projects. Also, the projects have been adapted with permission for the MSP430. We also emphasize good testing techniques. We conclude the chapter with several extended examples. The examples illustrate the embedded system design process in the development and prototype of a weather station, a submersible ROV, and a 4WD mountain maze navigating robot. We use the MSP-EXP430FR5994 LaunchPad in the examples to allow for the greatest project extension.

12.2 WHAT IS AN EMBEDDED SYSTEM?

An embedded system is typically designed for a specific task. It contains a processor to collect system inputs and generate system outputs. The link between system inputs and outputs is provided by a coded algorithm stored within the processor's resident memory. What makes embedded systems design so challenging and interesting is the design must also provide for proper electrical interface for the input and output devices, potentially limited on-chip resources, human interface concepts, the operating environment of the system, cost analysis, related standards, and manufacturing aspects [Anderson, 2008]. Through careful application of this material you will be able to design and prototype embedded systems based on MSP430.

12.3 EMBEDDED SYSTEM DESIGN PROCESS

There are many formal design processes that we could study. We concentrate on the steps that are common to most. We purposefully avoid formal terminology of a specific approach and instead concentrate on the activities that are accomplished during the development of a system prototype. The design process we describe is illustrated in Figure 12.1 using a UML activity diagram. We discuss the UML activity diagrams later in this section.

12.3.1 PROJECT DESCRIPTION

The goal of the project description step is to determine what the system is ultimately supposed to do. Questions to raise and answer during this step include, but are not limited to, the following.

- What is the system supposed to do?
- Where will it be operating and under what conditions?
- Are there any restrictions placed on the system design?

To answer these questions, the designer interacts with the client to ensure clear agreement on what is to be done. The establishment of clear, definable system requirements may require considerable interaction between the designer and the client. It is essential that both parties agree on system requirements before proceeding further in the design process. The final result of this step is a detailed listing of system requirements and related specifications. If you are completing this project for yourself, you must still carefully and thoughtfully complete this step.

12.3.2 BACKGROUND RESEARCH

Once a detailed list of requirements has been established, the next step is to perform background research related to the design. In this step, the designer will ensure they understand all requirements and features required by the project. This will again involve interaction between the designer and the client. The designer will also investigate applicable codes, guidelines, protocols, and standards related to the project. This is also a good time to start thinking about the

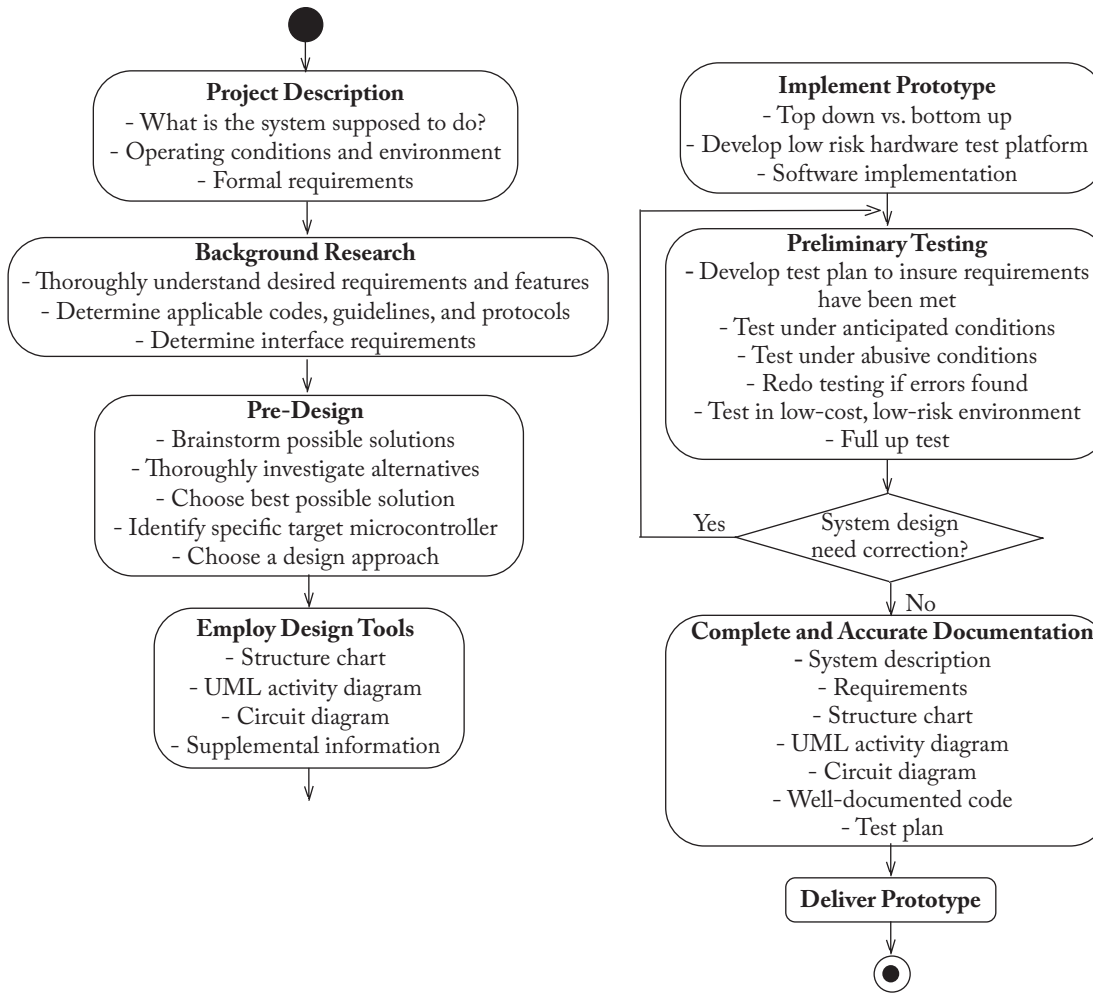


Figure 12.1: Embedded system design process.

interface between different portions of the input and output devices peripherally connected to the processor. The ultimate objective of this step is to have a thorough understanding of the project requirements, related project aspects, and any interface challenges within the project.

12.3.3 PRE-DESIGN

The goal of the pre-design step is to convert a thorough understanding of the project into possible design alternatives. Brainstorming is an effective tool in this step. Here, a list of alternatives is developed. Since an embedded system involves hardware and/or software, the designer can investigate whether requirements could be met with a hardware only solution or some combination of hardware and software. Generally speaking, a hardware only solution executes faster; however, the design is fixed once fielded. On the other hand, a software implementation provides flexibility but a slower execution speed. Most embedded design solutions will use a combination of both hardware and software to capitalize on the inherent advantages of each.

Once a design alternative has been selected, the general partition between hardware and software can be determined. It is also an appropriate time to select a specific hardware device to implement the prototype design. If a technology has been chosen, it is now time to select a specific processor. This is accomplished by answering the following questions.

- What processor systems or features (i.e., ADC, PWM, timer, etc.) are required by the design?
- How many I/O pins are required by the design?
- What type of memory components are required?
- What is the maximum anticipated operating speed of the processor expected to be?

Due to the variety of onboard systems, clock speed, and low cost; the MSP430 may be used in a wide array of applications typically held by microcontrollers and advanced processors.

12.3.4 DESIGN

With a clear view of system requirements and features, a general partition determined between hardware and software, and a specific processor chosen; it is now time to tackle the actual design. It is important to follow a systematic and disciplined approach to design. This will allow for low stress development of a documented design solution that meets requirements. In the design step, several tools are employed to ease the design process. They include the following:

- employing a top-down design, bottom-up implementation approach,
- using a structure chart to assist in partitioning the system,
- using a UML activity diagram to work out program flow, and

- developing a detailed circuit diagram of the entire system.

Let's take a closer look at each of these. The information provided here is an abbreviated version of the one provided in *Microcontrollers Fundamentals for Engineers and Scientists*. The interested reader is referred there for additional details and an in-depth example [Barrett and Pack, 2006].

Top-down design, bottom-up implementation. An effective tool to start partitioning the design is based on the techniques of top-down design, bottom-up implementation. In this approach, you start with the overall system and begin to partition it into subsystems. At this point of the design, you are not concerned with how the design will be accomplished but how the different pieces of the project will fit together. A handy tool to use at this design stage is the structure chart. The structure chart shows how the hierarchy of system hardware and software components will interact and interface with one another. You should continue partitioning system activity until each subsystem in the structure chart has a single definable function. Directional arrows are used to indicate data flow in and out of a function.

UML activity diagram. Once the system has been partitioned into pieces, the next step is to work out the details of the operation of each subsystem previously identified. Rather than beginning to code each subsystem as a function, work out the information and control flow of each subsystem using another design tool: the UML activity diagram. The activity diagram is simply a UML compliant flow chart. UML is a standardized method of documenting systems. The activity diagram is one of the many tools available from UML to document system design and operation. The basic symbols used in a UML activity diagram for a processor based system are provided in Figure 12.2 [Fowler, 2000].

To develop the UML activity diagram for the system, we can use a top-down, bottom-up, or a hybrid approach. In the top-down approach, we begin by modeling the overall flow of the algorithm from a high level. If we choose to use the bottom-up approach, we would begin at the bottom of the structure chart and choose a subsystem for flow modeling. The specific course of action chosen depends on project specifics. Often, a combination of both techniques, a hybrid approach, is used. You should work out all algorithm details at the UML activity diagram level prior to coding any software. If you cannot explain system operation at this higher level first, you have no business being down in the detail of developing the code. Therefore, the UML activity diagram should be of sufficient detail so you can code the algorithm directly from it [Dale and Lilly, 1995].

In the design step, a detailed circuit diagram of the entire system is developed. It will serve as a roadmap to implement the system. It is also a good idea at this point to investigate available design information relative to the project. This would include hardware design examples, software code examples, and application notes available from manufacturers. As before, use a subsystem approach to assemble the entire circuit. The basic building block interface circuits dis-

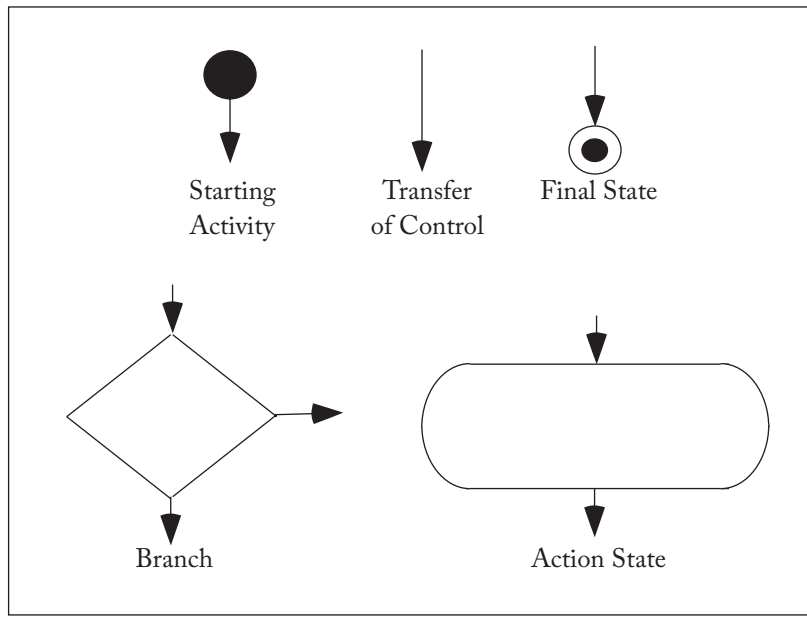


Figure 12.2: UML activity diagram symbols. (Adapted from Fowler [2000].)

cussed in the previous chapter may be used to assemble the complete circuit. At the completion of this step, the prototype design is ready for implementation and testing.

12.3.5 IMPLEMENT PROTOTYPE

To successfully implement a prototype, an incremental approach should be followed. Again, the top-down design, bottom-up implementation provides a solid guide for system implementation. In an embedded system design involving both hardware and software, the hardware system including the processor should be assembled first. This provides the software the required signals to interact with. As the hardware prototype is assembled on a prototype board, each component is tested for proper operation as it is brought online. This allows the designer to pinpoint malfunctions as they occur.

Once the hardware prototype is assembled, coding may commence. It is important to note that on larger projects software and hardware may be developed concurrently. As before, software should be incrementally brought online. You may use a top-down, bottom-up, or hybrid approach depending on the nature of the software. The important point is to bring the software online incrementally such that issues can be identified and corrected early on.

It is highly recommended that low-cost stand-in components be used when testing the software with the hardware components. For example, push buttons, potentiometers, and LEDs may be used as low-cost stand-in component simulators for expensive input instrumentation

devices and expensive output devices such as motors. This allows you to insure the software is properly operating before using it to control the actual components.

12.3.6 PRELIMINARY TESTING

To test the system, a detailed test plan must be developed. Tests should be developed to verify that the system meets all of its requirements and also intended system performance in an operational environment. The test plan should also include scenarios in which the system is used in an unintended manner. As before, a top-down, bottom-up, or hybrid approach can be used to test the system. In a bottom-up approach individual units are tested first.

Once the test plan is completed, actual testing may commence. The results of each test should be carefully documented. As you go through the test plan, you will probably uncover a number of run-time errors in your algorithm. After you correct a run-time error, the entire test plan must be repeated. This ensures that the new fix does not have an unintended effect on another part of the system. Also, as you process through the test plan, you will probably think of other tests that were not included in the original test document. These tests should be added to the test plan. As you go through testing, realize your final system is only as good as the test plan that supports it!

Once testing is complete, you should accomplish another level of testing where you intentionally try to "jam up" the system. In other words, try to get your system to fail by trying combinations of inputs that were not part of the original design. A robust system should continue to operate correctly in this type of an abusive environment. It is imperative that you design robustness into your system. When testing on a low cost simulator is complete, the entire test plan should be performed again with the actual system hardware. Once this is completed you should have a system that meets its requirements!

12.3.7 COMPLETE AND ACCURATE DOCUMENTATION

With testing complete, the system design should be thoroughly documented. Much of the documentation will have already been accomplished during system development. Documentation will include the system description, system requirements, the structure chart, the UML activity diagrams documenting program flow, the test plan, results of the test plan, system schematics, and properly documented code. To properly document code, you should carefully comment all functions describing their operation, inputs, and outputs. Also, comments should be included within the body of the function describing key portions of the code. Enough detail should be provided such that code operation is obvious. It is also extremely helpful to provide variables and functions within your code names that describe their intended use.

You might think that comprehensive system documentation is not worth the time or effort to complete it. Complete documentation pays rich dividends when it is time to modify, repair, or update an existing system. Also, well-documented code may be often reused in other projects: a method for efficient and timely development of new systems.

In the next sections we provide detailed examples of the system design process for a weather station, a submersible robot, and a 4WD robot capable of navigating through a mountainous maze.

12.4 MSP430FR5994: WEATHER STATION

In this project, we design a weather station to sense wind direction and ambient temperature. The wind direction will be displayed on LEDs arranged in a circular pattern. The wind direction and temperature will also be transmitted serially via the SPI from the microcontroller to the onboard microSD flash memory card for data logging.

12.4.1 REQUIREMENTS

The requirements for this system include:

- design a weather station to sense wind direction and ambient temperature;
- wind direction should be displayed on LEDs arranged in a circular pattern; and
- wind direction and temperature should be transmitted serially from the microcontroller to the onboard microSD card for storage.

12.4.2 STRUCTURE CHART

To begin, the design process, a structure chart is used to partition the system into definable pieces. We employ a top-down design/bottom-up implementation approach. The structure chart for the weather station is provided in Figure 12.3. The three main microcontroller subsystems needed for this project are the SPI for serial communication, the ADC12 system to convert the analog voltage from the LM34 temperature sensor and weather vane into digital signals, and the wind direction display. The system is partitioned until the lowest level of the structure chart contains "doable" pieces of hardware components or software functions. Data flow is shown on the structure chart as directed arrows.

