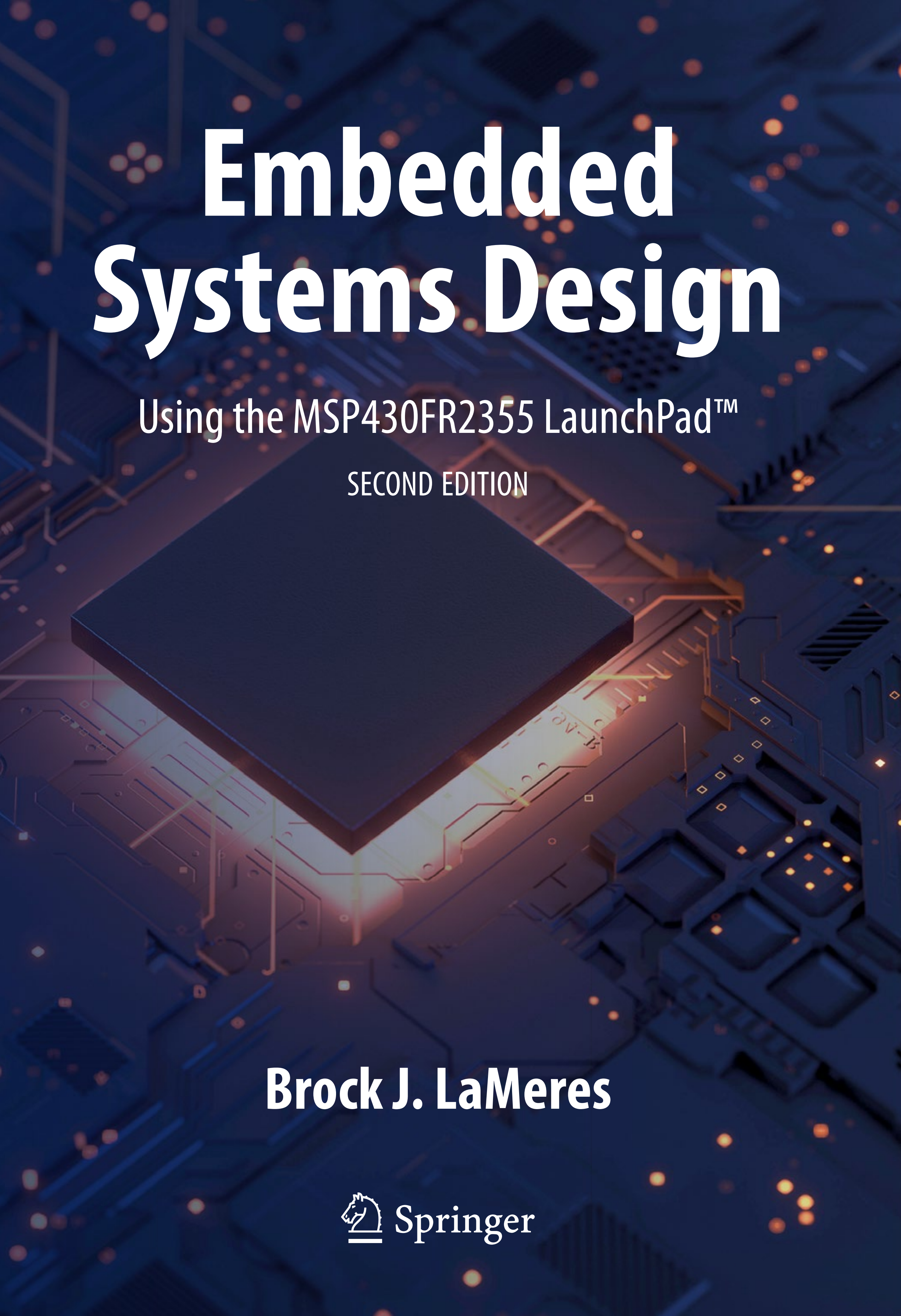
Embedded Systems Design

Using the MSP430FR2355 LaunchPad™

SECOND EDITION



Brock J. LaMeres

Springer

Embedded Systems Design using the MSP430FR2355 LaunchPad™

Embedded Systems Design using the MSP430FR2355 LaunchPad™

2nd Edition

Brock J. LaMeres

Springer

Brock J. LaMeres
Department of Electrical and Computer Engineering
Montana State University
Bozeman, MT, USA

ISBN 978-3-031-20887-4 ISBN 978-3-031-20888-1 (eBook)
https://doi.org/10.1007/978-3-031-20888-1

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

Why Another Book on Embedded Systems?

Embedded computers represent one of the most pervasive technologies of our time. When most people think of computers, they think of their laptops and workstations, or even the servers that are the backbone of the Internet; however, when one begins to contemplate how many technologies around them use small, inexpensive computers embedded as their "brains," one begins to more fully understand the magnitude of how many embedded computers exist in our society. If you look around any room you will see technology that is so commonplace, you may not even realize that most of it is run by an internal computer. Technologies such as appliances, thermostats, handheld devices, projectors, televisions, personal assistants, and radios all have small computers within them that control their operation. As one starts to notice all of the embedded computers around them, they will quickly notice that there are orders of magnitude more embedded computers than the standard general-purpose computers that run Windows® or iOS™. As these technologies become *smarter* and are Wi-Fi connected, we move into era called the *Internet of Things*. This next technological paradigm shift will turn all of the embedded computers into a collaborative network in an attempt to make our lives better by automating taking mundane tasks.

Simultaneous to the exponential increase in embedded computers in our society is the need for engineers and software developers that can build and program these systems. Due to the high popularity of embedded computers, there is also a wide range of embedded computer platforms. As such, the textbooks that exist to aid in the education of embedded computers are somewhat fragmented. Textbooks for embedded computers aren't able to be written at only the theoretical level such as circuits and electronics textbooks. Instead, they must identify a computer platform and focus the content on that specific embedded computer technology. This leads to a large number of embedded systems books to support the large number of embedded platforms. This also leads to books becoming obsolete much faster than traditional engineering textbooks.

One reason for a new book in this area is creating comprehensive content around a widely popular embedded computer, the Texas Instruments MSP430. The MSP430 is a modern computer platform that is mature in the market and has a great deal of technical and educational support behind it. This means that this platform will be relevant and supported for at least the next 10–15 years.

A second reason for a new book in this area is to provide content the way people actual learn (i.e., by doing). Current embedded systems books, even those targeting the MSP430, tend to only provide small code segments that can't be directly run on a target platform. This new book approaches content delivery around the model that the reader is sitting at a computer with an MSP430 plugged in and running each of the examples as they move through the book. This *learn-a-little*, *do-a-little* is a proven pedagogical strategy. It also supports both active learning within the classroom and self-regulated learning for individual readers.

A third reason for a new book in this area is to provide a seamless delivery of both assembly language and C language programming of the MSP430. Most embedded systems books are structured in one of three ways: (1) they only cover assembly; (2) they only cover C; (3) they attempt to cover both but don't provide sufficient details of either. This new book will begin in assembly language as a means to show the lower-level operation of the computer hardware. It will then move into C to implement more complex systems that require abstracting the lower-level hardware. With the design of the book consisting of examples in both languages that can be directly coded and run, there will not be a void between the examples and real implementation of the concepts as seen in other books.

The three guiding principles for the design of this book are:

Learn by example This book is designed to teach the material the way it is learned, through example. Every concept that is covered in this book is supported by numerous programming examples that provide the reader with a step-by-step explanation for how and why the computer is doing what it is doing.

Learn by doing This book targets the Texas Instruments MSP430 microcontroller. This platform is a widely popular, low-cost embedded system that is used to illustrate each concept in the book. The book is designed for a reader who is at their computer with the MSP430 plugged in so that each example can be coded and run as they learn.

Build a foundational understanding first, then move into abstraction This book teaches the basic operation of an embedded computer using assembly language first so that the computer operation can be explored at a lower level. Once complicated systems are introduced (i.e., serial communication and analog-to-digital converters), the book moves into the C programming language. Moving to C allows the learner to abstract the operation of the lower-level hardware and focus on understanding how to "make things work." By spending time in assembly to understand the underlying hardware, the transition to C can happen more rapidly while still leaving the reader with a foundational understanding of the underlying hardware.

If this learning model is followed, the reader will come away with much more than knowledge of facts and figures; they will have a skillset and experience with embedded systems that will be both marketable and of key importance to our society.

This second edition adds two highly requested chapters: Chaps. 16 and 17. Both chapters provide example programs on how to configure these systems and observe their operation.

How to Use This Book

This book is most effective when the reader has the *Texas Instruments Inc. MSP430FR2355 LaunchPad™ Development Kit* and codes along with the material. All examples can be directly run on the LaunchPad™ board. All programs were created and compiled using the *Texas Instruments Inc. Code Composer Studio (CCS)*. This development environment is free from Texas Instruments and can be run on multiple operating systems. Each of the examples noted with a keyboard are ones that are intended to be coded in CCS and run on the LaunchPad™ board.