12.4.3 CIRCUIT DIAGRAM

Analog sensors: The circuit diagram for the weather station is provided in Figure 12.4. The weather station is equipped with two input sensors: the LM34 to measure temperature and the weather vane to measure wind direction. Both of the sensors provide an analog output that is fed to the MSP430. The LM34 provides 10 mV output per degree Fahrenheit. The weather vane provides a voltage output from 0–3.3 VDC for different wind direction as shown in Figure 12.4. The weather vane must be oriented to a known direction with the output voltage at this direction noted. We assume that 0 VDC corresponds to North.

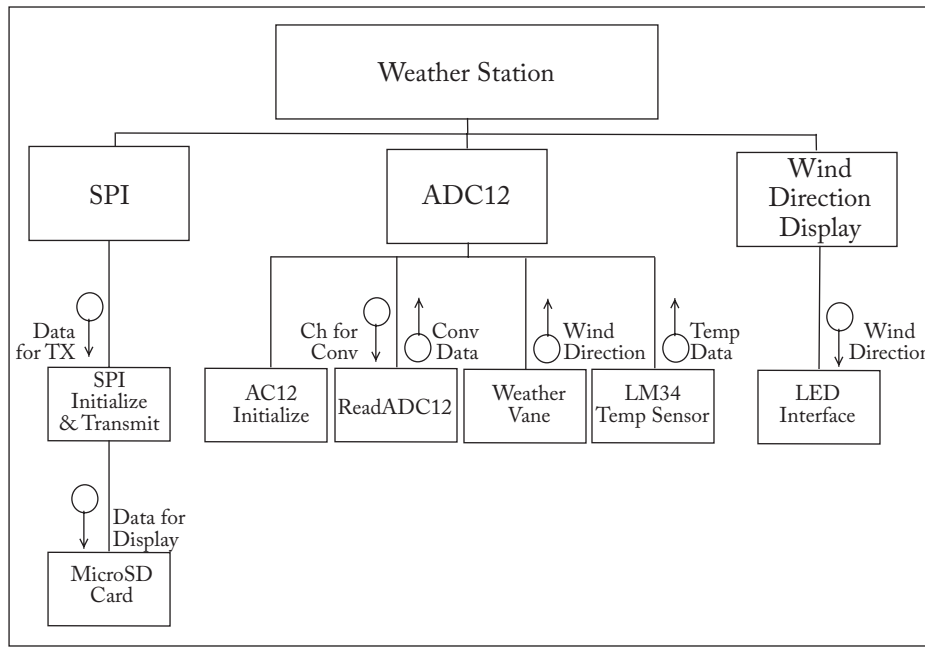


Figure 12.3: Weather station structure chart.

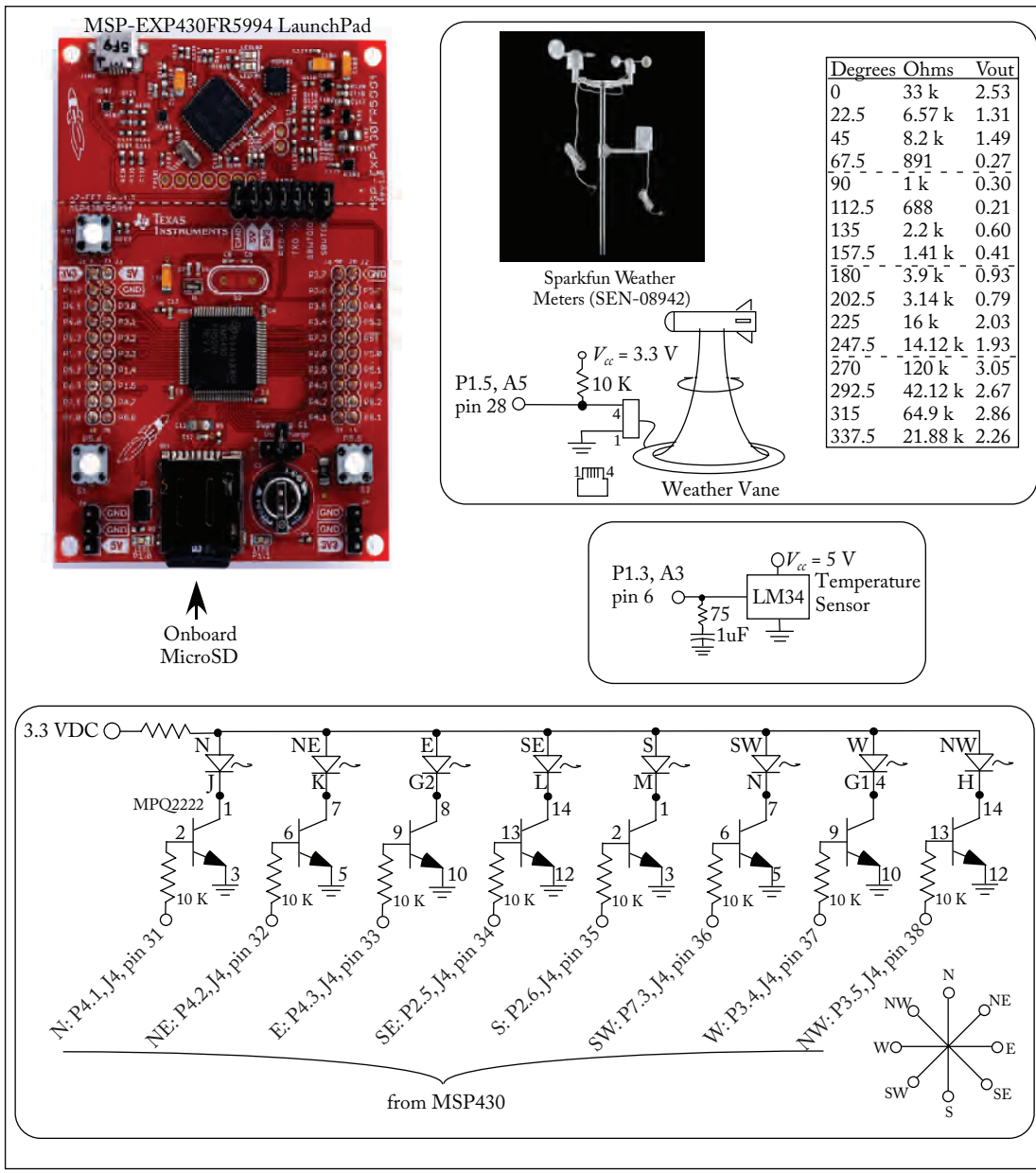
Wind direction display: There are eight different LEDs to drive for the wind direction indicator. An interface circuit is required for each LED as shown in the figure.

12.4.4 UML ACTIVITY DIAGRAMS

The UML activity diagram for the main program is provided in Figure 12.5. After initializing the subsystems, the program enters a continuous loop where temperature and wind direction is sensed and displayed on the LCD and the LED display. The sensed values are then transmitted via the SPI to the MMC/SD card. The system then enters a delay to set how often the temperature and wind direction parameters are updated. We have you construct the individual UML activity diagrams for each function as an end of chapter exercise.

12.4.5 MICROCONTROLLER CODE

For quick prototyping the first version of the code for this project is rendered in Energia. After initializing the system, the code continuously loops and reads temperature and wind direction data, and displays the data to the LED array. A delay should be inserted in the loop to determine how often the weather data should be collected. During development code status is sent to the serial monitor. Printing to the serial monitor is enabled with the variable "troubleshoot."



Degrees	Ohms	Vout
0	33 k	2.53
22.5	6.57 k	1.31
45	8.2 k	1.49
67.5	891	0.27
90	1 k	0.30
112.5	688	0.21
135	2.2 k	0.60
157.5	1.41 k	0.41
180	3.9 k	0.93
202.5	3.14 k	0.79
225	16 k	2.03
247.5	14.12 k	1.93
270	120 k	3.05
292.5	42.12 k	2.67
315	64.9 k	2.86
337.5	21.88 k	2.26

Figure 12.4: Circuit diagram for weather station. (Illustrations used with permission of Sparkfun Electronics (www.sparkfun.com) and Texas Instruments (www.ti.com).)

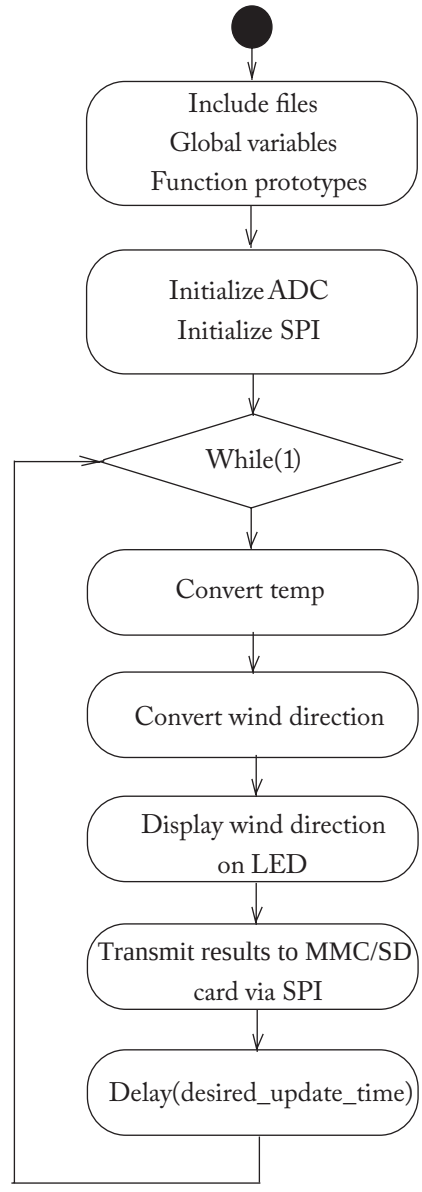


Figure 12.5: Weather station UML activity diagram.

```
 //*********************************************************************
//weather station
//- Equipped with:
//- Sparkfun weather meters (SEN-08942)
//  -- rain gauge
//  -- anemometer
//  -- wind vane
//- LM34 temperature sensor
//- Sparkfun LCD-09067, serial enabled 16x2 LCD, 3.3 VDC
//*********************************************************************

//analog input pins
#define wind_dir      28       //analog pin - weather vane
#define temp_sensor    6       //analog pin - LM34 temp sensor

//digital output pins - LED indicators
#define N_LED         31       //digital pin - N LED
#define NE_LED        32       //digital pin - NE LED
#define E_LED         33       //digital pin - E LED
#define SE_LED        34       //digital pin - SE LED
#define S_LED         35       //digital pin - S LED
#define SW_LED        36       //digital pin - SW LED
#define W_LED         37       //digital pin - W LED
#define NW_LED        38       //digital pin - NW LED

int wind_dir_value;            //declare variable for wind dir
int temp_value;                //declare variable for temp
int troubleshoot = 1;          //1: serial monitor prints

float wind_direction_float;
float temp_value_float;

void setup()
{
//LED indicators
pinMode(N_LED,  OUTPUT);             //config pin for digital out - N LED
pinMode(NE_LED, OUTPUT);             //config pin for digital out - NE LED
pinMode(E_LED,  OUTPUT);             //config pin for digital out - E LED
pinMode(SE_LED, OUTPUT);             //config pin for digital out - SE LED
```

```
pinMode(S_LED,  OUTPUT);              //config pin for digital out - S LED
pinMode(SW_LED, OUTPUT);              //config pin for digital out - SW LED
pinMode(W_LED,  OUTPUT);              //config pin for digital out - W LED
pinMode(NW_LED, OUTPUT);              //config pin for digital out - NW LED

//Serial monitor - open serial communications
if(troubleshoot == 1) Serial.begin(9600);
}

void loop()
{
//read two sensors and append to the string
//analog read returns value between 0 and 1023
wind_dir_value   = analogRead(wind_dir);
temp_value      = analogRead(temp_sensor);

if(troubleshoot == 1)Serial.println(wind_dir_value);
if(troubleshoot == 1)Serial.println(temp_value);

//LM34 provides 10 mV/degree
temp_value =(int)(((temp_value/1023.0) * 3.3)/.010);
if(troubleshoot == 1)Serial.println(temp_value);

//display wind direction
display_wind_direction(wind_dir_value);
}

//*********************************************************************

void display_wind_direction(unsigned int wind_dir_int)
{
float wind_dir_float;
                                      //convert wind direction to float
wind_dir_float = wind_dir_int/1023.0 * 3.3;

if(troubleshoot == 1)Serial.println(wind_dir_float);

//N - LED0
if((wind_dir_float <= 2.56)&&(wind_dir_float > 2.50))
```

```
   {
   digitalWrite(N_LED,  HIGH);  digitalWrite(NE_LED, LOW);
   digitalWrite(E_LED,  LOW);   digitalWrite(SE_LED, LOW);
   digitalWrite(S_LED,  LOW);   digitalWrite(SW_LED, LOW);
   digitalWrite(W_LED,  LOW);   digitalWrite(NW_LED, LOW);
   }

//NE - LED1
if((wind_dir_float > 1.46)&&(wind_dir_float <= 1.52))
   {
   digitalWrite(N_LED,  LOW);   digitalWrite(NE_LED,  HIGH);
   digitalWrite(E_LED,  LOW);   digitalWrite(SE_LED, LOW);
   digitalWrite(S_LED,  LOW);   digitalWrite(SW_LED, LOW);
   digitalWrite(W_LED,  LOW);   digitalWrite(NW_LED, LOW);
   }

//E - LED2
if((wind_dir_float > 0.27)&&(wind_dir_float <= 0.33))
   {
   digitalWrite(N_LED,  LOW);   digitalWrite(NE_LED, LOW);
   digitalWrite(E_LED, HIGH);   digitalWrite(SE_LED, LOW);
   digitalWrite(S_LED,  LOW);   digitalWrite(SW_LED, LOW);/
   digitalWrite(W_LED,  LOW);   digitalWrite(NW_LED, LOW);
   }

//SE - LED3
if((wind_dir_float > 0.57)&&(wind_dir_float <= 0.63))
   {
   digitalWrite(N_LED,  LOW);   digitalWrite(NE_LED, LOW);
   digitalWrite(E_LED,  LOW);   digitalWrite(SE_LED, HIGH);
   digitalWrite(S_LED,  LOW);   digitalWrite(SW_LED, LOW);
   digitalWrite(W_LED,  LOW);   digitalWrite(NW_LED, LOW);
   }

//S - LED4
if((wind_dir_float > 0.9)&&(wind_dir_float <= 0.96))
   {
   digitalWrite(N_LED,  LOW);   digitalWrite(NE_LED, LOW);
   digitalWrite(E_LED,  LOW);   digitalWrite(SE_LED, LOW);
```

```
   digitalWrite(S_LED,  HIGH);  digitalWrite(SW_LED, LOW);
   digitalWrite(W_LED,  LOW);   digitalWrite(NW_LED, LOW);
   }

//SW - LED5
if((wind_dir_float > 2.0)&&(wind_dir_float <= 2.06))
   {
   digitalWrite(N_LED,  LOW);   digitalWrite(NE_LED, LOW);
   digitalWrite(E_LED,  LOW);   digitalWrite(SE_LED, LOW);
   digitalWrite(S_LED,  LOW);   digitalWrite(SW_LED, HIGH);
   digitalWrite(W_LED,  LOW);   digitalWrite(NW_LED, LOW);
   }

//W - LED6
if((wind_dir_float > 3.02)&&(wind_dir_float <= 3.08))
   {
   digitalWrite(N_LED,  LOW);   digitalWrite(NE_LED, LOW);
   digitalWrite(E_LED,  LOW);   digitalWrite(SE_LED, LOW);
   digitalWrite(S_LED,  LOW);   digitalWrite(SW_LED, LOW);
   digitalWrite(W_LED,  HIGH);  digitalWrite(NW_LED, LOW);
   }
//NW - LED7
if((wind_dir_float > 2.83)&& (wind_dir_float <= 2.89))
   {
   digitalWrite(N_LED,  LOW);   digitalWrite(NE_LED, LOW);
   digitalWrite(E_LED,  LOW);   digitalWrite(SE_LED, LOW);
   digitalWrite(S_LED,  LOW);   digitalWrite(SW_LED, LOW);
   digitalWrite(W_LED,  LOW);   digitalWrite(NW_LED, HIGH);
   }
}

//*********************************************************************
```

12.4.6 PROJECT EXTENSIONS

The control system provided above has a set of very basic features. Here are some possible extensions for the system.

- Equip the weather station with an LCD display.

- In addition to the wind vane, the Sparkfun weather meters (SEN-08942) includes a rain gauge and anemometer. Add these features to the weather station.
- In Chapter 6 we discussed the onboard microSD card. Equip the weather station project with the onboard microSD card to log time hacks and weather data.

12.5 SUBMERSIBLE ROBOT

The area of submersible robots is fascinating and challenging. To design a robot is quite complex (yet fun). To add the additional requirement of waterproofing key components provides an additional level of challenge. (Water and electricity do not mix!) In this section we provide the construction details and a control system for a remotely operated vehicle, an ROV. Specifically, we develop the structure and control system for the SeaPerch style ROV, as shown in Figure 12.6. By definition, an ROV is equipped with a tether umbilical cable that provides power and control signals from a surface support platform. An autonomous underwater vehicle (AUV) carries its own power and control equipment and does not require surface support [Seaperch].

Details on the construction and waterproofing of an ROV are provided in the excellent and fascinating *Build Your Own Underwater Robot and Other Wet Projects* by Harry Bohm and Vickie Jensen. For an advanced treatment, please see *The ROV Manual–A User Guide for Remotely Operated Vehicles* by Robert Crist and Robert Wernli, Sr. There is a national-level competition for students based on the SeaPerch ROV. The goal of the program is to stimulate interest in the next generation of marine-related engineering specialties [Seaperch].

12.5.1 APPROACH

This is a challenging project; however, we take a methodical, step-by-step approach to successful design and construction of the ROV. We complete the design tasks in the following order.

1. Determine requirements.
2. Design and construct ROV structure.
3. Design and fabricate control electronics.
4. Design and implement control software using Energia.
5. Construct and assemble a prototype.
6. Test the prototype.

12.5.2 REQUIREMENTS

The requirements for the ROV system include the following.

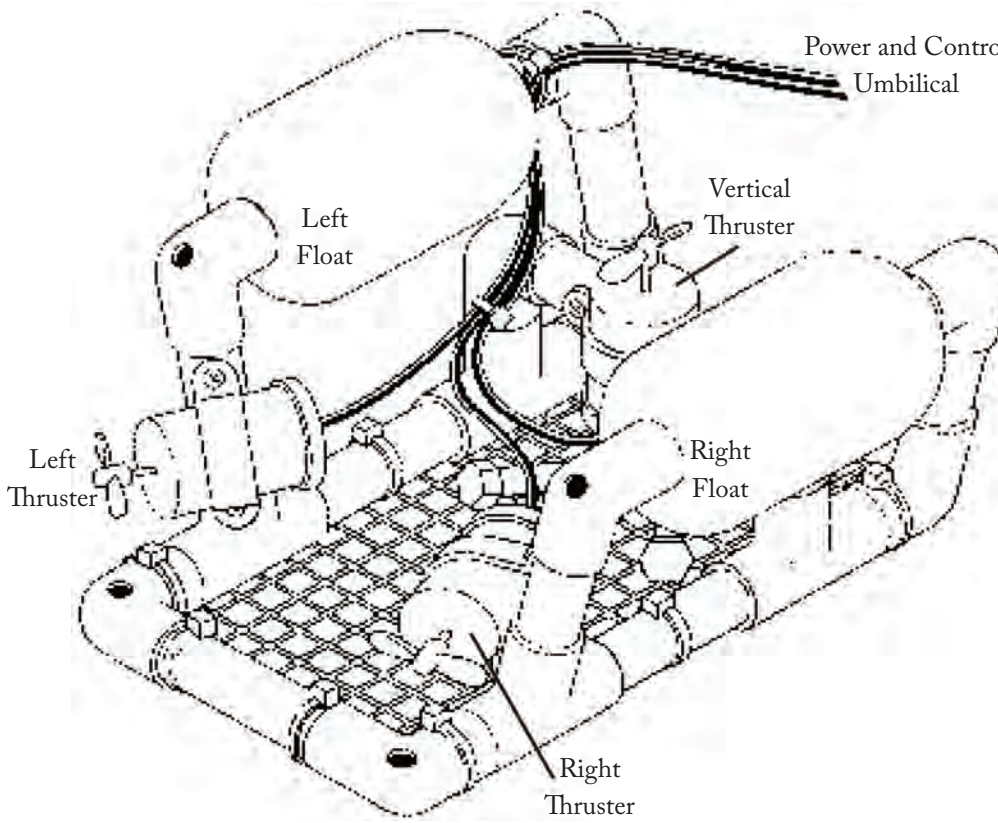


Figure 12.6: SeaPerch ROV. (Adapted and used with permission of Bohm and Jensen, West Coast Words Publishing.)

- Develop a control system to allow a three thruster (motor or bilge pump) ROV to move forward, left (port) and right (starboard).
- The ROV will be pushed down to a shallow depth via a vertical thruster and return to surface based on its own, slightly positive buoyancy.
- ROV movement will be under joystick control.
- LEDs are used to indicate thruster assertion.
- All power and control circuitry will be maintained in a surface support platform as shown in Figure 12.7.
- An umbilical cable connects the support platform to the ROV.

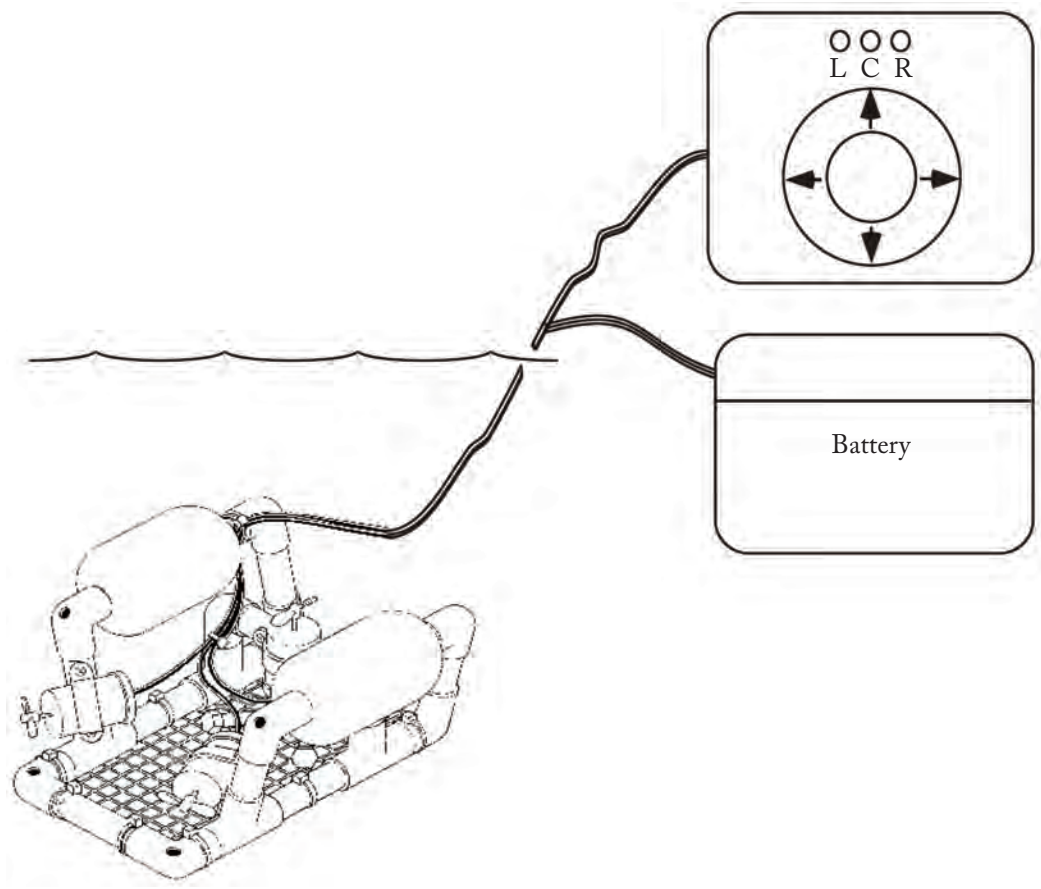


Figure 12.7: Power and control are provided remotely to the SeaPerch ROV. (Adapted and used with permission of Bohm and Jensen, West Coast Words Publishing.)

12.5.3 ROV STRUCTURE

The ROV structure is shown in Figure 12.8. The structure is constructed with 0.75-in PVC piping. The structure is assembled quickly using "T" and corner connectors. The pieces are connected using PVC glue or machine screws. The PVC pipe and connectors are readily available in hardware and home improvement stores.

The fore or bow portion of the structure is equipped with plexiglass panels to serve as mounting bulkheads for the thrusters. The panels are mounted to the PVC structure using ring clamps. Either waterproofed electric motors or submersible bilge pumps are used as thrusters. A bilge pump is a pump specifically designed to remove water from the inside of a boat. The pumps are powered from a 12 VDC source and have typical flow rates from 360 to over 3,500 gallons per

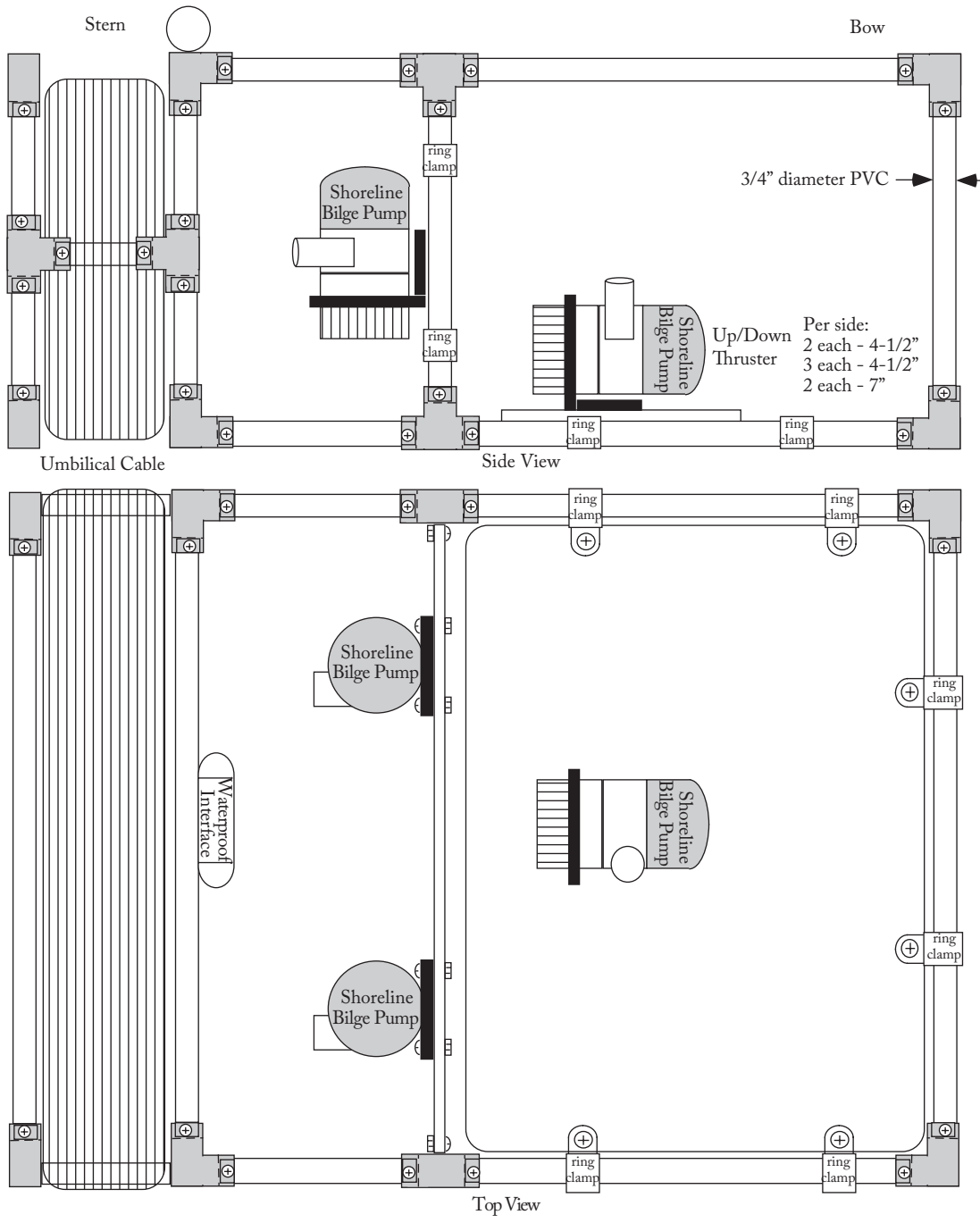


Figure 12.8: SeaPerch ROV structure.

minute. They range in price from U.S. $20–$80 (www.shorelinemarinedevelopment.com). Details on waterproofing electric motors are provided in *Build Your Own Underwater Robot and Other Wet Projects*. We use three Shoreline Bilge Pumps rated at 600 gallons per minute (GPM). They are available online from www.walmart.com.

The aft or stern portion of the structure is designed to hold the flexible umbilical cable. The cable provides a link between the MSP430-based control system and the thrusters. Each thruster may require up to 1–2 amps of current. Therefore, a four-conductor, 16 AWG, braided (for flexibility) conductor cable is recommended. The cable is interfaced to the bilge pump leads using soldered connections or Butt connectors. The interface should be thoroughly waterproofed using caulk. For this design the interface was placed within a section of PVC pipe equipped with end caps. The resulting container is filled with waterproof caulk.

Once the ROV structure is complete its buoyancy is tested. This is accomplished by placing the ROV structure in water. The goal is to achieve a slightly positive buoyancy. With positive buoyancy the structure floats. With neutral buoyancy the structures hovers beneath the surface. With negative buoyancy the structure sinks. A more positive buoyancy way be achieved by attaching floats or foam to the structure tubing. A more negative buoyancy may be achieved by adding weights to the structure [Bohm and Jensen, 2012].

12.5.4 STRUCTURE CHART

The SeaPerch structure chart is provided in Figure 12.9. As can be seen in the figure, the SeaPerch control system will accept input from the five position joystick (left, right, select, up, and down). We use the Sparkfun thumb joystick (Sparkfun COM-09032) mounted to a breakout board (Sparkfun BOB-09110), as shown in Figure 12.10. The joystick schematic and connections to MSP430 are provided in Figures 12.11 and 12.12.

In response to user joystick input, the SeaPerch control algorithm will issue a control command indicating desired ROV direction. In response to this desired direction command, the motor control algorithm will issue control signals to assert the appropriate thrusters and LEDs.

12.5.5 CIRCUIT DIAGRAM

The circuit diagram for the SeaPerch control system is provided in Figure 12.11. The thumb joystick is used to select desired ROV direction. The thumb joystick contains two built-in potentiometers (horizontal and vertical). A reference voltage of 3.3 VDC is applied to the VCC input of the joystick. As the joystick is moved, the horizontal (HORZ) and vertical (VERT) analog output voltages will change to indicate the joystick position. The joystick is also equipped with a digital select (SEL) button. The SEL button is used to activate an ROV dive. The joystick is interfaced to MSP430 as shown in Figure 12.11.

There are three LED interface circuits connected to MSP430 header pins P4.2, P4.4, and P4.5. The LEDs illuminate to indicate the left, vertical, and right thrusters have been asserted.