There are three supporting documents that should also be downloaded from Texas Instruments to provide background for the material in this book. The first is the *MSP430FR4xx and MSP430FR2xx Family User's Guide*[1]. This document gives the general concept of operation for the MSP430 architecture that is used on the specific microcontroller targeted in this book. Throughout this book, this document is referred to as the "MSP430 User's Guide." The second document that should be retrieved is the *MSP430FR235x, MSP430FR215x Mixed-Signal Microcontrollers Device-Specific Data Sheet*[2]. This second document provides the specific details of the MSP430FR2355 microcontroller device that is used in each example. Throughout this book, this document is referred to as the" Device-Specific Data Sheet." The final document that should be retrieved is the *MSP430FR235x LaunchPad™ Development Kit User's Guide*[3]. This document describes the circuitry implemented on the LaunchPad™ board. This document is critical to understanding the types of input/output and programming capability available on the LaunchPad™ board.

Additional Resources

Supporting videos will be posted to the author's YouTube channel, which can be found at https://www.youtube.com/c/DigitalLogicProgramming_LaMeres. This channel contains many other videos beyond embedded systems to help people learn digital logic, basic programming, and robotics. Subscribe to this channel in order to be notified when new videos for this book are posted.

Bozeman, MT, USA

Brock J. LaMeres

Acknowledgments

Endless thanks to my family for support of this project. JoAnn, Alexis, and Kylie: You are my rocks and rockstars.

A huge thank you to all the Montana State University engineering students that helped with the development of this book. They proofed, made suggestions, helped create examples and labs, created slides, and even drew out explanations for the solutions manual. Go Bobcats!

Contents

Chapter 1: Introduction to Embedded Systems

This chapter introduces the general concept of an embedded system. The goal is to provide a high-level understanding of what an embedded computer is, their role in modern society, and motivation to learn more about them.

Learning Outcomes—After completing this chapter, you will be able to:

1.1 Describe the basic concept of an embedded system.

1.1 What Is an Embedded System?

When most people hear the term "computer" they immediately think of their laptops or PCs. More recently, people have come to also associate portable devices such as smart phones and tablets with the term computer; however, most people don't tend to think of all of the modern electronics surrounding them as computers. But in fact, the majority of the consumer electronics in our daily lives have a computer that is controlling their operation. Items such as household appliances, thermostats, the mirror controller in a vehicle, an office copy machine, and even a hearing aid are examples of electronics in our daily lives that all have embedded computers at their core. The term to describe the small computers within these electronic devices is an *embedded computer*. The entire system then becomes an *embedded system*, implying that a small computer is included with the system and controls its operation [4–6].

So, what is the difference between the computers that we think of in laptops and workstations and those embedded in the majority of consumer electronics? We will classify computers into two broad categories: general-purpose and embedded. Let's start with a general-purpose computer. This type of computer is designed to run any type of software that the user desires. A user is able to install, uninstall, and update software applications that meet the user's current need. To support the abundance of potential software applications that may be run, general-purpose computers have abundant resources at their disposal. It is common for a general-purpose computer to have a variety of peripherals such as displays, keyboards/keypads, mice, Internet connections, and wireless communication capability. General-purpose computers also typically contain relatively high-performance hardware components such as fast processors and large amounts of data and program storage. To manage the large amounts of resources in a general-purpose computer, these devices require an *operating system* (OS). An operating system controls all of the resources and allocates them to the various software applications that are running. Common operating systems that you may be familiar with are Windows, iOS, and Linux. All of these operating systems have the same purpose, to manage the hardware resources of the computer so that any arbitrary software program may be successfully run on it. General-purpose computers also are designed for heavy user interaction. A typical general-purpose computer is designed to support a user sitting in front of it interacting with the software program using the keyboard, mouse, and monitor.

Despite a general-purpose computer having abundant resources, the vast majority of the time these resources are not being used. Activities such as typing an email or browsing the web use a very little amount of the computer's resources. But even though any single software program may only use a small amount of the resources available, a general-purpose computer must contain everything that is potentially needed to support a future program that may require the resources. To provide all of this capability, general-purpose computers are relatively expensive compared to their embedded systems'

B. J. LaMeres, *Embedded Systems Design using the MSP430FR2355 LaunchPad™*,
https://doi.org/10.1007/978-3-031-20888-1_1

counterparts with laptops and workstations costing hundreds to thousands of dollars. Additionally, to support the large number of resources, general-purpose computers are typically implemented using a group of integrated circuits (ICs), or chips. A general-purpose computer typically has one chip that implements its central processing unit (CPU), other chips to implement the data memory (i.e., RAM), and others to implement the program storage (i.e., hard drive or solid-state drive). A general-purpose computer is considered a *distributed* architecture because the full functionality of the computer is spread across multiple IC chips. This type of architecture supports very sophisticated CPUs and large amounts of data and program memory.

Now let's consider the concept of an embedded computer. Many of the electronics that are used in our daily lives don't require a high-performance computer that runs Windows or iOS. Your coffee maker doesn't require a cutting-edge processor to make your coffee. Your thermostat doesn't need the ability to run Microsoft Word to control the temperature of the room. What these types of applications need is a computer that can respond to inputs from the outside world (i.e., button presses or sensor inputs) and the ability to send control signals to other sub-systems to accomplish a task (i.e., heat the water reservoir or turn on the furnace). In these types of applications, the computer needs *just enough* resources to get the job done. The small computers used for these dedicated applications are *embedded computers*. Embedded computers are also referred to as *microcontrollers*, or MCUs, because they are the primary controller for their dedicated application.

Embedded computers have a variety of traits that differentiate them from general-purpose computers. First, embedded computers are mostly implemented on a single IC. An embedded computer does not require the large amounts of RAM and program storage as in a general-purpose computer, so all of the components of the computer can be put onto a single IC. Embedded computers also are able to include a variety of common peripherals such as timers, analog-to-digital converters, digital-to-analog converters, and serial interfaces to make them as versatile as possible when it comes to controlling all types of electronic devices. Depending on the amount of resources included, the physical size of an embedded computer chip can be extremely small. Consider the size of a thumb drive. This small storage stick has a full embedded computer system on it. Now compare that to the physical size of the hardware that resides in a general-purpose laptop. The small size of an embedded computer chip gives it the ability to be used in an endless list of applications.

Another attribute of an embedded computer is that it isn't designed to run arbitrary software programs that are installed at the will of the user. An embedded computer in a toaster isn't designed to have a new program installed by the user to make it control a microwave. The software that an embedded system runs is called *firmware* to highlight that it is not intended to be changed frequently. While some embedded systems can have their software updated by downloading a new program, this is a highly infrequent occurrence in the life of the computer (how many times have you upgraded the firmware in your car's mirrors?). Since an embedded computer does not need to support general-purpose software programs, its software can be designed to optimize the functionality for the application at hand. This means there is a tight coupling between the hardware and software components of the embedded computer. This provides an optimized design that is typically only suited for the application that the embedded computer is made for. Embedded systems can contain operating systems; however, these OSs aren't anything like Windows or iOS. Instead, these operating systems act as task schedulers coordinating real-time activities such as reading from buttons or sensors and performing response actions. Operating systems for embedded computers are called *real-time operating systems* (RTOS), reflecting their purpose as a task scheduler. Not all embedded computers run an RTOS, but many just simply run dedicated software to accomplish a specific task.

Another attribute of an MCU is their low cost. While a general-purpose microprocessor chip can cost tens to hundreds of dollars, an MCU can cost 10s of cents to a few dollars. This low cost is driven by the high-volume manufacturing process used to create MCUs. As semiconductor manufacturing volume goes up, the cost of the individual chips produced goes down. The low cost of an embedded computer chip reveals the prevalence and popularity of the embedded computer. The number of embedded computers sold globally is multiple orders of magnitude larger than general-purpose computers, and this gap is only expected to grow as our devices become smarter and interconnected and more functionality is embedded into automobiles. In 2018, over 25 billion MCUs were sold globally [1]. Compare this to the ~400 million personal computers (PCs) sold annually that are based on general-purpose CPUs [2]. In fact, it is estimated that only ~2% of computer chips produced end up in PCs. To put this in perspective, every year there are tens of billions of new embedded systems being created to improve the quality of our lives [7,8].

Smart phones and tablets are a technology that has moved from the embedded systems category into the general-purpose description over the past decade. When cellular phones and personal desktop assistances first emerged, they were very much embedded systems. These devices were not able to support arbitrary software being installed on them and were designed to perform very specific tasks (phone calls, text messaging, scheduling, etc.); however, as processing technology advanced, cellular phones began to have the power to support more general-purpose operating systems such as Android and iOS. This gave them the ability to support different applications (i.e., *apps*) that could access the phone or tablets' abundant resources. Today, smart phones and tablets are considered general-purpose computers and only different from laptops and workstations in their size, portability, and software support.

From an educational perspective, general-purpose computers and embedded computers are effectively the same. They are both computers that consist of hardware resources that are programmed to accomplish tasks using software. We can learn about the architecture of a computer using either, but focusing on embedded computers has a few advantages. First, they are inexpensive enough that every reader of this book can have an embedded computer sitting next to them where they can gain experience programming the MCU to do a variety of tasks. This allows embedded systems education to move from simply reading facts and figures to an active learning process where a deeper understanding of computers can be obtained. An additional bonus of this approach is that after completing the exercises in this book, the reader will have a practical skill set that is highly sought after in industry. Figure 1.1 shows a graphical depiction of the applications of computers and categorizing them into either general-purpose or embedded.