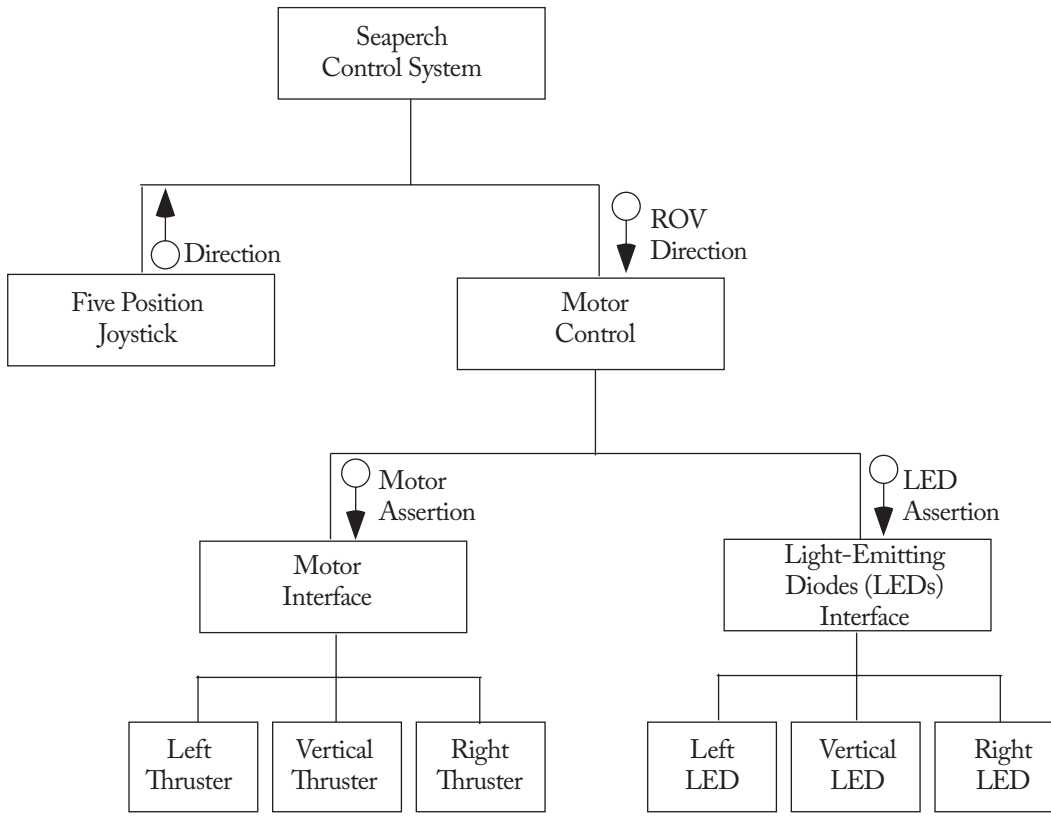
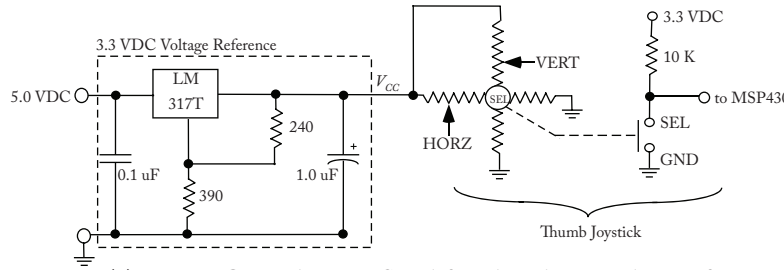
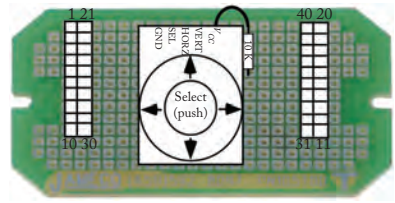


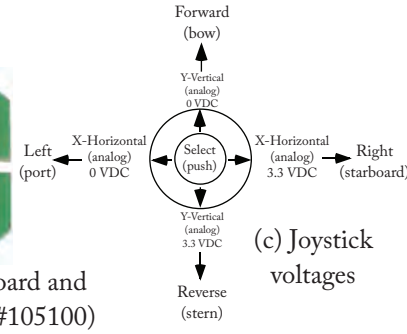
Figure 12.9: SeaPerch ROV structure chart.



(a) 3.3 VDC regulator to Sparkfun thumb joystick interface



(b) Sparkfun thumb joystick on breakout board and Jameco Prototype Builder 1.6" × 2.7" PCB (#105100)



(c) Joystick voltages

Figure 12.10: Thumb joystick mounted to a breakout board. (Illustration used with permission of Jameco (www.jameco.com).)

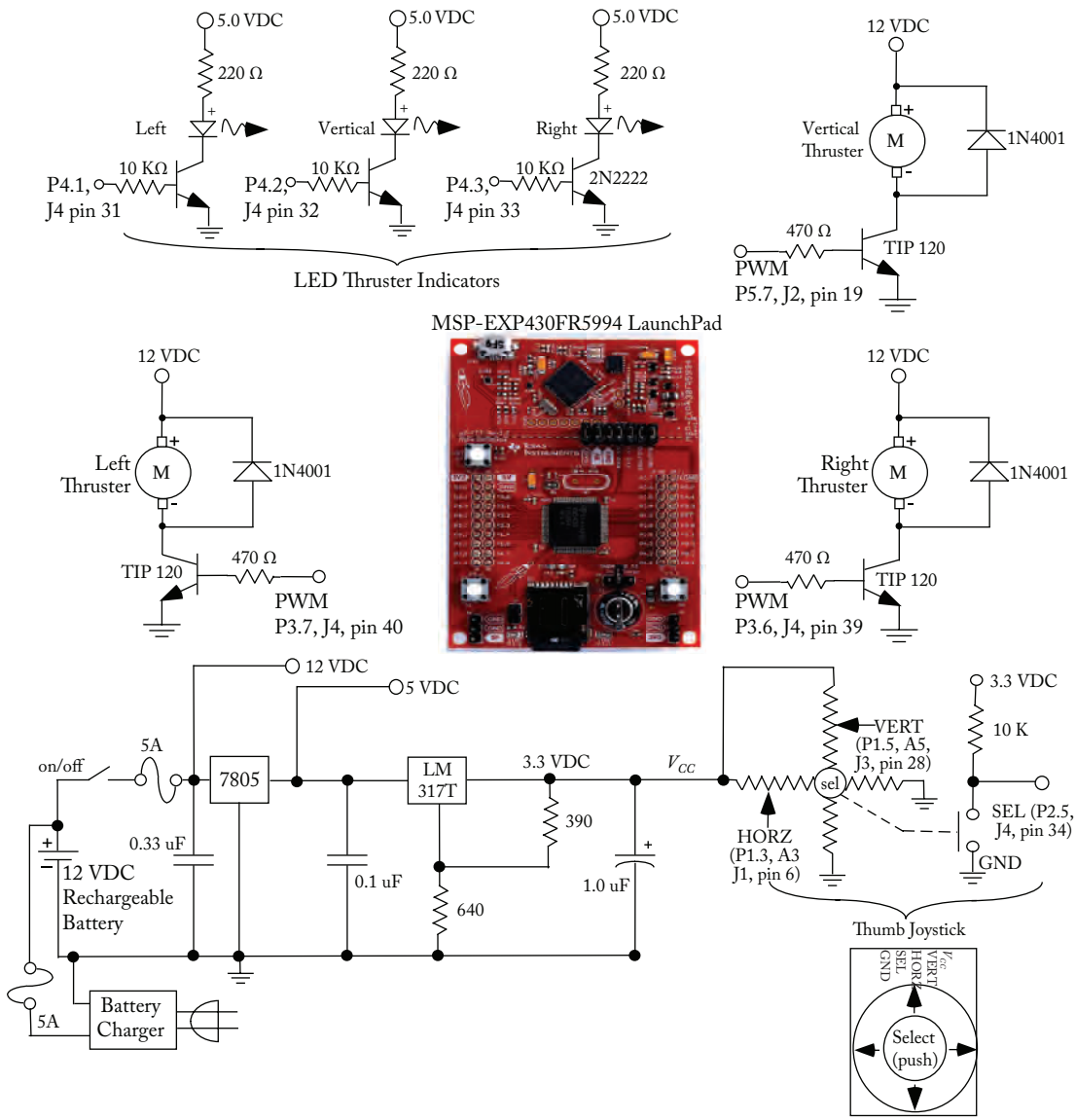


Figure 12.11: SeaPerch ROV interface control. (Illustration used with permission of Texas Instruments (www.ti.com).)

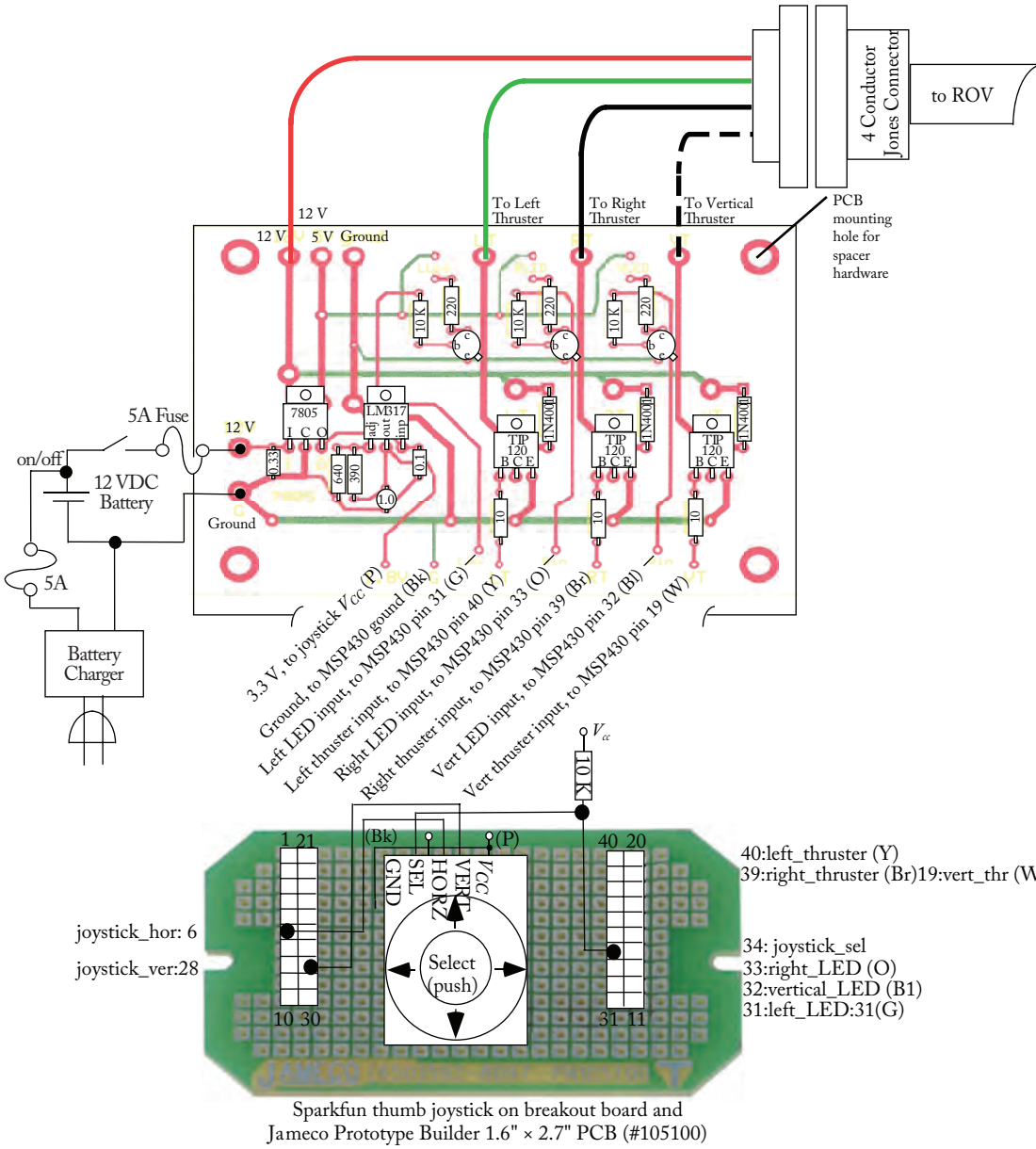


Figure 12.12: SeaPerch ROV printed circuit board interface. (Illustration used with permission of Jameco (www.jameco.com).)

As previously mentioned, the prime mover for the ROV are three bilge pumps. The left and right bilge pumps are driven by pulse width modulation channels (MSP430 P2.4 and P2.6) via power NPN Darlington transistors (TIP 120), as shown in Figure 12.11. The vertical thrust is under digital pin control P2.5 equipped with NPN Darlington transistor (TIP 120) interface. Both the LED and the pump interfaces were discussed in an earlier chapter.

The interface circuitry between the MSP430 LaunchPad and the bilge pumps is mounted on a printed circuit board (PCB) within the control housing. The interface between MSP430, the PCB, and the umbilical cable is provided in Figure 12.12.

12.5.6 UML ACTIVITY DIAGRAM

The SeaPerch control system UML activity diagram is provided in Figure 12.13. After initializing the MSP430 pins the control algorithm is placed in a continuous loop awaiting user input. In response to user input, the algorithm determines desired direction of ROV travel and asserts appropriate control signals for the LED and motors.

12.5.7 MSP430 CODE

In this example we use the thumb joystick to control the left and right thruster (motor or bilge pump). The joystick provides a separate voltage from 0–3.3 VDC for the horizontal (HORZ) and vertical (VERT) position of the joystick. We use this voltage to set the duty cycle of the pulse width modulated (PWM) signals sent to the left and right thrusters. The select push-button (SEL) on the joystick is used to assert the vertical thruster. The analog read function (analogRead) is used to read the X and Y position of the joystick. A value from 0–1023 is reported from the analog read function corresponding to 0–3.3 VDC. After the voltage readings are taken they are scaled to 3.3 VDC for further processing. Joystick activity is divided into multiple zones (0–8), as shown in Figure 12.14. The joystick signal is further processed consistent with the joystick zone selected.

```
//*********************************************************************
//ROV
//In response to joystick input, the SeaPerch control algorithm issues
//a control command indicating desired ROV direction.  In response to
//desired direction command, the motor control algorithm issues
//control signals to assert the appropriate thrusters and LEDs.
//*********************************************************************

//analog input pins
#define joystick_hor       6      //analog pin - joystick horizontal in
#define joystick_ver      28      //analog pin - joystick vertical in

//digital input pin
```

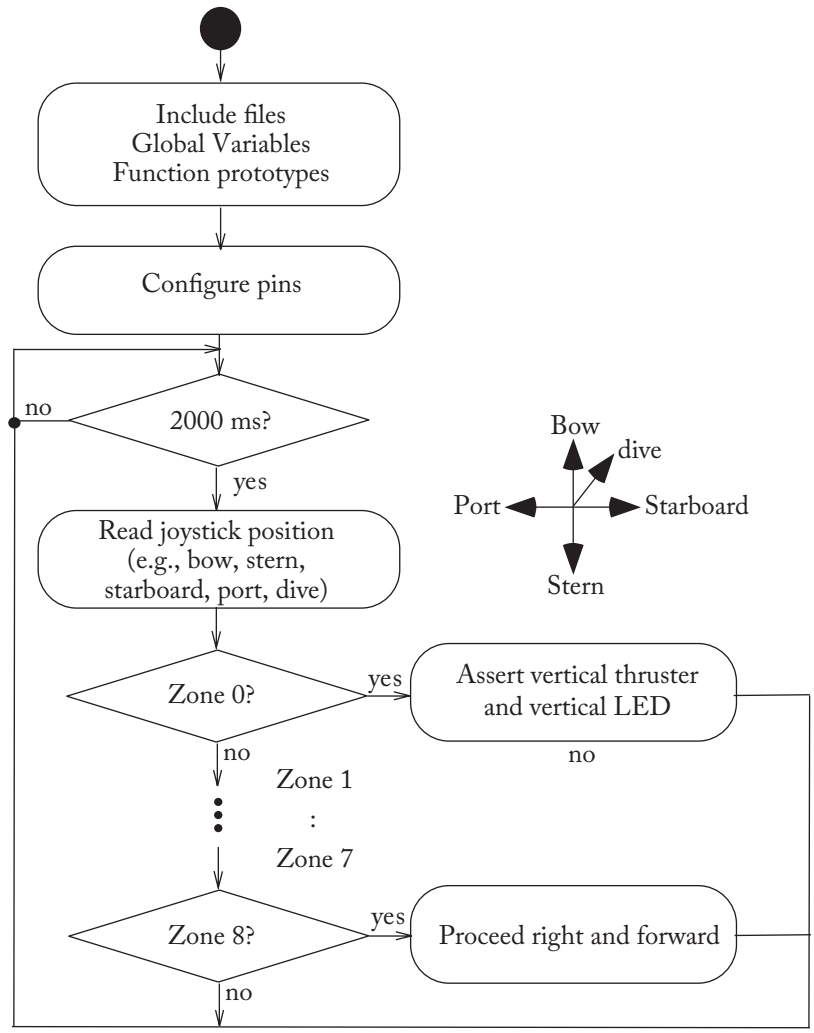


Figure 12.13: SeaPerch ROV UML activity diagram.

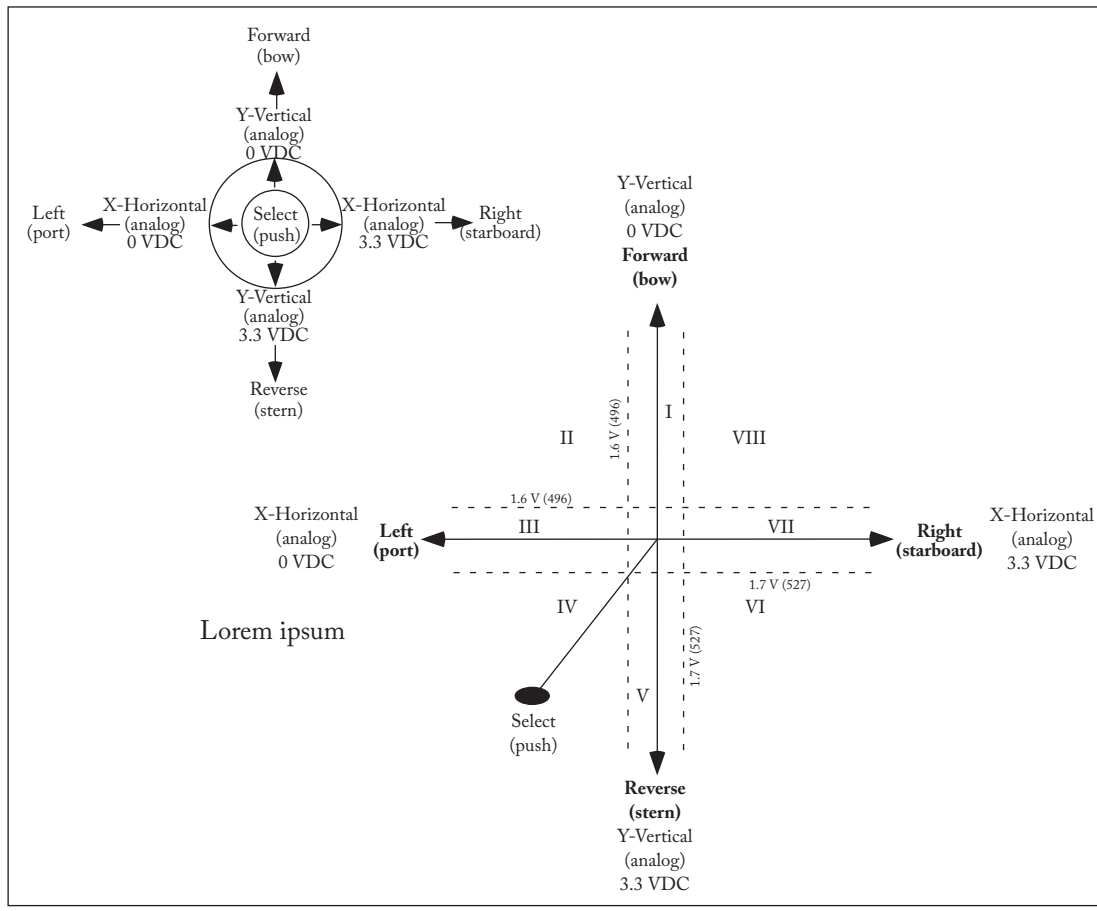


Figure 12.14: Joystick position as related to thruster activity.

```
#define joystick_sel      34      //digital pin - joystick select in

//digital output pins - LED indicators
#define left_LED          31      //digital pin - left LED out
#define vertical_LED      32      //digital pin - vertical LED out
#define right_LED         33      //digital pin - right LED out

//thruster outputs
#define left_thruster     40      //digital pin - left thruster
#define right_thruster    39      //digital pin - right thruster
#define vertical_thruster 19      //digital pin - vertical thruster
```

```
int joystick_hor_value;           //horizontal joystick value
int joystick_ver_value;           //vertical joystick value
int joystick_sel_value;           //joystick select value
int joystick_thrust_on;           //1: thrust on; 0: off
int troubleshoot = 1;             //1: serial monitor prints

void setup()
{
//LED indicators
pinMode(left_LED,     OUTPUT);    //config pin for digital out - left LED
pinMode(vertical_LED, OUTPUT);    //config pin for digital out -
                                  //vertical LED
pinMode(right_LED,    OUTPUT);    //config pin for digital out - right LED

//joystick select input
pinMode(joystick_sel, INPUT);     //config pin for digital in - joystick sel

//thruster outputs
pinMode(left_thruster,     OUTPUT);  //config digital out - left thruster
pinMode(vertical_thruster, OUTPUT);  //config digital out -
                                     //vertical thruster
pinMode(right_thruster,    OUTPUT);  //config digital out - right thruster

//Serial monitor - open serial communications
if(troubleshoot == 1) Serial.begin(9600);
}

void loop()
{
//set update interval
delay(1000);

//turn off LEDs
digitalWrite(left_LED,     LOW);        //left LED     - off
digitalWrite(vertical_LED, LOW);        //vertical LED - off
digitalWrite(right_LED,    LOW);       //right LED    - off

//read hor and vert joystick position
```

```
//analog read returns value between 0 and 1023
joystick_hor_value = analogRead(joystick_hor);
joystick_ver_value = analogRead(joystick_ver);

if(troubleshoot == 1) Serial.println(joystick_hor_value);
if(troubleshoot == 1) Serial.println(joystick_ver_value);

//Convert 0 to 1023 to 0 to 3.3 VDC value
joystick_hor_value = ((joystick_hor_value/1023.0) * 3.3);
if(troubleshoot == 1) Serial.println(joystick_hor_value);

joystick_ver_value = ((joystick_ver_value/1023.0) * 3.3);
if(troubleshoot == 1) Serial.println(joystick_ver_value);

//Read vertical thrust
joystick_thrust_on  = digitalRead(joystick_sel); //vertical thrust?

//*************************************************************
//vertical thrust - active low pushbutton on joystick
//*************************************************************
if(joystick_thrust_on == 0)
  {
  digitalWrite(vertical_thruster, HIGH);
  digitalWrite(vertical_LED,  HIGH);
  if(troubleshoot == 1) Serial.println("Thrust is on!");
  }
else
  {
  digitalWrite(vertical_thruster, LOW);
  digitalWrite(vertical_LED,  LOW);
  if(troubleshoot == 1) Serial.println("Thrust is off!");
  }

//*************************************************************
//*************************************************************
//process different joystick zones
//*************************************************************
//Case 0: Joystick in null position
//Inputs:
```

```
//  X channel between 1.60 to 1.70 VDC - null zone
//  Y channel between 1.60 to 1.70 VDC - null zone
//Output:
//  Shut off thrusters
//*************************************************************

if((joystick_hor_value > 1.60)&&(joystick_hor_value < 1.70)&&
   (joystick_ver_value > 1.60)&&(joystick_ver_value < 1.70))

  {
  if(troubleshoot == 1) Serial.println("Zone 0");

  if(troubleshoot == 1)
  {
  if(troubleshoot == 1) Serial.println(joystick_hor_value);
  if(troubleshoot == 1) Serial.println(joystick_ver_value);
  if(troubleshoot == 1) Serial.println(joystick_thrust_on);
  }

  //assert thrusters to move forward
  analogWrite(left_thruster,  0);
  analogWrite(right_thruster, 0);

  //assert LEDs
  digitalWrite(left_LED,  LOW);       //de-assert left LED
  digitalWrite(right_LED, LOW);       //de-assert right LED
  }

//*************************************************************
//*************************************************************
//process different joystick zones
//*************************************************************
//Case 1:
//Inputs:
//  X channel between 1.60 to 1.70 VDC - null zone
//  Y channel <= 1.60 VDC
//Output:
//  Move forward - provide same voltage to left and right thrusters
//*************************************************************
```

```
if((joystick_hor_value > 1.60)&&(joystick_hor_value < 1.70)&&
    (joystick_ver_value <= 1.60))
  {
  if(troubleshoot == 1) Serial.println("Zone 1");

  //scale joystick vertical to value from 0 to 1
  joystick_ver_value = 1.60 - joystick_ver_value;

  if(troubleshoot == 1) Serial.println(joystick_hor_value);
  if(troubleshoot == 1) Serial.println(joystick_ver_value);
  if(troubleshoot == 1) Serial.println(joystick_thrust_on);

  //assert thrusters to move forward
  analogWrite(left_thruster,  joystick_ver_value);
  analogWrite(right_thruster, joystick_ver_value);

  //assert LEDs
  digitalWrite(left_LED, HIGH);       //assert left LED
  digitalWrite(right_LED,HIGH);       //assert right LED
  }

//*****************************************************************
//*****************************************************************
//Case 2:
//Inputs:
//  X channel <= 1.60 VDC
//  Y channel <= 1.60 VDC
//Output:
//  Move forward and bare left
//  - Which joystick direction is asserted more?
//  - Scale PWM voltage to left and right thruster accordingly
//*****************************************************************

if((joystick_hor_value <= 1.60)&&(joystick_ver_value <= 1.60))
  {
  if(troubleshoot == 1) Serial.println("Zone 2");

  //scale joystick horizontal and vertical to value from 0 to 1
```

```
  joystick_hor_value = 1.60 - joystick_hor_value;
  joystick_ver_value = 1.60 - joystick_ver_value;

  if(troubleshoot == 1)  Serial.println(joystick_hor_value);
  if(troubleshoot == 1)  Serial.println(joystick_ver_value);
  if(troubleshoot == 1)  Serial.println(joystick_thrust_on);

  //assert thrusters and LEDs
  if(joystick_hor_value > joystick_ver_value)
    {
    analogWrite(left_thruster, (joystick_hor_value - joystick_ver_value));
    analogWrite(right_thruster, joystick_hor_value);

    //assert LEDs
    digitalWrite(left_LED,  HIGH);        //assert left LED
    digitalWrite(right_LED,  HIGH);       //assert right LED
    }
  else
    {
    analogWrite(left_thruster, joystick_ver_value);
    analogWrite(right_thruster, (joystick_ver_value - joystick_hor_value));

    //assert LEDs
    digitalWrite(left_LED,  HIGH);        //assert left LED
    digitalWrite(right_LED,  HIGH);       //assert right LED
    }
  }

//************************************************************
//************************************************************
//Case 3:
//Inputs:
//  X channel <= 1.60 VDC
//  Y channel between 1.60 to 1.70 VDC - null zone
//Output:
//  Bare left
//************************************************************

if((joystick_hor_value <= 1.60)&&(joystick_ver_value > 1.60)&&
```

```
    (joystick_ver_value < 1.70))
  {
  if(troubleshoot == 1)  Serial.println("Zone 3");

  //scale joystick vertical to value from 0 to 1
  joystick_hor_value = 1.60 - joystick_hor_value;

  if(troubleshoot == 1)  Serial.println(joystick_hor_value);
  if(troubleshoot == 1)  Serial.println(joystick_ver_value);
  if(troubleshoot == 1)  Serial.println(joystick_thrust_on);

  //assert thrusters
  analogWrite(left_thruster, 0);
  analogWrite(right_thruster, joystick_hor_value);

  //assert LEDs
  digitalWrite(left_LED,  LOW);        //de-assert left LED
  digitalWrite(right_LED,  HIGH);      //assert right LED
  }

//*****************************************************************
//*****************************************************************
//Case 4:
//Inputs:
//  X channel <= 1.60 VDC
//  Y channel >= 1.70 VDC
//Output:
//  Bare left to turn around
//*****************************************************************

if((joystick_hor_value <= 1.60)&&(joystick_ver_value >= 1.70))
  {
  if(troubleshoot == 1) Serial.println("Zone 4");

  //scale joystick horizontal and vertical to value from 0 to 1
  joystick_hor_value = 1.60 - joystick_hor_value;
  joystick_ver_value = joystick_ver_value - 1.70;

  if(troubleshoot == 1)  Serial.println(joystick_hor_value);
```

```
  if(troubleshoot == 1)  Serial.println(joystick_ver_value);
  if(troubleshoot == 1)  Serial.println(joystick_thrust_on);

  //assert thrusters and LEDs
  if(joystick_hor_value > joystick_ver_value)
    {
    analogWrite(left_thruster, 0);
    analogWrite(right_thruster, (joystick_hor_value-joystick_ver_value));

    //assert LEDs
    digitalWrite(left_LED,  LOW);         //de-assert left LED
    digitalWrite(right_LED,  HIGH);       //assert right LED
    }
  else
    {
    analogWrite(left_thruster, 0);
    analogWrite(right_thruster, (joystick_ver_value-joystick_hor_value));

    //assert LEDs
    digitalWrite(left_LED,  LOW);         //de-assert left LED
    digitalWrite(right_LED,  HIGH);       //assert right LED
    }
   }

//*****************************************************************
//*****************************************************************
//Case 5:
//Inputs:
//  X channel between 1.60 to 1.70 VDC - null zone
//  Y channel >= 1.70 VDC
//Output:
//  Move backward - provide same voltage to left and right thrusters
//*****************************************************************

if((joystick_hor_value > 1.60)&&(joystick_hor_value < 1.70)&&
   (joystick_ver_value >= 1.70))
  {
  if(troubleshoot ==1) Serial.println("Zone 5");
```

```
  //scale joystick vertical to value from 0 to 1
  joystick_ver_value = joystick_ver_value - 1.70;

  if(troubleshoot == 1)  Serial.println(joystick_hor_value);
  if(troubleshoot == 1)  Serial.println(joystick_ver_value);
  if(troubleshoot == 1)  Serial.println(joystick_thrust_on);

  //assert thrusters
  analogWrite(left_thruster, 0);
  analogWrite(right_thruster, joystick_ver_value);

  //assert LEDs
  digitalWrite(left_LED,  LOW);         //de-assert left LED
  digitalWrite(right_LED,  HIGH);       //assert right LED
  }

//************************************************************
//************************************************************
//Case 6:
//Inputs:
//  X channel >= 1.70 VDC
//  Y channel >= 1.70 VDC
//Output:
//  Bare left to turn around
//************************************************************

if((joystick_hor_value >= 1.70)&&(joystick_ver_value >= 1.70))
  {
  if(troubleshoot == 1) Serial.println("Zone 6");

  //scale joystick horizontal and vertical to value from 0 to 1
  joystick_hor_value = joystick_hor_value - 1.70;
  joystick_ver_value = joystick_ver_value - 1.70;

  if(troubleshoot == 1)  Serial.println(joystick_hor_value);
  if(troubleshoot == 1)  Serial.println(joystick_ver_value);
  if(troubleshoot == 1)  Serial.println(joystick_thrust_on);

  //assert thrusters and LEDs
```

```
  if(joystick_hor_value > joystick_ver_value)
    {
    analogWrite(left_thruster, (joystick_hor_value-joystick_ver_value));
    analogWrite(right_thruster, 0);

    //assert LEDs
    digitalWrite(left_LED,  HIGH);        //assert left LED
    digitalWrite(right_LED,  LOW);        //de-assert right LED
    }
  else
    {
    analogWrite(left_thruster, (joystick_ver_value-joystick_hor_value));
    analogWrite(right_thruster, 0);

    //assert LEDs
    digitalWrite(left_LED,  HIGH);       //assert left LED
    digitalWrite(right_LED,  LOW);       //de-assert right LED
    }
   }

//*****************************************************************
//*****************************************************************
//Case 7:
//Inputs:
//  X channel >= 1.70 VDC
//  Y channel between 1.60 to 1.70 VDC - null zone
//Output:
//  Bare right
//*****************************************************************

if((joystick_hor_value >= 1.70)&&(joystick_ver_value > 1.60)&&
   (joystick_ver_value < 1.70))
  {
  if(troubleshoot == 1) Serial.println("Zone 7");

  //scale joystick vertical to value from 0 to 1
  joystick_hor_value = joystick_hor_value - 1.70;

  if(troubleshoot == 1)  Serial.println(joystick_hor_value);
```

```
  if(troubleshoot == 1)  Serial.println(joystick_ver_value);
  if(troubleshoot == 1)  Serial.println(joystick_thrust_on);

  //assert thrusters
   analogWrite(left_thruster, joystick_hor_value);
   analogWrite(right_thruster, 0);

  //assert LEDs
  digitalWrite(left_LED,  HIGH);       //assert left LED
  digitalWrite(right_LED,  LOW);       //de-assert right LED
  }

//*************************************************************
//*************************************************************
//Case 8:
//Inputs:
//  X channel >= 1.70 VDC
//  Y channel <= 1.60 VDC
//Output:
//  Move forward and bare right
//  - Which joystick direction is asserted more?
//  - Scale PWM voltage to left and right thruster accordingly
//*************************************************************

if((joystick_hor_value >= 1.70)&&(joystick_ver_value <= 1.60))
  {
  if(troubleshoot == 1) Serial.println("Zone 8");

  //scale joystick horizontal and vertical to value from 0 to 1
  joystick_hor_value = joystick_hor_value - 1.70;
  joystick_ver_value = 1.60 - joystick_ver_value;

  if(troubleshoot == 1)  Serial.println(joystick_hor_value);
  if(troubleshoot == 1)  Serial.println(joystick_ver_value);
  if(troubleshoot == 1)  Serial.println(joystick_thrust_on);

  //assert thrusters and LEDs
  if(joystick_hor_value > joystick_ver_value)
    {
```

```
    analogWrite(left_thruster, joystick_hor_value);
    analogWrite(right_thruster, (joystick_hor_value-joystick_ver_value));

    //assert LEDs
    digitalWrite(left_LED,  HIGH);        //assert left LED
    digitalWrite(right_LED,  HIGH);       //assert right LED
    }
  else
    {
    analogWrite(left_thruster, (joystick_ver_value-joystick_hor_value));
    analogWrite(right_thruster, joystick_ver_value);

    //assert LEDs
    digitalWrite(left_LED,  HIGH);        //assert left LED
    digitalWrite(right_LED,  HIGH);       //assert right LED
    }
  }
}

//************************************************************
```