Concept Check

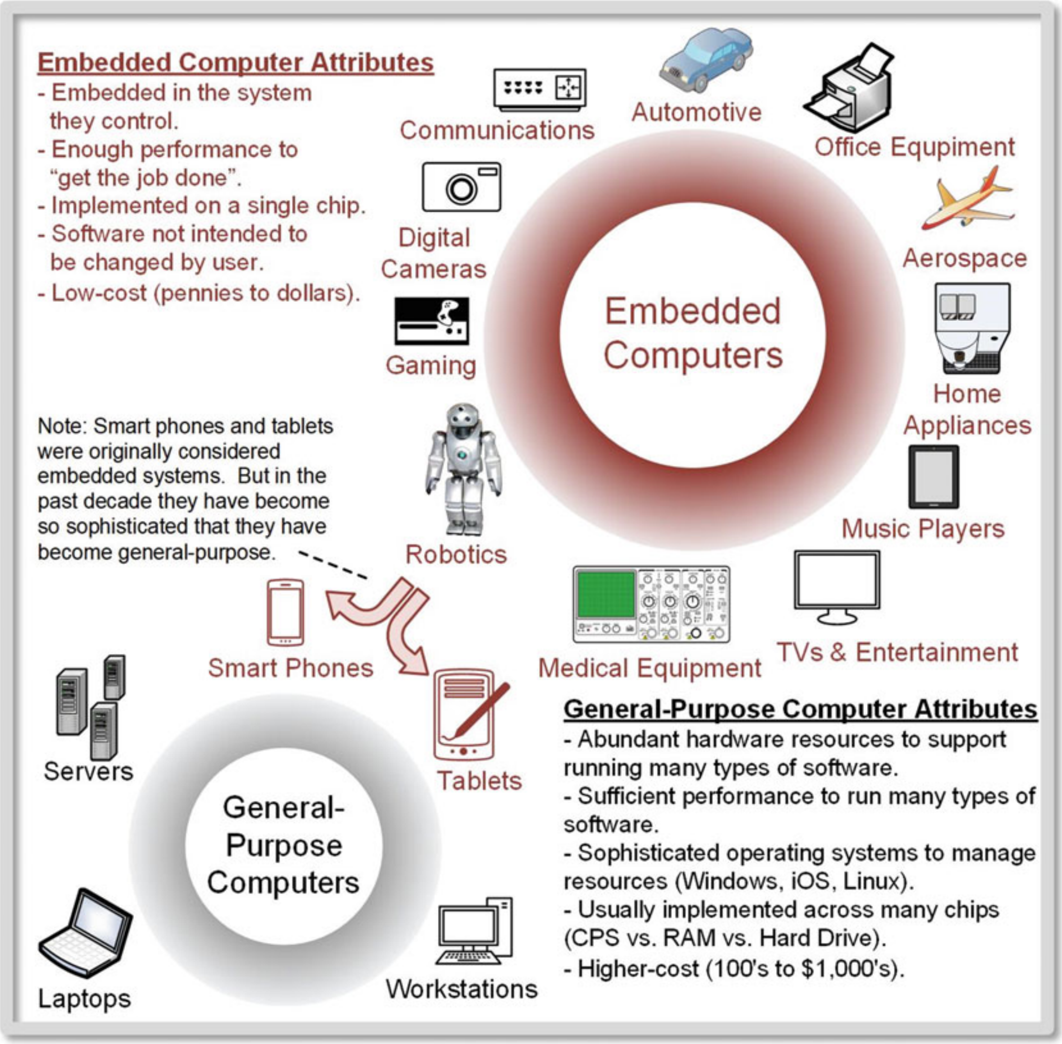
CC1.1 Why don't we just make one computer chip that serves the needs of every application on Earth? It seems like that would be a lot simpler than having thousands of different computer products.

A) That isn't practical because every application has a different set of requirements. If we tried to create a "super chip" that could be simultaneously used in high-end servers and microwave ovens, then our microwave ovens would cost tens of thousands of dollars and consume much more electrical power.

B) It is illegal to just have one computer chip design. It goes against the idea of capitalism and competition in the marketplace.

C) They are still trying to create one computer chip that meets every application. It is just really hard.

D) The answer to this concept check is A.



Embedded Computer Attributes

- Embedded in the system they control.
- Enough performance to "get the job done".
- Implemented on a single chip.
- Software not intended to be changed by user.
- Low-cost (pennies to dollars).

Note: Smart phones and tablets were originally considered embedded systems. But in the past decade they have become so sophisticated that they have become general-purpose.

General-Purpose Computer Attributes

- Abundant hardware resources to support running many types of software.
- Sufficient performance to run many types of software.
- Sophisticated operating systems to manage resources (Windows, iOS, Linux).
- Usually implemented across many chips (CPS vs. RAM vs. Hard Drive).
- Higher-cost (100's to $1,000's).

Fig. 1.1
Applications of embedded system [9]

Summary

- Computers can be put into two broad categories: general-purpose and embedded.
- General-purpose computers are designed to run any arbitrary software program. Thus, they contain abundant hardware resources to support potential applications and a sophisticated operating system to manage the resources.
- Embedded systems have a small computer embedded within them that controls the operation of the device.
- Embedded computers have enough resources to "get the job done." They are implemented on a single IC, are very low in cost, and have a tight coupling between their hardware and software that optimizes their operation for the application they are designed for.
- The number of embedded computer chips sold each year outnumbers general-purpose computer chips by multiple orders of magnitude.

Exercise Problems

Section 1.1: What Is an Embedded System?

1.1.1 Classify a computer with the following attributes into either "general-purpose" or "embedded": *runs a sophisticated operating system such as Windows*.

1.1.2 Classify a computer with the following attributes into either "general-purpose" or "embedded": *implemented on a single chip*.

1.1.3 Classify a computer with the following attributes into either "general-purpose" or "embedded": *can cost less than a US dollar*.

1.1.4 Classify a computer with the following attributes into either "general-purpose" or "embedded": *designed to run any arbitrary software application the user desires*.

1.1.5 Classify a computer with the following attributes into either "general-purpose" or "embedded": *can cost hundreds to thousands of US dollars*.

1.1.6 Classify a computer with the following attributes into either "general-purpose" or "embedded": *its software is called firmware to highlight that it is rarely changed after being deployed*.

1.1.7 Classify a computer with the following attributes into either "general-purpose" or "embedded": *also called a microcontroller because it controls the sub-systems around it*.

1.1.8 Classify a computer with the following attributes into either "general-purpose" or "embedded": *can have a real-time operating system, but the OS primarily acts as a task scheduler*.

1.1.9 Classify a computer with the following attributes into either "general-purpose" or "embedded": *has a tight coupling between the hardware and software, which optimizes its operation for the application at hand*.

1.1.10 Classify a computer with the following attributes into either "general-purpose" or "embedded": *is the most popular type of computer on Earth*.

1.1.11 For the following application, would a general-purpose computer or an embedded computer be better suited: *running the Windows operating system*.

1.1.12 For the following application, would a general-purpose computer or an embedded computer be better suited: *controlling an Xbox*.

1.1.13 For the following application, would a general-purpose computer or an embedded computer be better suited: *running a washing machine*.

1.1.14 For the following application, would a general-purpose computer or an embedded computer be better suited: *controlling the radio in a car*.

1.1.15 For the following application, would a general-purpose computer or an embedded computer be better suited: *controlling a satellite*.

1.1.16 For the following application, would a general-purpose computer or an embedded computer be better suited: *controlling a wireless router*.

1.1.17 For the following application, would a general-purpose computer or an embedded computer be better suited: *running servers in a data farm based on the Windows platform*.

1.1.18 For the following application, would a general-purpose computer or an embedded computer be better suited: *a laptop running iOS*.

1.1.19 For the following application, would a general-purpose computer or an embedded computer be better suited: *a Linux workstation used to develop for an MCU*.

1.1.20 For the following application, would a general-purpose computer or an embedded computer be better suited: *controlling a video doorbell*.

Chapter 2: Digital Logic Basics

This chapter provides a brief summary of classical digital logic design [10]. The point of this chapter is to expose the reader to the main concepts of digital logic so that they are familiar with them when moving into the computer organization topics covered later in the book. It is not intended to be an in-depth coverage of digital logic design. Rather, it is intended to expose the reader to the terminology used in digital logic, the most common number systems used in embedded systems, the basic logic operations used in combinational logic, and the basic operation of sequential logic, including synchronous registers and finite state machines.

Learning Outcomes—After completing this chapter you will be able to:

2.1 Describe the formation and use of positional number systems including conversions between bases and basic arithmetic.
2.2 Describe the basic operation of combinational logic circuits.
2.3 Describe the basic operation of sequential logic circuits.
2.4 Describe the basic operation of semiconductor memory including different implementation technologies.