12.5.8 CONTROL HOUSING LAYOUT

A Plano Model 1312-00 water-resistant field box is used to house the control circuitry and rechargeable battery. The battery is a rechargeable, sealed, lead-acid battery, 12 VDC, with an 8.5 amp-hour capacity. It is available from McMaster-Carr (#7448K82). A battery charger (12 VDC, 4–8 amp-hour rating) is also available (#7448K67). The layout for the ROV control housing is provided in Figure 12.15.

The control circuitry consists of two connected plastic panels, as shown in Figure 12.15. The top panel has the on/off switch, the LED thruster indicators (left, dive, and right), an access hole for the joystick, and a 1/4-in jack for the battery recharge cable.

The lower panel is connected to the top panel using aluminum spacers, screws, and corresponding washers, lock washers, and nuts. The lower panel contains MSP430 equipped with the thumb joystick on the breakout board. The MSP430 LaunchPad is connected to the lower panel using a Jameco stand off kit (#106551). The MSP430 LaunchPad is interfaced to the thrusters via interface circuitry described in Figures 12.11 and 12.12. The interface printed circuit board is connected to the four-conductor thruster cable via a four-conductor Jones connector.

12.5.9 FINAL ASSEMBLY TESTING

The final system is tested a subassembly at a time. The following sequence is suggested.

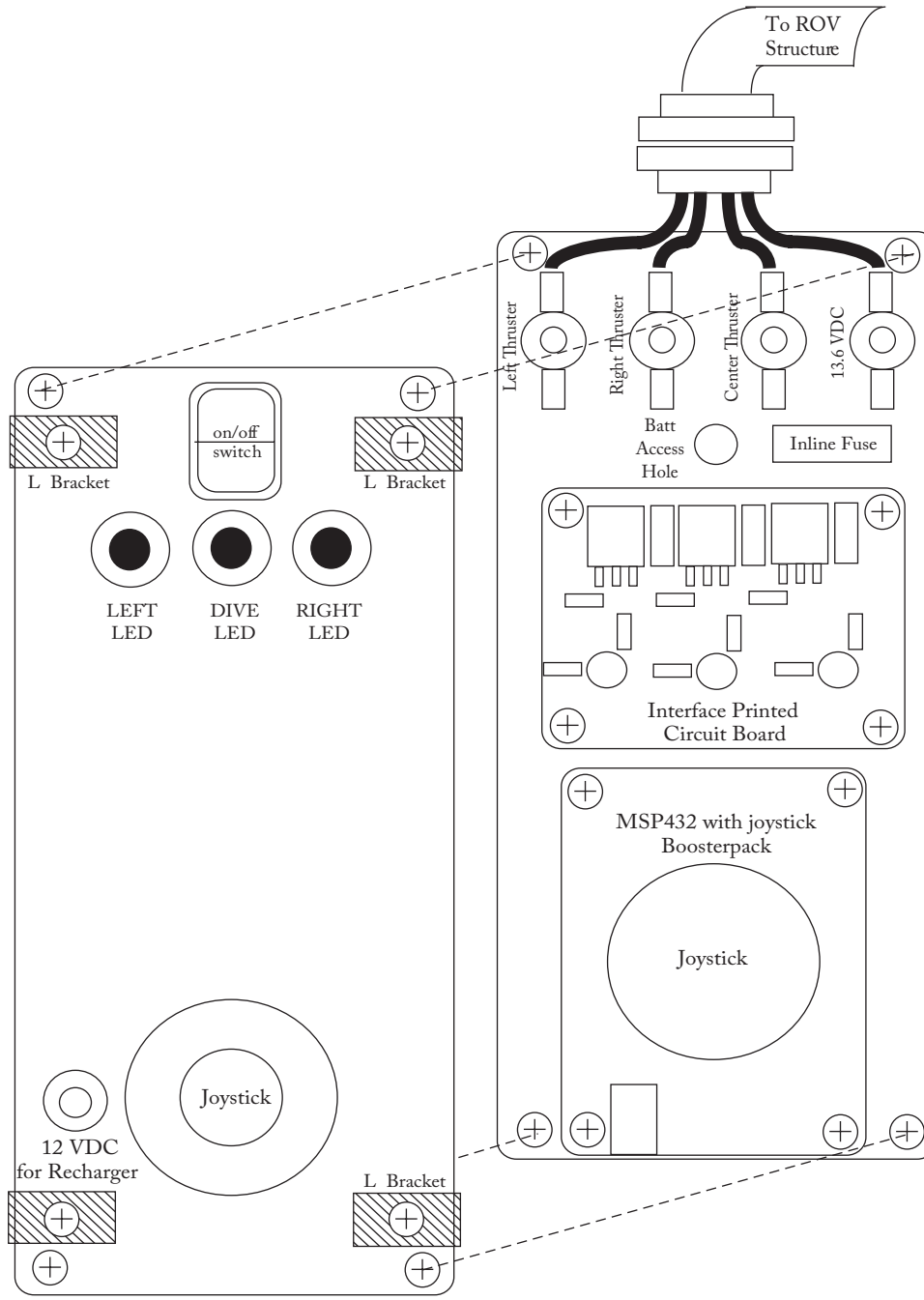


Figure 12.15: ROV control housing layout.

1. Recheck all waterproofed connections. Reapply waterproof caulk as necessary.

2. With the thumb joystick on the breakout board disconnected from MSP430 LaunchPad, test each LED indicator (left, dive, and right). This is accomplished by applying a 3.3 VDC signal in turn to the base resistor of each LED drive transistor.

3. In a similar manner each thruster (left, right, and vertical) may be tested. If available, a signal generator may be used to generate a pulse width modulated signal to test each thruster.

4. With power applied, the voltage regulators aboard the printed circuit board should be tested for proper voltages.

5. The output voltages from the thumb joystick may be verified at the appropriate header pins.

6. With the software fully functional, the thumb joystick on the breakout board may be connected to MSP430 LaunchPad for end-to-end testing.

12.5.10 FINAL ASSEMBLY

The fully assembled ROV is shown in Figure 12.16.

12.5.11 PROJECT EXTENSIONS

The control system provided above has a set of very basic features. Here are some possible extensions for the system.

- Provide a powered dive and surface thruster. To provide for a powered dive and surface capability, the ROV must be equipped with a vertical thruster equipped with an H-bridge to allow for motor forward and reversal. This modification is given as an assignment at the end of the chapter.

- Left and right thruster reverse. Currently, the left and right thrusters may only be powered in one direction. To provide additional maneuverability, the left and right thrusters could be equipped with an H-bridge to allow bi-directional motor control. This modification is given as an assignment at the end of the chapter.

- Proportional speed control with bi-directional motor control. Both of these advanced features may be provided by driving the H-bridge circuit with PWM signals. This modification is given as an assignment at the end of the chapter.

Figure 12.16: ROV fully assembled. (Photo courtesy of J. Barrett, Closer to the Sun International, Inc.)

12.6 MOUNTAIN MAZE NAVIGATING ROBOT

In this project we extend the Dagu Magician maze navigating project described in Chapter 3 to a three-dimensional mountain pass. Also, we use a robot equipped with four motorized wheels. Each of the wheels is equipped with an H-bridge to allow bidirectional motor control. In this example we will only control two wheels. We leave the development of a 4WD robot as an end-of-chapter homework assignment.

12.6.1 DESCRIPTION

For this project, a DF Robot 4WD mobile platform kit was used (DFROBOT ROB0003, Jameco #2124285). The robot kit is equipped with four powered wheels. As in the Dagu Magician project, we equipped the DF Robot with three Sharp GP2Y0A21YKOF IR sensors as shown in Figure 12.17. The robot will be placed in a three-dimensional maze with reflective walls modeled after a mountain pass. The goal of the project is for the robot to detect wall placement and navigate through the maze. The robot will not be provided any information about the maze. The control algorithm for the robot is hosted on MSP430.

12.6.2 REQUIREMENTS

The requirements for this project are simple, the robot must autonomously navigate through the maze without touching maze walls as quickly as possible. Furthermore, the robot must be able to safely navigate through the rugged maze without becoming "stuck" on maze features.

12.6.3 CIRCUIT DIAGRAM

The circuit diagram for the robot is provided in Figure 12.18. The three IR sensors (left, middle, and right) are mounted on the leading edge of the robot to detect maze walls. The sensors' outputs are fed to three separate ADC channels. The robot motors will be driven by PWM channels via an H-bridge. The robot is powered by a 7.5 VDC battery pack (5 AA batteries) which is fed to a 3.3 and 5 VDC voltage regulator. Alternatively, the robot may be powered by a 7.5 VDC power supply rated at several amps. In this case, the power is delivered to the robot by a flexible umbilical cable. The circuit diagram includes the inertial measurement unit (IMU) to measure vehicle tilt and a liquid crystal display. Both were discussed in Chapter 3.

12.6.4 STRUCTURE CHART

The structure chart for the robot project is provided in Figure 12.19.

12.6.5 UML ACTIVITY DIAGRAMS

The UML activity diagram for the robot is provided in Figure 12.20.

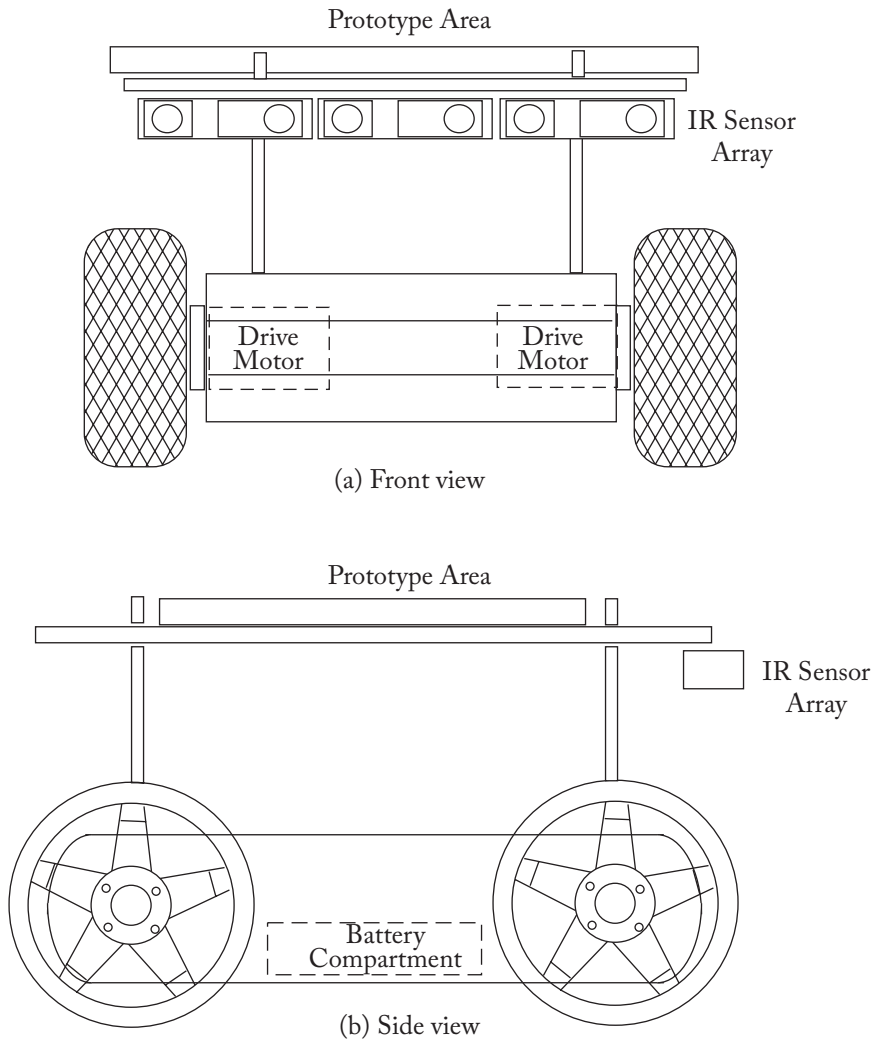


Figure 12.17: Robot layout.

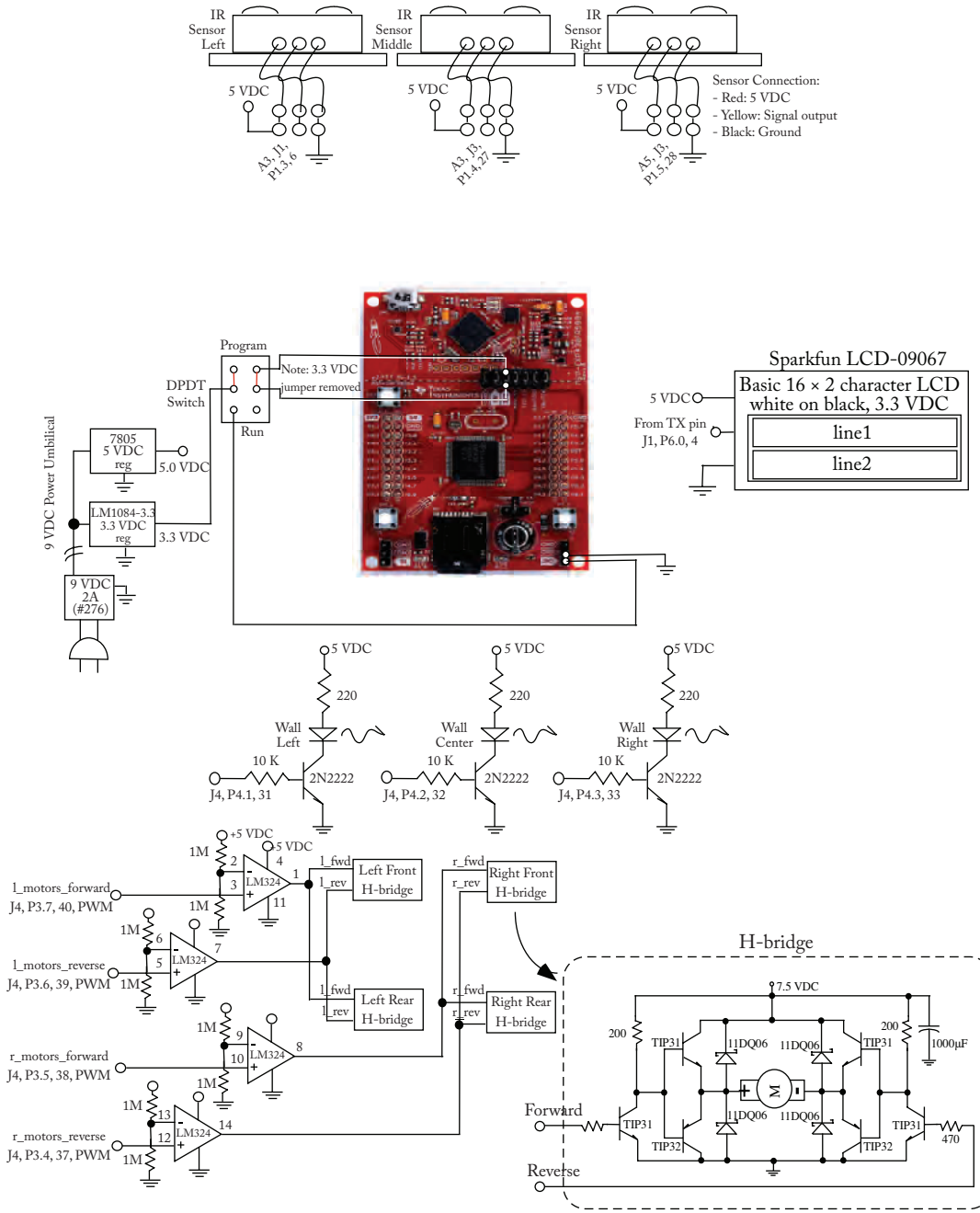


Figure 12.18: Robot circuit diagram. (Illustration used with permission of Texas Instruments (www.ti.com).)

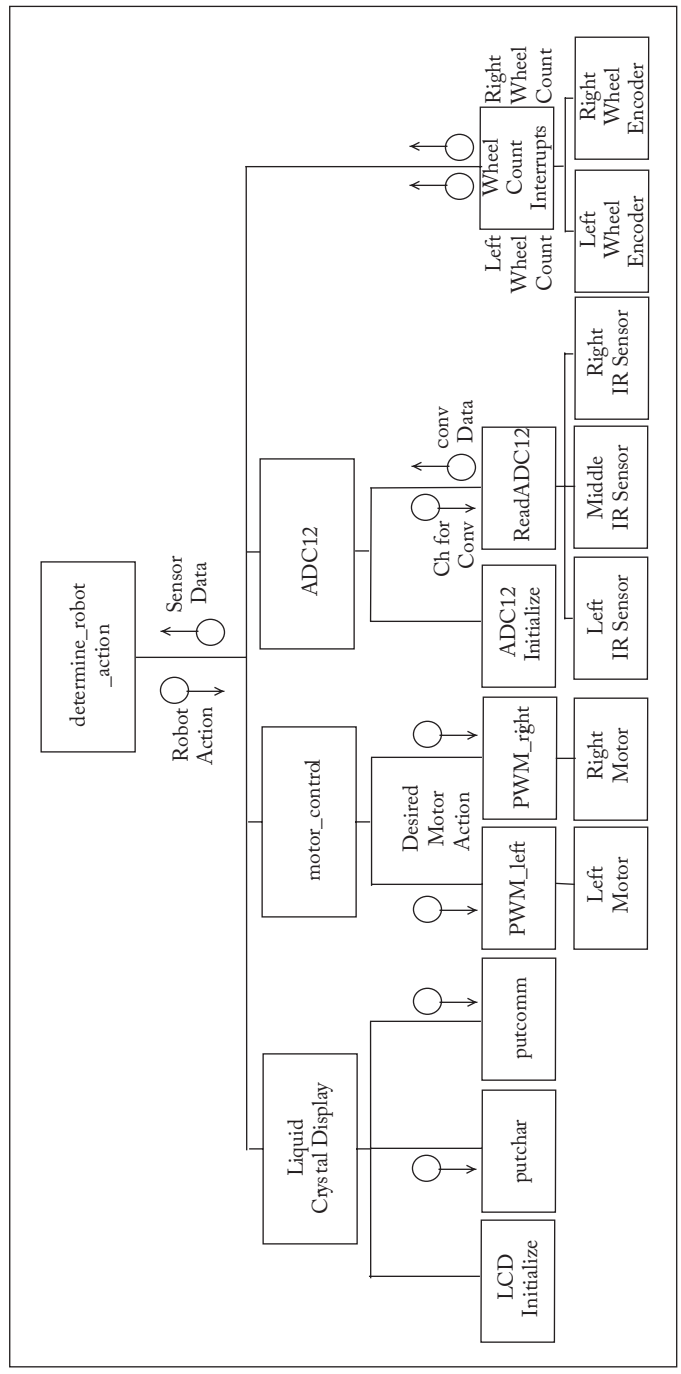


Figure 12.19: Robot structure diagram.

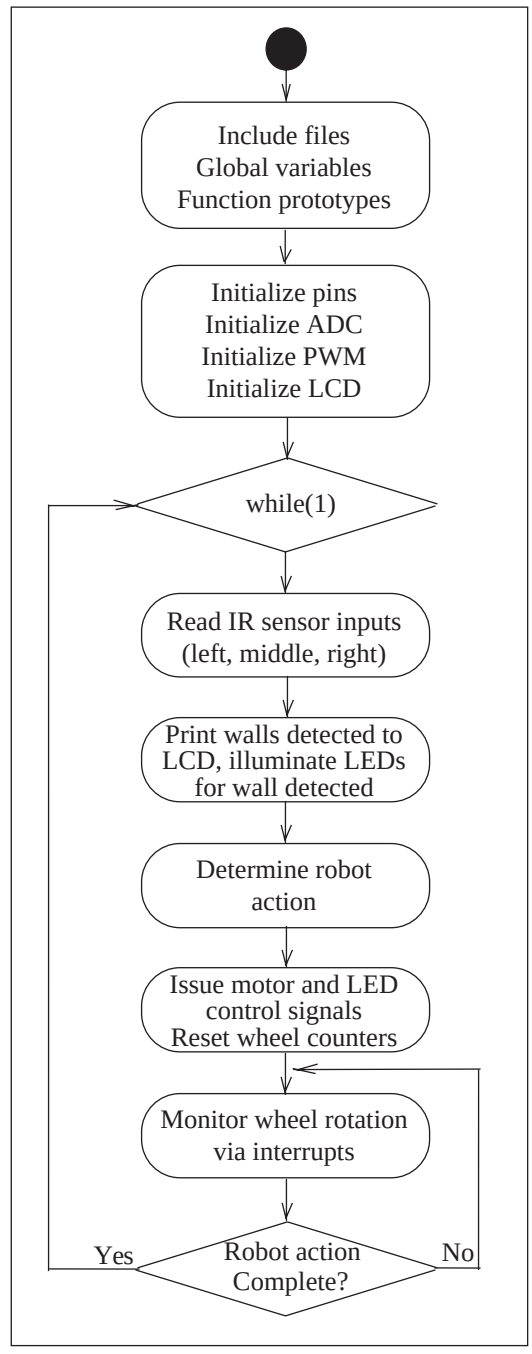


Figure 12.20: Abbreviated robot UML activity diagram. The "determine robot action" consists of multiple decision statements.

12.6.6 ROBOT CODE

The code for the robot may be adapted from that for the Dagu Magician robot. Since the motors are equipped with an H-bridge, slight modifications are required to the robot turning code. These modifications include an additional signal (*forward*/$\overline{reverse}$) for each H-bridge configuration to provide forward and reverse capability. For example, when forward is desired a PWM signal is delivered to one side of the H-bridge and a logic zero to the other side. A level shifter (Texas Instruments PCA9306) is used to adapt the 3.3 VDC signal output from MSP430 LaunchPad to 5.0 VDC levels.

We only provide the basic framework for the code here.

```
//*********************************************************************
//robot
//
//Three IR sensors (left, middle, and right) are mounted on the leading
//edge of the robot to detect maze walls.  The sensors' outputs are
//fed to three separate ADC channels on pins 6, 27, and 28.
//
//The robot is equipped with:
//  - serial LCD at Serial 1 accessible at:
//      - RX: P6.1, pin 3
//      - TX: P6.0, pin 4
//  - LEDs to indicate wall detection: 31, 32, 33
//  - Robot motors are driven by PWM channels via an H-bridge.
//      - the same control signal is sent to left paired motors
//        and the right paired motors.
//      - For example, when forward robot movement is desired,
//        PWM signals are sent to both of the left and right forwards
//        signals and a logic zero signal to the left and right
//        reverse signals.
//      - To render a left robot turn, a PWM signal is sent to the
//        left_reverse control line and a logic zero to the left_forward
//        control line.   Also, a PWM signal is sent to the right_forward
//        control line and a logic zero to the right_reverse
//        control line.
//        The signals are held in this configuration until the wheel
//        encoders indicate the turns have been completed.  The wheel
//        encoders provide ten counts per revolution.
//  - A separate interrupt is used to count left and right wheel counts.
//
//This example code is in the public domain.
```

```
//*********************************************************************

                                    //analog input pins
#define left_IR_sensor     6        //analog pin - left IR sensor
#define center_IR_sensor  27        //analog pin - center IR sensor
#define right_IR_sensor   28        //analog pin - right IR sensor

                                    //digital output pins
                                    //LED indicators - wall detectors
#define wall_left         31        //digital pin - wall_left
#define wall_center       32        //digital pin - wall_center
#define wall_right        33        //digital pin - wall_right

                                    //motor outputs
#define l_motors_forward  40        //digital pin - left motors forward
#define l_motors_reverse  39        //digital pin - left motors reverse
#define r_motors_forward  38        //digital pin - right motors forward
#define r_motors_reverse  37        //digital pin - right motors reverse

                                    //sensor value
int left_IR_sensor_value;           //declare variable for left IR sensor
int center_IR_sensor_value;         //declare variable for center IR sensor
int right_IR_sensor_value;          //declare variable for right IR sensor

int troubleshoot;                   //asserts troubleshoot statements

void setup()
{
troubleshoot = 1;
                                    //enable serial monitor
if(troubleshoot == 1) Serial.begin(9600);

//Initialize serial channel 1 to 9600 BAUD and wait for port to open
//Serial LCD, 3.3 VDC connected to P3.3, pin 4 Sparkfun LCD-09052
Serial1.begin(9600);
delay(1000);                        //allow LCD to boot up

                                    //LED indicators - wall detectors
pinMode(wall_left,   OUTPUT);       //configure pin for digital output
```

```
pinMode(wall_center, OUTPUT);        //configure pin for digital output
pinMode(wall_right,  OUTPUT);        //configure pin for digital output

                                     //motor outputs - PWM
pinMode(l_motors_forward, OUTPUT);   //configure pin for digital output
pinMode(l_motors_reverse, OUTPUT);   //configure pin for digital output
pinMode(r_motors_forward, OUTPUT);   //configure pin for digital output
pinMode(r_motors_reverse, OUTPUT);   //configure pin for digital output
}

void loop()
{
//read analog output from IR sensors
left_IR_sensor_value   = analogRead(left_IR_sensor);
center_IR_sensor_value = analogRead(center_IR_sensor);
right_IR_sensor_value  = analogRead(right_IR_sensor);

//Print sensor values to Serial Monitor
if(troubleshoot == 1)
  {
  Serial.print("Left IR sensor: ");
  Serial.println(left_IR_sensor_value);
  Serial.print("Center IR sensor: ");
  Serial.println(center_IR_sensor_value);
  Serial.print("Right IR sensor: ");
  Serial.println(right_IR_sensor_value);
  Serial.println("");
  }

//to LCD
Serial1.write(254);                  //Command to LCD
delay(5);
Serial1.write(1);                    //Cursor to home position
delay(5);

Serial1.write(254);                  //Command to LCD
delay(5);
Serial1.write(128);                  //Cursor to home position
delay(5);
```

```
Serial1.write("Left  Ctr  Right");
delay(50);
Serial1.write(254);                 //Command to LCD
delay(5);
Serial1.write(192);                 //Cursor to line 2, position 1
delay(5);
Serial1.print(left_IR_sensor_value);
delay(5);
Serial1.write(254);                 //Command to LCD
delay(5);
Serial1.write(198);                 //Cursor to line 2, position 1
delay(5);
Serial1.print(center_IR_sensor_value);
delay(5);
Serial1.write(254);                 //Command to LCD
delay(5);
Serial1.write(203);                 //Cursor to line 2, position 1
delay(5);
Serial1.print(right_IR_sensor_value);
delay(5);


delay(500);

//robot action table row 0 - robot forward
if((left_IR_sensor_value < 300)&&(center_IR_sensor_value < 300)&&
  (right_IR_sensor_value < 300))
  {
                                        //wall detection LEDs
  digitalWrite(wall_left,   LOW);       //turn LED off
  digitalWrite(wall_center, LOW);       //turn LED off
  digitalWrite(wall_right,  LOW);       //turn LED off
                                        //motor control
  analogWrite(l_motors_forward,  64);   //0(off)-255(full speed)
  analogWrite(l_motors_reverse,   0);   //0(off)-255(full speed)
  analogWrite(r_motors_forward,  64);   //0(off)-255(full speed)
  analogWrite(r_motors_reverse,   0);   //0(off)-255(full speed)

  if(troubleshoot == 1) Serial.print("Table row 0 \n\n");
```

```
  }

//robot action table row 1 - robot forward
else if((left_IR_sensor_value < 300)&&(center_IR_sensor_value < 300)&&
       (right_IR_sensor_value > 300))
  {
                                        //wall detection LEDs
  digitalWrite(wall_left,   LOW);       //turn LED off
  digitalWrite(wall_center, LOW);       //turn LED off
  digitalWrite(wall_right,  HIGH);      //turn LED on
                                        //motor control
  analogWrite(l_motors_forward,  64);   //0(off)-255(full speed)
  analogWrite(l_motors_reverse,   0);   //0(off)-255(full speed)
  analogWrite(r_motors_forward,  64);   //0(off)-255(full speed)
  analogWrite(r_motors_reverse,   0);   //0(off)-255(full speed)

  if(troubleshoot == 1) Serial.print("Table row 1 \n\n");
  }

//robot action table row 2 - robot right
else if((left_IR_sensor_value < 300)&&(center_IR_sensor_value > 300)&&
       (right_IR_sensor_value < 300))
  {
                                        //wall detection LEDs
  digitalWrite(wall_left,   LOW);       //turn LED off
  digitalWrite(wall_center, HIGH);      //turn LED on
  digitalWrite(wall_right,  LOW);       //turn LED off
                                        //motor control
  analogWrite(l_motors_forward,  64);   //0(off)-255(full speed)
  analogWrite(l_motors_reverse,   0);   //0(off)-255(full speed)
  analogWrite(r_motors_forward,   0);   //0(off)-255(full speed)
  analogWrite(r_motors_reverse,  64);   //0(off)-255(full speed)

  if(troubleshoot == 1) Serial.print("Table row 2 \n\n");
  }

//robot action table row 3 - robot left
else if((left_IR_sensor_value < 300)&&(center_IR_sensor_value > 300)&&
       (right_IR_sensor_value > 300))
```

```
  {
                                        //wall detection LEDs
  digitalWrite(wall_left,   LOW);       //turn LED off
  digitalWrite(wall_center, HIGH);      //turn LED on
  digitalWrite(wall_right,  HIGH);      //turn LED on
                                        //motor control
  analogWrite(l_motors_forward,    0);  //0(off)-255(full speed)
  analogWrite(l_motors_reverse,  64);   //0(off)-255(full speed)
  analogWrite(r_motors_forward,  64);   //0(off)-255(full speed)
  analogWrite(r_motors_reverse,    0);  //0(off)-255(full speed)

  if(troubleshoot == 1) Serial.print("Table row 3 \n\n");

  }

//robot action table row 4 - robot forward
else if((left_IR_sensor_value > 300)&&(center_IR_sensor_value < 300)&&
       (right_IR_sensor_value < 300))
  {
                                        //wall detection LEDs
  digitalWrite(wall_left,   HIGH);      //turn LED on
  digitalWrite(wall_center, LOW);       //turn LED off
  digitalWrite(wall_right,  LOW);       //turn LED off
                                        //motor control
  analogWrite(l_motors_forward,  64);   //0(off)-255(full speed)
  analogWrite(l_motors_reverse,    0);  //0(off)-255(full speed)
  analogWrite(r_motors_forward,  64);   //0(off)-255(full speed)
  analogWrite(r_motors_reverse,    0);  //0(off)-255(full speed)

  if(troubleshoot == 1) Serial.print("Table row 4 \n\n");

  }

//robot action table row 5 - robot forward
else if((left_IR_sensor_value > 300)&&(center_IR_sensor_value < 300)&&
       (right_IR_sensor_value > 300))
  {
                                        //wall detection LEDs
  digitalWrite(wall_left,   HIGH);      //turn LED on
```

```
  digitalWrite(wall_center, LOW);          //turn LED off
  digitalWrite(wall_right,  HIGH);         //turn LED on
                                           //motor control
  analogWrite(l_motors_forward,  64);      //0(off)-255(full speed)
  analogWrite(l_motors_reverse,    0);     //0(off)-255(full speed)
  analogWrite(r_motors_forward,  64);      //0(off)-255(full speed)
  analogWrite(r_motors_reverse,    0);     //0(off)-255(full speed)

  if(troubleshoot == 1) Serial.print("Table row 5 \n\n");

  }

//robot action table row 6 - robot right
else if((left_IR_sensor_value > 300)&&(center_IR_sensor_value > 300)&&
       (right_IR_sensor_value < 300))
  {
                                           //wall detection LEDs
  digitalWrite(wall_left,   HIGH);         //turn LED on
  digitalWrite(wall_center, HIGH);         //turn LED on
  digitalWrite(wall_right,  LOW);          //turn LED off
                                           //motor control
  analogWrite(l_motors_forward,  64);      //0(off)-255(full speed)
  analogWrite(l_motors_reverse,    0);     //0(off)-255(full speed)
  analogWrite(r_motors_forward,    0);     //0(off)-255(full speed)
  analogWrite(r_motors_reverse,  64);      //0(off)-255(full speed)

  if(troubleshoot == 1) Serial.print("Table row 6 \n\n");
  }

//robot action table row 7 - robot reverse
else if((left_IR_sensor_value > 300)&&(center_IR_sensor_value > 300)&&
       (right_IR_sensor_value > 300))
  {
                                           //wall detection LEDs
  digitalWrite(wall_left,   HIGH);         //turn LED on
  digitalWrite(wall_center, HIGH);         //turn LED on
  digitalWrite(wall_right,  HIGH);         //turn LED on
                                           //motor control
  analogWrite(l_motors_forward,  64);      //0(off)-255(full speed)
```

```
  analogWrite(l_motors_reverse,    0);    //0(off)-255(full speed)
  analogWrite(r_motors_forward,    0);    //0(off)-255(full speed)
  analogWrite(r_motors_reverse,  64);     //0(off)-255(full speed)

  if(troubleshoot == 1) Serial.print("Table row 7 \n\n");

  }
}

//*********************************************************************
```