2.1 Number Systems

Logic circuits are used to generate and transmit 1s and 0s to compute and convey information. This two-valued number system is called *binary*. As presented earlier, there are many advantages of using a binary system; however, the human brain has been taught to count, label, and measure using the *decimal* number system. The decimal number system contains ten unique symbols (0 → 9). In order to bridge the gap between the way our brains think (decimal) and how we build our computers (binary), we need to understand the basics of number systems. This includes the formal definition of a positional number system and how it can be extended to accommodate any arbitrarily large (or small) value. This also includes how to convert between different number systems that contain different numbers of symbols. In this section, we cover three different number systems: decimal (ten symbols), binary (two symbols), and hexadecimal (16 symbols). The study of decimal and binary is obvious as they represent how our brains interpret the physical world (decimal) and how our computers work (binary). Hexadecimal is studied because it is a useful means to represent large sets of binary values using a manageable number of symbols. This section will also discuss how to perform basic arithmetic (addition and subtraction) in the binary number system and how to represent negative numbers. The goal of this section is to provide an understanding of the basic principles of number systems so that we can design computer programs that use arbitrary number bases.

2.1.1 Positional Number Systems

A positional number system allows the expansion of the original set of symbols so that they can be used to represent any arbitrarily large (or small) value. For example, if we use the ten symbols in our decimal system, we can count from 0 to 9. Using just the individual symbols we do not have enough symbols to count beyond 9. To overcome this, we use the same set of symbols but assign a different value to the symbol based on its position within the number. The *position* of the symbol with respect to other symbols in the number allows an individual symbol to represent greater (or lesser) values. We can use this approach to represent numbers larger than the original set of symbols. For example, let's say we

B. J. LaMeres, *Embedded Systems Design using the MSP430FR2355 LaunchPad™*,
https://doi.org/10.1007/978-3-031-20888-1_2

want to count from 0 upward by 1. We begin counting 0, 1, 2, 3, 4, 5, 6, 7, 8 to 9. When we are out of symbols and wish to go higher, we bring on a symbol in a different position with that position being valued higher and then start counting over with our original symbols (e.g., . . ., 9, 10, 11,. . . 19, 20, 21,. . .). This is repeated each time a position runs out of symbols (e.g., . . ., 99, 100, 101. . . 999, 1000, 1001,. . .).

First, let’s look at the formation of a number system. The first thing that is needed is a set of symbols. The formal term for one of the symbols in a number system is a *numeral***.** One or more numerals are used to form a *number*. We define the number of numerals in the system using the terms *radix* or *base*. For example, our decimal number system is said to be *base 10*, or have a *radix of 10*, because it consists of ten unique numerals or symbols.

Radix = Base≡the number of numerals in thenumber system

The next thing that is needed is the relative value of each numeral with respect to the other numerals in the set. We can say $0 < 1 < 2 < 3$, etc., to define the relative magnitudes of the numerals in this set. The numerals are defined to be greater or less than their neighbors by a magnitude of 1. For example, in the decimal number system each of the subsequent numerals is greater than its predecessor by exactly 1. When we define this relative magnitude, we are defining that the numeral 1 is greater than the numeral 0 by a magnitude of 1; the numeral 2 is greater than the numeral 1 by a magnitude of 1; etc. At this point we have the ability to count from 0 to 9 by 1s. We also have the basic structure for mathematical operations that have results that fall within the numeral set from 0 to 9 (e.g., $1 + 2 = 3$). In order to expand the values that these numerals can represent, we need define the rules of a positional number system.

2.1.1.1 Generic Structure

In order to represent larger or smaller numbers than the lone numerals in a number system can represent, we adopt a positional system. In a positional number system, the relative position of the numeral within the overall number dictates its value. When we begin talking about the position of a numeral, we need to define a location to which all of the numerals are positioned with respect to. We define the *radix point* as the point within a number to which numerals to the left represent whole numbers and numerals to the right represent fractional numbers. The radix point is denoted with a period (i.e., “.”). A particular number system often renames this radix point to reflect its base. For example, in the base 10 number system (i.e., decimal), the radix point is commonly called the *decimal point*; however, the term *radix point* can be used across all number systems as a generic term. If the radix point is not present in a number, it is assumed to be to the right of number. Figure 2.1 shows an example number highlighting the radix point and the relative positions of the whole and fractional numerals.

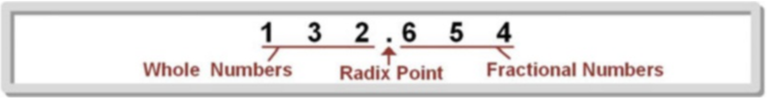


Fig. 2.1
Definition of radix point

Next, we need to define the position of each numeral with respect to the radix point. The position of the numeral is assigned a whole number with the number to the left of the radix point having a position value of 0. The position number increases by 1 as numerals are added to the left (2, 3, 4...) and decreased by 1 as numerals are added to the right (−1, −2, −3). We will use the variable *p* to represent position. The position number will be used to calculate the value of each numeral in the number based on its relative position to the radix point. Figure 2.2 shows the example number with the position value of each numeral highlighted.

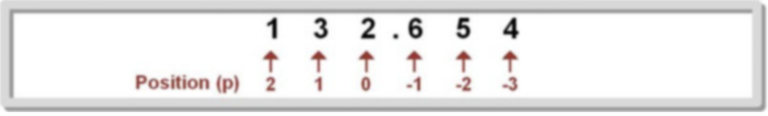


Fig. 2.2
Definition of position number (p) within the number

In order to create a generalized format of a number, we assign the term *digit* (d) to each of the numerals in the number. The term digit signifies that the numeral has a position. The position of the digit within the number is denoted as a subscript. The term *digit* can be used as a generic term to describe a numeral across all systems, although some number systems will use a unique term instead of digit which indicates its base. For example, the binary system uses the term *bit* instead of digit; however, using the term digit to describe a generic numeral in any system is still acceptable. Figure 2.3 shows the generic subscript notation used to describe the position of each digit in the number.

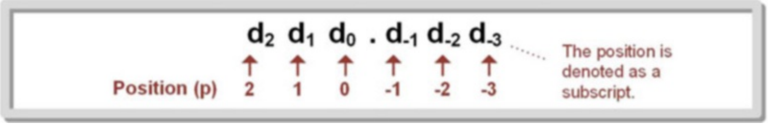

d_2 d_1 d_0 . d_{-1} d_{-2} d_{-3}
Position (p) 2 1 0 -1 -2 -3
The position is denoted as a subscript.

Fig. 2.3
Digit notation

We write a number from left to right starting with the highest position digit that is greater than 0 and end with the lowest position digit that is greater than 0. This reduces the number of numerals that are written; however, a number can be represented with an arbitrary number of 0s to the left of the highest position digit greater than 0 and an arbitrary number of 0s to the right of the lowest position digit greater than 0 without affecting the value of the number. For example, the number 132.654 could be written as 0132.6540 without affecting the value of the number. The 0s to the left of the number are called *leading 0s* and the 0s to the right of the number are called *trailing 0s*. The reason this is being stated is because when a number is implemented in circuitry, the number of numerals is fixed, and each numeral must have a value. The variable *n* is used to represent the number of numerals in a number. If a number is defined with n=4, that means four numerals are always used. The number 0 would be represented as 0000 with both representations having an equal value.

2.1.1.2 Decimal Number System (Base 10)

As mentioned earlier, the decimal number system contains ten unique numerals (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9). This system is thus a base 10 or a radix 10 system. The relative magnitudes of the symbols are 0 < 1 < 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9.

2.1.1.3 Binary Number System (Base 2)

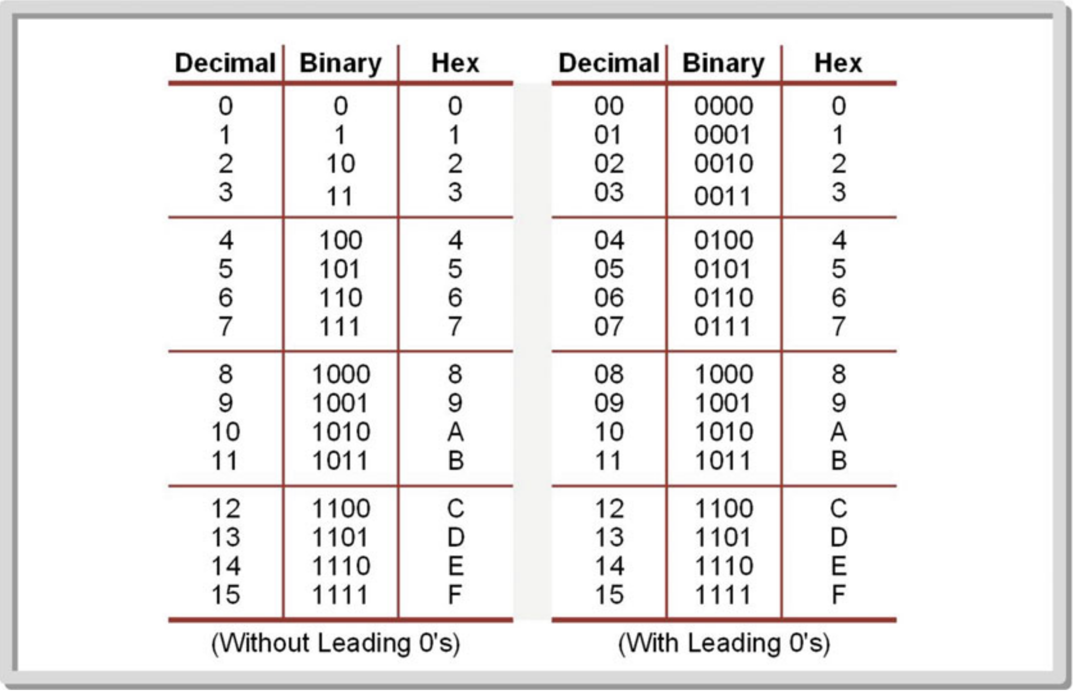
The binary number system contains two unique numerals (0 and 1). This system is thus a base 2 or a radix 2 system. The relative magnitudes of the symbols are 0 < 1. At first glance, this system looks very limited in its ability to represent large numbers due to the small number of numerals. When counting up, as soon as you count from 0 to 1, you are out of symbols and must increment the $p + 1$ position in order to represent the next number (e.g., 0, 1, 10, 11, 100, 101, . . .); however, magnitudes of each position scale quickly so that circuits with a reasonable amount of digits can represent very large numbers. The term *bit* is used instead of *digit* in this system to describe the individual numerals and at the same time indicate the base of the number.