12.6.7 MOUNTAIN MAZE

The mountain maze was constructed from plywood, chicken wire, expandable foam, plaster cloth, and Bondo. A rough sketch of the desired maze path was first constructed. Care was taken to insure the pass was wide enough to accommodate the robot. The maze platform was constructed from 3/8-in plywood on 2-by-4-in framing material. Maze walls were also constructed from the plywood and supported with steel L brackets.

With the basic structure complete, the maze walls were covered with chicken wire. The chicken wire was secured to the plywood with staples. The chicken wire was then covered with plaster cloth (Creative Mark Artist Products #15006). To provide additional stability, expandable foam was sprayed under the chicken wire (Guardian Energy Technologies, Inc. Foam It Green 12). The mountain scene was then covered with a layer of Bondo for additional structural stability. Bondo is a two-part putty that hardens into a strong resin. Mountain pass construction steps are illustrated in Figure 12.21. The robot is shown in the maze in Figure 12.22

12.6.8 PROJECT EXTENSIONS

- Modify the turning commands such that the PWM duty cycle and the length of time the motors are on are sent in as variables to the function.
- Develop a function for reversing the robot.
- Equip the motor with another IR sensor that looks down toward the maze floor for "land mines." A land mine consists of a paper strip placed in the maze floor that obstructs a portion of the maze. If a land mine is detected, the robot must deactivate the maze by moving slowly back and forth for 3 s and flashing a large LED.
- The current design is a two-wheel, front-wheel drive system. Modify the design for a two-wheel, rear-wheel drive system.
- The current design is a two-wheel, front-wheel drive system. Modify the design for a 4WD system.

Figure 12.21: Mountain maze.

Figure 12.22: Robot in maze. (Photo courtesy of J. Barrett, Closer to the Sun International, Inc.).

- Develop a 4WD system which includes a tilt sensor. The robot should increase motor RPM (duty cycle) for positive inclines and reduce motor RPM (duty cycle) for negatives inclines.

- Equip the robot with an analog inertial measurement unit (IMU) to measure vehicle tilt. Use the information provided by the IMU to optimize robot speed going up and down steep grades.

12.7 SUMMARY

In this chapter, we discussed the design process, related tools, and applied the process to a real-world design. It is essential to follow a systematic, disciplined approach to embedded systems design to successfully develop a prototype that meets established requirements.

12.8 REFERENCES AND FURTHER READING

Anderson, M. Help wanted: Embedded engineers why the United States is losing its edge in embedded systems. *IEEE—USA Today's Engineer*, February 2008. 488

Barrett, S. F. and Pack, D. J. *Atmel AVR Processor Primer Programming and Interfacing*, Morgan & Claypool Publishers, 2008. www.morganclaypool.com DOI: 10.2200/s00100ed1v01y200712dcs015.

Barrett, S. F. and Pack, D. J. *Embedded Systems Design and Applications with the 68HC12 and HCS12*, Pearson Prentice Hall, Upper Saddle River, NJ, 2005.

Barrett, S. F. and Pack, D. J. *Embedded Systems Design with the Atmel Microcontroller*, Morgan & Claypool Publishers, 2010. DOI: 10.2200/s00225ed1v01y200910dcs025.

Barrett, S. F. and Pack, D. J. *Microcontrollers Fundamentals for Engineers and Scientists*, Morgan & Claypool Publishers, 2006. www.morganclaypool.com DOI: 10.2200/s00025ed1v01y200605dcs001. 491

Bohm, H. and Jensen, V. *Build your Own Underwater Robot and Other Wet Projects*, 11th ed., Westcoast Words, Vancouver, BC, Canada, 2012. 506

Christ, R. and Wernli, R. Sr. *The ROV Manual—A User Guide for Remotely Operated Vehicle*, 2nd ed., Oxford, UK Butterworth–Heinemann imprint of Elsevier, 2014.

Dale, N. and Lilly, S. C. *Pascal Plus Data Structures*, 4th ed., Jones and Bartlett, Englewood Cliffs, NJ, 1995. 491

Fowler, M. with K. Scott. *UML Distilled A Brief Guide to the Standard Object Modeling Language*, 2nd ed., Addison–Wesley, Boston, MA, 2000. 491, 492

Seaperch. www.seaperch.com 502

Texas Instruments MSP430FR2433 Mixed-Signal Microcontroller, (SLASE59D), Texas Instruments, Revised 2018.

Texas Instruments MSP430FR4xx and MSP430FR2xx Family User's Guide, (SLAU445G), Texas Instruments, 2016.

Texas Instruments MSP430FR58xx, MSP430FR59xx, and MSP430FR6xx Family, (SLAU367O), Texas Instruments, 2017.

Texas Instruments MSP430FR599x, MSP430FR596x Mixed-Signal Microcontrollers, (SLASE54C), Texas Instruments, Revised 2018.

12.9 CHAPTER EXERCISES

1. What is an embedded system?
2. What aspects must be considered in the design of an embedded system?
3. What is the purpose of the structure chart, UML activity diagram, and circuit diagram?
4. Why is a system design only as good as the test plan that supports it?
5. During the testing process, when an error is found and corrected, what should now be accomplished?
6. Discuss the top-down design, bottom-up implementation concept.
7. Describe the value of accurate documentation.
8. What is required to fully document an embedded systems design?
9. For the Dagu Magician robot, modify the PWM turning commands such that the PWM duty cycle and the length of time the motors are on are sent in as variables to the function.
10. For the Dagu Magician robot, equip the motor with another IR sensor that looks down for "land mines." A land mine consists of a paper strip placed in the maze floor that obstructs a portion of the maze. If a land mine is detected, the robot must deactivate it by rotating about its center axis three times and flashing a large LED while rotating.
11. For the Dagu Magician robot, develop a function for reversing the robot.
12. Provide a powered dive and surface thruster for the SeaPerch ROV. To provide for a powered dive and surface capability, the ROV must be equipped with a vertical thruster equipped with an H-bridge to allow for motor forward and reversal.
13. Provide a left and right thruster reverse for the SeaPerch ROV. Currently, the left and right thrusters may only be powered in one direction. To provide additional maneuverability, the left and right thrusters could be equipped with an H-bridge to allow bi-directional motor control.
14. Provide proportional speed control with bi-directional motor control for the SeaPerch ROV. Both of these advanced features may be provided by driving the H-bridge circuit with PWM signals.
15. For the 4WD robot, modify the PWM turning commands such that the PWM duty cycle and the length of time the motors are on are sent in as variables to the function.

16. For the 4WD robot, equip the motor with another IR sensor that looks down for "land mines." A land mine consists of a paper strip placed in the maze floor that obstructs a portion of the maze. If a land mine is detected, the robot must deactivate it by rotating about its center axis three times and flashing a large LED while rotating.
17. For the 4WD robot, develop a function for reversing the robot.
18. For the 4WD robot, the current design is a two-wheel, front-wheel drive system. Modify the design for a two-wheel, rear-wheel drive system.
19. For the 4WD robot, the current design is a two wheel, front wheel drive system. Modify the design for a 4WD system.
20. For the 4WD robot, develop a 4WD system which includes a tilt sensor. The robot should increase motor RPM (duty cycle) for positive inclines and reduce motor RPM (duty cycle) for negatives inclines.
21. Equip the robot with an inertial measurement unit (IMU) to measure vehicle tilt. Use the information provided by the IMU to optimize robot speed going up and down steep grades.
22. Develop an embedded system controlled dirigible/blimp (`www.microflight.com`, `www.rctoys.com`).
23. Develop a trip odometer for your bicycle (Hint: use a Hall Effect sensor to detect tire rotation).
24. Develop a timing system for a four lane Pinewood Derby track.
25. Develop a playing board and control system for your favorite game (Yahtzee, Connect Four, Battleship, etc.).
26. You have a very enthusiastic dog that loves to chase balls. Develop a system to launch balls for the dog.
27. Construct the UML activity diagrams for all functions related to the weather station.
28. It is desired to updated weather parameters every 15 min. Write a function to provide a 15 min delay.
29. Add one of the following sensors to the weather station:
    - anemometer
    - barometer
    - hygrometer

- rain gauge
- thermocouple

You will need to investigate background information on the selected sensor, develop an interface circuit for the sensor, and modify the weather station code.

30. Modify the weather station software to also employ the 138 x 110 LCD. Display pertinent weather data on the display.
31. Voice output (Hint: Use an ISD 4003 Chip Corder.)
32. Develop an embedded system controlled submarine (www.seaperch.org).
33. Equip the MSP430 with automatic cell phone dialing capability to notify you when a fire is present in your home.

Authors' Biographies

STEVEN F. BARRETT

Steven F. Barrett, Ph.D., P.E., received a B.S. in Electronic Engineering Technology from the University of Nebraska at Omaha in 1979, an M.E.E.E. from the University of Idaho at Moscow in 1986, and a Ph.D. from The University of Texas at Austin in 1993. He was formally an active duty faculty member at the United States Air Force Academy, Colorado and is now the Associate Dean of Academic Programs at the University of Wyoming. He is a member of IEEE (senior) and Tau Beta Pi (chief faculty advisor). His research interests include digital and analog image processing, computer–assisted laser surgery, and embedded controller systems. He is a registered Professional Engineer in Wyoming and Colorado. He co-wrote with Dr. Daniel Pack several textbooks on microcontrollers and embedded systems. In 2004, Barrett was named "Wyoming Professor of the Year" by the Carnegie Foundation for the Advancement of Teaching and in 2008 was the recipient of the National Society of Professional Engineers (NSPE) in Higher Education, Engineering Education Excellence Award.

DANIEL J. PACK

Daniel J. Pack, Ph.D., P.E., is the Dean of the College of Engineering and Computer Science at the University of Tennessee, Chattanooga (UTC). Prior to joining UTC, he was Professor and Mary Lou Clarke Endowed Department Chair of the Electrical and Computer Engineering Department at the University of Texas, San Antonio (UTSA). Before his service at UTSA, Dr. Pack was Professor (now Professor Emeritus) of Electrical and Computer Engineering at the United States Air Force Academy (USAFA), CO, where he served as founding Director of the Academy Center for Unmanned Aircraft Systems Research. He received a B.S. in Electrical Engineering, an M.S. in Engineering Sciences, and a Ph.D. in Electrical Engineering from Arizona State University, Harvard University, and Purdue University, respectively. He was a visiting scholar at the Massachusetts Institute of Technology-Lincoln Laboratory. Dr. Pack has co-authored seven textbooks on embedded systems (including *68HC12 Microcontroller: Theory and Applications* and *Embedded Systems: Design and Applications with the 68HC12 and HCS12*) and published over 160 book chapters, technical journal/transactions, and conference papers on unmanned systems, cooperative control, robotics, pattern recognition, and engineering education. He is the recipient of a number of teaching and research awards including Carnegie U.S. Professor of the Year Award, Frank J. Seiler Research Excellence Award, Tau Beta Pi Outstanding Professor Award, Academy Educator Award, and Magoon Award. He is a member of Eta

Kappa Nu (Electrical Engineering Honorary), Tau Beta Pi (Engineering Honorary), IEEE, and the American Society of Engineering Education. He is a registered Professional Engineer in Colorado, serves as Associate Editor of *IEEE Systems Journal*, and is a member on a number of executive advisory or editorial boards including the *Journal of Intelligent & Robotic Systems*, *International Journal of Advanced Robotic Systems*, and *SimCenter Enterprise*. His research interests include unmanned aerial vehicles, intelligent control, automatic target recognition, robotics, and engineering education. E-mail: daniel-pack@utc.edu

Index