Due to the need for multiple bits to represent meaningful information, there are terms dedicated to describing the number of bits in a group. When 4 bits are grouped together, they are called a *nibble*. When 8 bits are grouped together, they are called a *byte*. Larger groupings of bits are called *words***.** The size of the word can be stated as either an *n-bit word* or omitted if the size of the word is inherently implied. For example, if you were using a 32-bit microprocessor, using the term *word* would be interpreted as a *32-bit word.* For example, if there was a 32-bit grouping, it would be referred to as a 32-bit word. The leftmost bit in a binary number is called the *Most Significant Bit* (MSB). The rightmost bit in a binary number is called the *Least Significant Bit (LSB)*.

2.1.1.4 Hexadecimal Number System (Base 16)

The hexadecimal number system contains 16 unique numerals. This system is most often referred to in spoken word as “hex” for short. Since we only have ten Arabic numerals in our familiar decimal system, we need to use other symbols to represent the remaining six numerals. We use the alphabetic characters A–F in order to expand the system to 16 numerals. The 16 numerals in the hexadecimal system are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. The relative magnitudes of the symbols are 0 < 1 < 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < A < B < C < D < E < F. We use the generic term digit to describe the numerals within a hexadecimal number.

At this point, it becomes necessary to indicate the base of a written number. The number 10 has an entirely different value if it is a decimal number or binary number. In order to handle this, a subscript is typically included at the end of the number to denote its base when writing out the number. For example, 10_{10} indicates that this number is decimal “ten.” If the number was written as 10_2, this number would represent binary “one zero.” Table 2.1 lists the equivalent values in each of the four number systems just described for counts from 0_{10} to 15_{10}. The left side of the table does not include leading 0s. The right side of the table contains the same information but includes the leading 0s. The equivalencies of decimal, binary, and hexadecimal in this table are typically committed to memory.



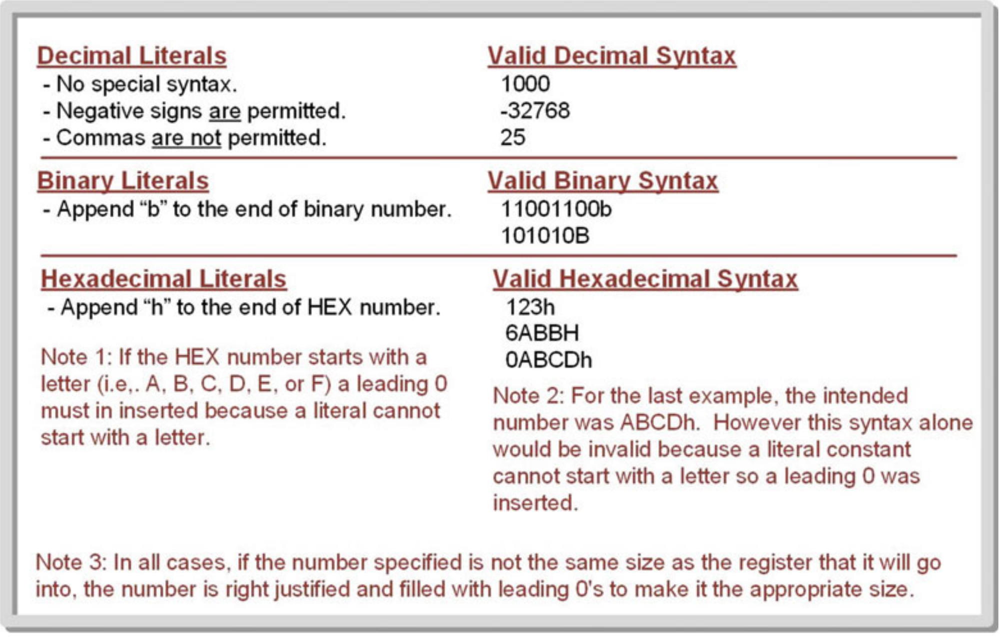
Decimal	Binary	Hex	Decimal	Binary	Hex
0	0	0	00	0000	0
1	1	1	01	0001	1
2	10	2	02	0010	2
3	11	3	03	0011	3
4	100	4	04	0100	4
5	101	5	05	0101	5
6	110	6	06	0110	6
7	111	7	07	0111	7
8	1000	8	08	1000	8
9	1001	9	09	1001	9
10	1010	A	10	1010	A
11	1011	B	11	1011	B
12	1100	C	12	1100	C
13	1101	D	13	1101	D
14	1110	E	14	1110	E
15	1111	F	15	1111	F

(Without Leading 0's)

(With Leading 0's)

Table 2.1
Number system equivalency (decimal, binary, hexadecimal)

When typing out different number bases within the MSP430 low-level programming environment, special syntax is used to specify the desired base. Figure 2.4 shows the allowable syntax for specifying constant *literals* for the MSP430. The term "literal" simply means that the number is to be treated as a number, not as something else such as a location in memory.



Decimal Literals
- No special syntax.
- Negative signs are permitted.
- Commas are not permitted.

Valid Decimal Syntax
1000
-32768
25

Binary Literals
- Append "b" to the end of binary number.

Valid Binary Syntax
11001100b
101010B

Hexadecimal Literals
- Append "h" to the end of HEX number.

Note 1: If the HEX number starts with a letter (i.e,. A, B, C, D, E, or F) a leading 0 must in inserted because a literal cannot start with a letter.

Valid Hexadecimal Syntax
123h
6ABBH
0ABCDh

Note 2: For the last example, the intended number was ABCDh. However this syntax alone would be invalid because a literal constant cannot start with a letter so a leading 0 was inserted.

Note 3: In all cases, if the number specified is not the same size as the register that it will go into, the number is right justified and filled with leading 0's to make it the appropriate size.

Fig. 2.4
Specifying different number bases in the MSP430 low-level programming environment

CONCEPT CHECK

CC2.1.1 The base of a number system is arbitrary and is commonly selected to match a particular aspect of the physical system in which it is used (e.g., base 10 corresponds to our 10 fingers, base 2 corresponds to the 2 states of a switch). If a physical system contained 3 unique modes and a base of 3 was chosen for the number system, what is the base 3 equivalent of the decimal number 3?

A) $3_{10} = 11_3$

B) $3_{10} = 3_3$

C) $3_{10} = 10_3$

D) $3_{10} = 21_3$

2.1.2 Base Conversion

Now we look at converting between bases. There are distinct techniques for converting to and from decimal. There are also techniques for converting between bases that are powers of 2 (e.g., base 2, 4, 8, and 16).

2.1.2.1 Converting to Decimal

The value of each digit within a number is based on the individual digit value and the digit's position. Each position in the number contains a different *weight* based on its relative location to the radix point. The weight of each position is based on the radix of the number system that is being used. The weight of each position in decimal is defined as:

$$\textbf{Weight} = (\textbf{Radix})^{\textbf{p}}$$

This expression gives the number system the ability to represent fractional numbers since an expression with a negative exponent (e.g., x^{-y}) is evaluated as 1 over the expression with the exponent change to positive (e.g., $1/x^y$). Figure 2.5 shows the generic structure of a number with its positional weight highlighted.

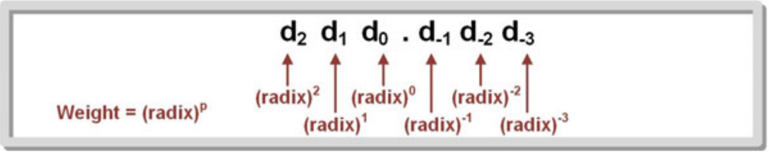


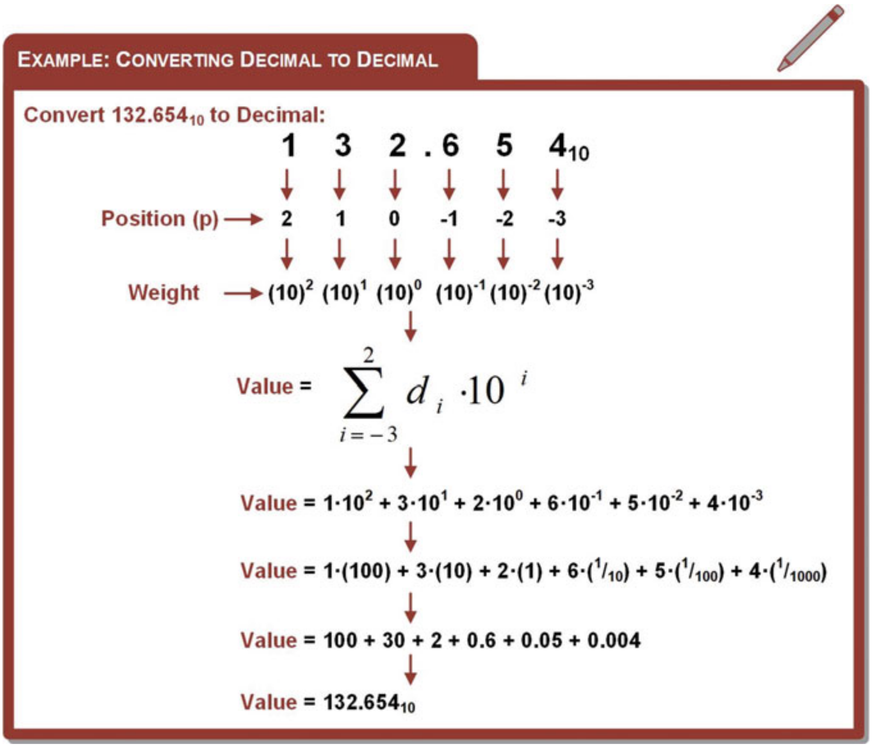
Fig. 2.5
Weight definition

In order to find the decimal value of each of the numerals in the number, its individual numeral value is multiplied by its positional weight. In order to find the value of the entire number, each value of the individual numeral-weight products is summed. The generalized format of this conversion is written as:

$$\textbf{Total Decimal Value} = \sum_{i=p_{\min}}^{p_{\max}} d_i \cdot (\text{radix})^i$$

In this expression, $p_{\max}$ represents the highest position number that contains a numeral greater than 0. The variable $p_{\min}$ represents the lowest position number that contains a numeral greater than 0. These limits are used to simplify the hand calculations; however, these terms theoretically could be $+\infty$ to $-\infty$ with no effect on the result since the summation of every leading 0 and every trailing 0 contributes nothing to the result.

As an example, let's evaluate this expression for a decimal number. The result will yield the original number but will illustrate how positional weight is used. Let's take the number 132.654_{10}. To find the decimal value of this number, each numeral is multiplied by its positional weight and then all of the products are summed. The positional weight for the digit 1 is $(\text{radix})^p$ or $(10)^2$. In decimal this is called the hundred's position. The positional weight for the digit 3 is $(10)^1$, referred to as the ten's position. The positional weight for digit 2 is $(10)^0$, referred to as the 1s position. The positional weight for digit 6 is $(10)^{-1}$, referred to as the tenth's position. The positional weight for digit 5 is $(10)^{-2}$, referred to as the hundredth's position. The positional weight for digit 4 is $(10)^{-3}$, referred to as the thousandth's position. When these weights are multiplied by their respective digits and summed, the result is the original decimal number 132.654_{10}. Example 2.1 shows this process step by step.



EXAMPLE: CONVERTING DECIMAL TO DECIMAL

Convert 132.654_{10} to Decimal:

$$1\quad 3\quad 2\,.\,6\quad 5\quad 4_{10}$$

Position (p) → 2 1 0 -1 -2 -3

Weight → $(10)^2$ $(10)^1$ $(10)^0$ $(10)^{-1}$ $(10)^{-2}$ $(10)^{-3}$

$$\text{Value} = \sum_{i=-3}^{2} d_i \cdot 10^i$$

$$\text{Value} = 1{\cdot}10^2 + 3{\cdot}10^1 + 2{\cdot}10^0 + 6{\cdot}10^{-1} + 5{\cdot}10^{-2} + 4{\cdot}10^{-3}$$

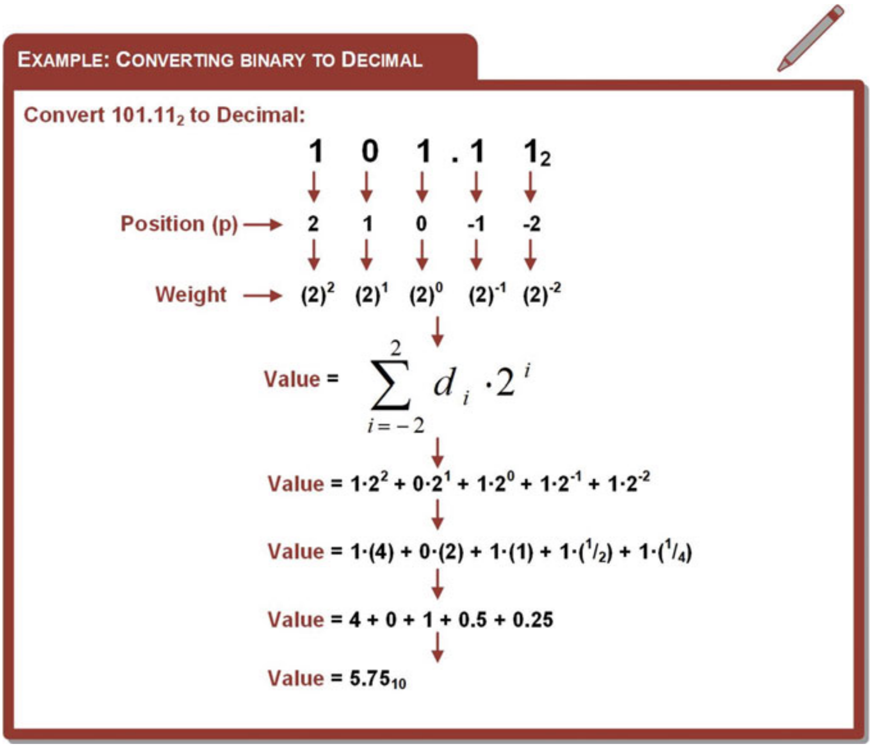
$$\text{Value} = 1{\cdot}(100) + 3{\cdot}(10) + 2{\cdot}(1) + 6{\cdot}({}^1/_{10}) + 5{\cdot}({}^1/_{100}) + 4{\cdot}({}^1/_{1000})$$

$$\text{Value} = 100 + 30 + 2 + 0.6 + 0.05 + 0.004$$

$$\text{Value} = 132.654_{10}$$

Example 2.1
Converting decimal to decimal

This process is used to convert between any other base to decimal. Let's convert 101.11_2 to decimal. The same process is followed with the exception that the base in the summation is changed to 2. Converting from binary to decimal can be accomplished quickly in your head due to the fact that the bit values in the products are either 1 or 0. That means any bit that is a 0 has no impact on the outcome and any bit that is a 1 simply yields the weight of its position. Example 2.2 shows the step-by-step process converting a binary number to decimal.



EXAMPLE: CONVERTING BINARY TO DECIMAL

Convert 101.11_2 to Decimal:

$$\begin{array}{rccccc} & 1 & 0 & 1\,. & 1 & 1_2 \\ & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ \text{Position (p)} \longrightarrow & 2 & 1 & 0 & -1 & -2 \\ & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ \text{Weight} \longrightarrow & (2)^2 & (2)^1 & (2)^0 & (2)^{-1} & (2)^{-2} \end{array}$$

$$\text{Value} = \sum_{i=-2}^{2} d_i \cdot 2^i$$

$$\text{Value} = 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 + 1\cdot 2^{-1} + 1\cdot 2^{-2}$$

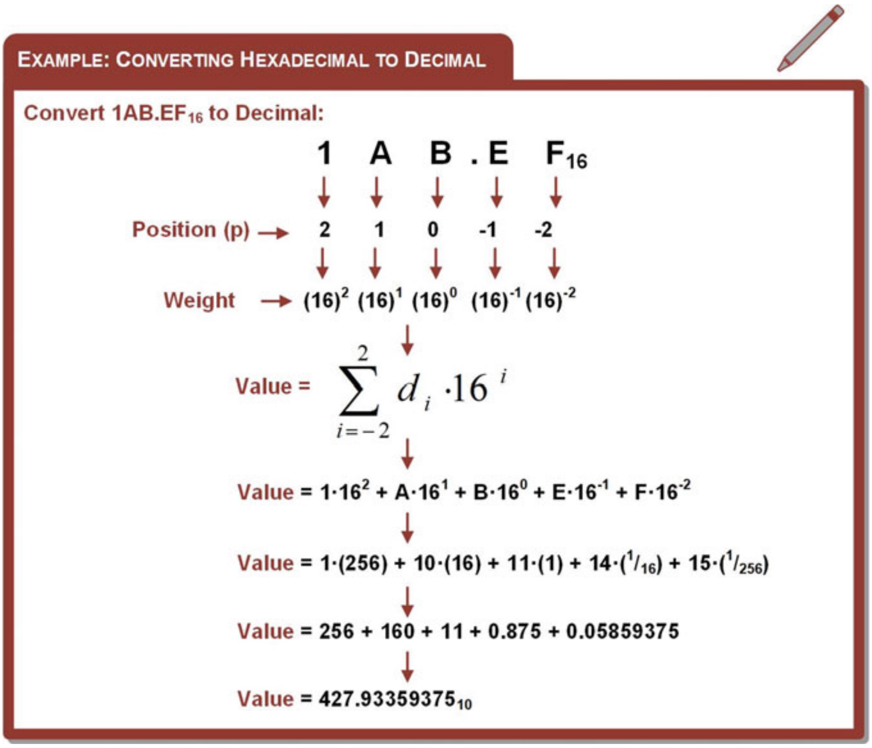
$$\text{Value} = 1\cdot(4) + 0\cdot(2) + 1\cdot(1) + 1\cdot(^1/_2) + 1\cdot(^1/_4)$$

$$\text{Value} = 4 + 0 + 1 + 0.5 + 0.25$$

$$\text{Value} = 5.75_{10}$$

Example 2.2
Converting binary to decimal

Let's now convert $1AB.EF_{16}$ to decimal. The same process is followed with the exception that the base is changed to 16. When performing the conversion, the decimal equivalent of the numerals A–F needs to be used. Example 2.3 shows the step-by-step process converting a hexadecimal number to decimal.



EXAMPLE: CONVERTING HEXADECIMAL TO DECIMAL

Convert $1AB.EF_{16}$ to Decimal:

$$\begin{array}{llllll} & 1 & A & B\,. & E & F_{16} \\ \text{Position (p)} \rightarrow & 2 & 1 & 0 & -1 & -2 \\ \text{Weight} \rightarrow & (16)^2 & (16)^1 & (16)^0 & (16)^{-1} & (16)^{-2} \end{array}$$

$$\text{Value} = \sum_{i=-2}^{2} d_i \cdot 16^{i}$$

$$\text{Value} = 1 \cdot 16^2 + A \cdot 16^1 + B \cdot 16^0 + E \cdot 16^{-1} + F \cdot 16^{-2}$$

$$\text{Value} = 1 \cdot (256) + 10 \cdot (16) + 11 \cdot (1) + 14 \cdot (^1/_{16}) + 15 \cdot (^1/_{256})$$

$$\text{Value} = 256 + 160 + 11 + 0.875 + 0.05859375$$

$$\text{Value} = 427.93359375_{10}$$

Example 2.3
Converting hexadecimal to decimal

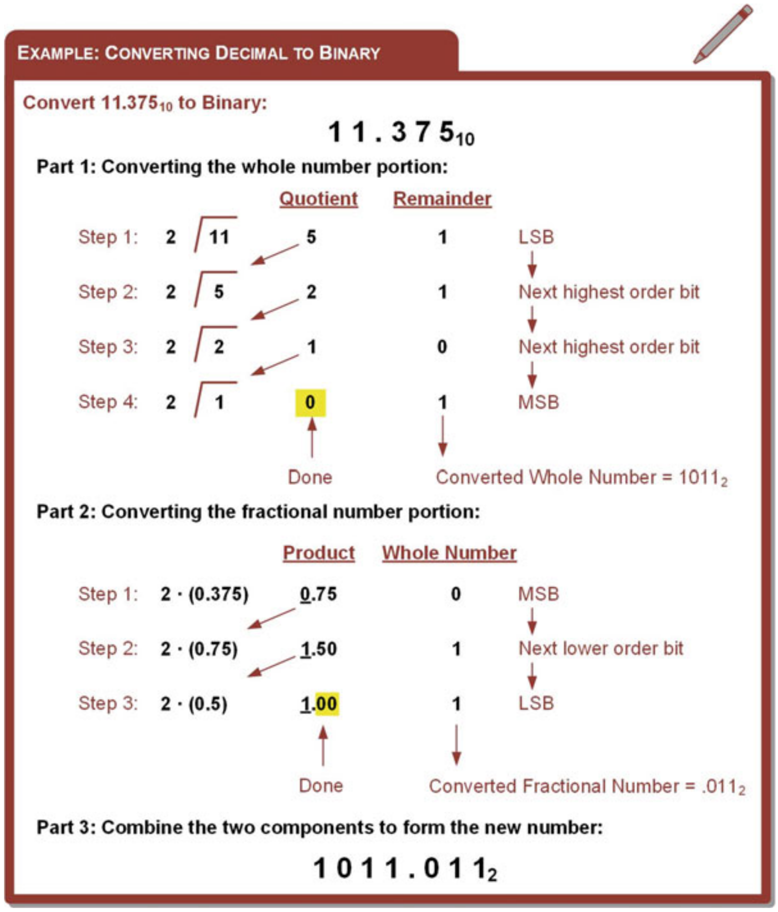
In some cases, it is desired to specify a *level of accuracy* for the conversion in order to bound the number of fractional digits in the final result. For example, if the conversion in Example 2.3 was stated as "convert $1AB.EF_{16}$ to decimal with a *fractional accuracy of 2 digits*," the final result would be 427.93_{10}. How rounding is handled can also be specified with the two options being *with* or *without rounding.* In the case where the conversion is performed *with rounding*, additional fractional digits may need to be computed to determine if the least significant digit of the new decimal fraction needs to be altered. For example, let's say the conversion in Example 2.3 is stated as "convert $1AB.EF_{16}$ to decimal with a fractional accuracy of *4 digits with rounding*." In this case, the final result would be 427.9336_{10}. Notice how rounding was applied to the digit in position $p = -3$ changing it from a 5 to a 6 based on the value in position $p = -4$. Now let's say the conversion in Example 2.3 is stated as "convert $1AB.EF_{16}$ to decimal with a fractional accuracy *of 4 digits without rounding*." In this case, the final result would be 427.9335_{10}. Notice how without rounding simply drops all of the digits beyond the specified level of accuracy.

2.1.2.2 Converting from Decimal

The process of converting from decimal to another base consists of two separate algorithms. There is one algorithm for converting the whole number portion of the number and another algorithm for converting the fractional portion of the number. The process for converting the whole number portion is to divide the decimal number by the base of the system you wish to convert to. The division will result in a quotient and a whole number remainder. The remainder is recorded as the *least significant numeral* in the converted number. The resulting quotient is then divided again by the base, which results in a new quotient and new remainder. The remainder is recorded as the next higher-order numeral in the new number. This process is repeated until a quotient of 0 is achieved. At that point the conversion is complete. The remainders will always be within the numeral set of the base being converted to.

The process for converting the fractional portion is to multiply just the fractional component of the number by the base. This will result in a product that contains a whole number and a fraction. The whole number is recorded as the *most significant digit* of the new converted number. The new fractional portion is then multiplied again by the base with the whole number portion being recorded as the next lower-order numeral. This process is repeated until the product yields a fractional component equal to 0 or the desired level of accuracy has been achieved. The level of accuracy is specified by the number of numerals in the new converted number. For example, the conversion would be stated as “convert this decimal number to binary with a fractional accuracy of 4 bits.” This means the final result would only have 4 bits in the fraction. In cases where the conversion does not yield exactly 4 fractional bits, there are two approaches that can be used. The first is to have *no rounding*, which means the conversion simply stops at the desired accuracy. The second is to apply *rounding*, which means additional bits beyond the desired accuracy are computed in order to determine whether the least significant bit is reported.

Let’s convert 11.375_{10} to binary. Example 2.4 shows the step-by-step process converting a decimal number to binary.



EXAMPLE: CONVERTING DECIMAL TO BINARY

Convert 11.375_{10} to Binary:

11.375_{10}

Part 1: Converting the whole number portion:

		Quotient	Remainder	
Step 1:	2 ⟌ 11	5	1	LSB
Step 2:	2 ⟌ 5	2	1	Next highest order bit
Step 3:	2 ⟌ 2	1	0	Next highest order bit
Step 4:	2 ⟌ 1	0	1	MSB

Done

Converted Whole Number = 1011_2

Part 2: Converting the fractional number portion:

		Product	Whole Number	
Step 1:	2 · (0.375)	0.75	0	MSB
Step 2:	2 · (0.75)	1.50	1	Next lower order bit
Step 3:	2 · (0.5)	1.00	1	LSB

Done

Converted Fractional Number = $.011_2$

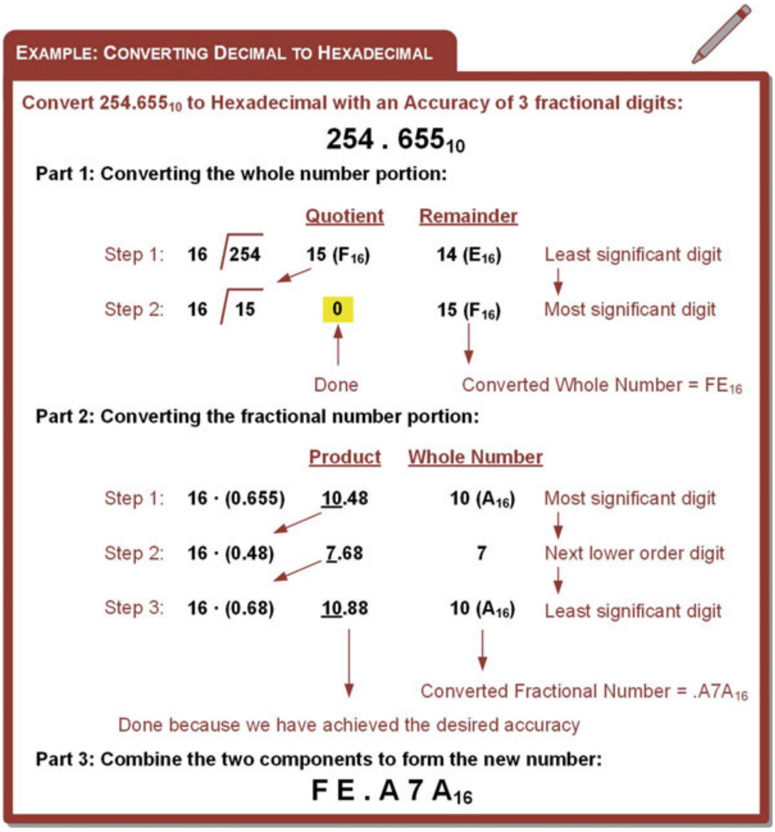
Part 3: Combine the two components to form the new number:

1011.011_2

Example 2.4
Converting decimal to binary

In many binary conversions to binary, the number of fractional bits that result from the conversion is more than what is needed. In this case, rounding is applied to limit the fractional accuracy. The simplest rounding approach for binary numbers is to continue the conversion for one more bit beyond the desired fractional accuracy. If the next bit is a 0, then you leave the fractional component of the number as is. If the next bit is a 1, you round the least significant bit of your number up. Often this rounding will result in a cascade of roundings from the LSB to the MSB. As an example, let's say that the conversion in Example 2.4 was specified to have a fractional accuracy of 2 bits. If the bit in position $p = -3$ was a 0 (which it is not, but let's just say it is for the sake of this example), then the number would be left as is and the final converted number would be 1011.01_2; however, if the bit in position $p = -3$ was a 1 (as it actually is in Example 2.4), then we would need to apply rounding. We would start with the bit in position $p = -2$. Since it is a 1, we would round that up to a 0, but we would need to apply the overflow of this rounding to the next higher-order bit in position $p =-1$. That would then cause the value of $p = -1$ to go from a 0 to a 1. The final result of the conversion with rounding would be 1011.10_2.

Let's now convert 254.655_{10} to hexadecimal with an accuracy of three fractional digits. When doing this conversion, all of the divisions and multiplications are done using decimal. If the results end up between 10_{10} and $15_{10,}$ then the decimal numbers are substituted with their hex symbol equivalent (i.e., A to F). Example 2.5 shows the step-by-step process of converting a decimal number to hex with a fractional accuracy of three digits.


EXAMPLE: CONVERTING DECIMAL TO HEXADECIMAL

Convert 254.655_{10} to Hexadecimal with an Accuracy of 3 fractional digits:

$254 \,.\, 655_{10}$

Part 1: Converting the whole number portion:

		Quotient	Remainder	
Step 1:	16 / 254	15 (F_{16})	14 (E_{16})	Least significant digit
Step 2:	16 / 15	0	15 (F_{16})	Most significant digit

Done

Converted Whole Number = FE_{16}

Part 2: Converting the fractional number portion:

		Product	Whole Number	
Step 1:	16 · (0.655)	10.48	10 (A_{16})	Most significant digit
Step 2:	16 · (0.48)	7.68	7	Next lower order digit
Step 3:	16 · (0.68)	10.88	10 (A_{16})	Least significant digit

Converted Fractional Number = $.A7A_{16}$

Done because we have achieved the desired accuracy

Part 3: Combine the two components to form the new number:

$FE.A7A_{16}$

Example 2.5
Converting decimal to hexadecimal

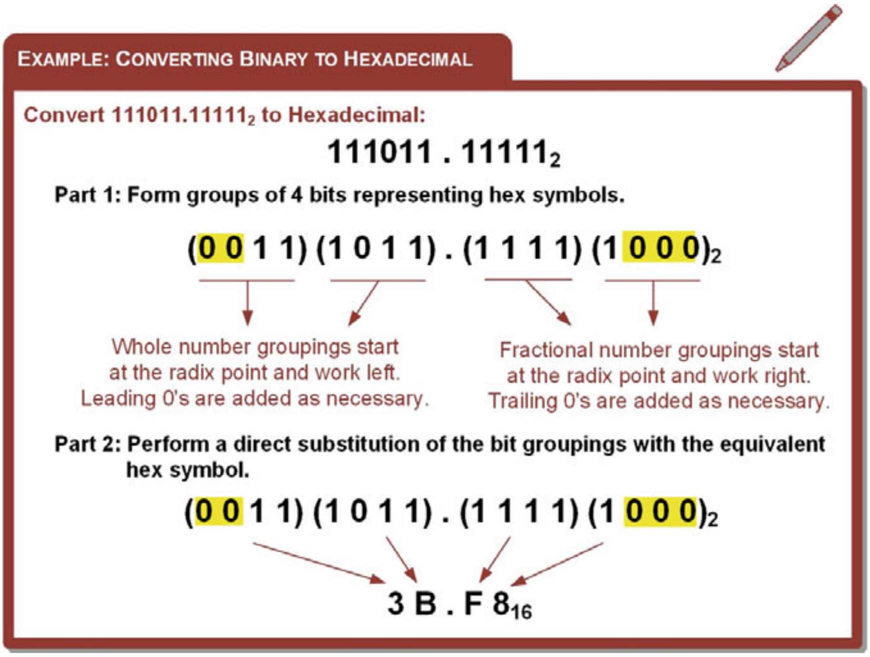
Rounding of hexadecimal digits uses a similar approach as when rounding decimal numbers, with the exception that the middle of the range of the numbers lies between digits 7_{16} and 8_{16}. This means that any number to be rounded that is 8_{16} or greater will be rounded up. Numbers that are 7_{16} or less will be rounded down, which means the fractional component of the converted number is left as in.

2.1.2.3 Converting Between 2^n Bases

Converting between 2^n bases (e.g., 2, 4, 8, and 16) takes advantage of the direct mapping that each of these bases has back to binary. Base 8 numbers take exactly 3 binary bits to represent all eight symbols (i.e., $0_8 = 000_2$, $7_8 = 111_2$). Base 16 numbers take exactly 4 binary bits to represent all 16 symbols (i.e., $0_{16} = 0000_2$, $F_{16} = 1111_2$).

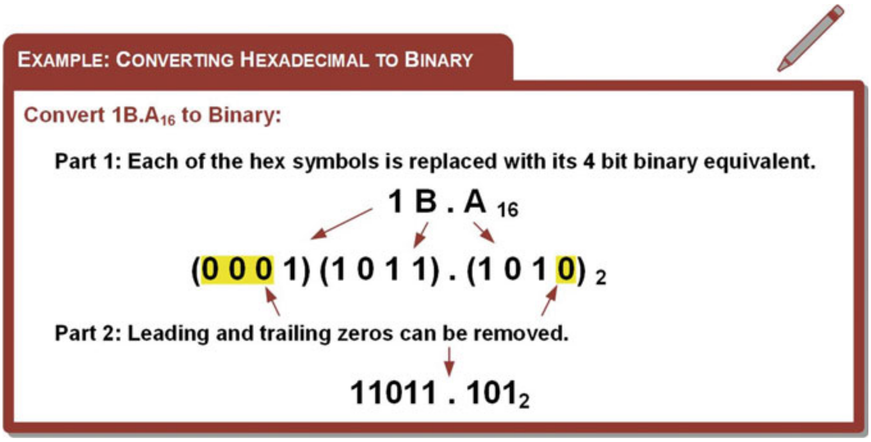
When converting *from* binary to any other 2^n base, the whole number bits are grouped into the appropriate-sized sets starting from the radix point and working left. If the final leftmost grouping does not have enough symbols, it is simply padded on left with leading 0s. Each of these groups is then directly substituted with their 2^n base symbol. The fractional number bits are also grouped into the appropriate-sized sets starting from the radix point, but this time working right. Again, if the final rightmost grouping does not have enough symbols, it is simply padded on the right with trailing 0s. Each of these groups is then directly substituted with their 2^n base symbol.

Example 2.6 shows the step-by-step process of converting a binary number to hexadecimal.


EXAMPLE: CONVERTING BINARY TO HEXADECIMAL
Convert 111011.11111_2 to Hexadecimal:
$111011\ .\ 11111_2$
Part 1: Form groups of 4 bits representing hex symbols.
$(0\,0\,1\,1)\,(1\,0\,1\,1)\,.\,(1\,1\,1\,1)\,(1\,0\,0\,0)_2$
Whole number groupings start at the radix point and work left. Leading 0's are added as necessary.
Fractional number groupings start at the radix point and work right. Trailing 0's are added as necessary.
Part 2: Perform a direct substitution of the bit groupings with the equivalent hex symbol.
$(0\,0\,1\,1)\,(1\,0\,1\,1)\,.\,(1\,1\,1\,1)\,(1\,0\,0\,0)_2$
$3\,B\,.\,F\,8_{16}$

Example 2.6
Converting binary to hexadecimal

Example 2.7 shows the step-by-step process of converting a hexadecimal number to binary.



EXAMPLE: CONVERTING HEXADECIMAL TO BINARY

Convert $1B.A_{16}$ to Binary:

Part 1: Each of the hex symbols is replaced with its 4 bit binary equivalent.

$$1\,B\,.\,A_{16}$$

$$(0\,0\,0\,1)\,(1\,0\,1\,1)\,.\,(1\,0\,1\,0)_2$$

Part 2: Leading and trailing zeros can be removed.

$$11011\,.\,101_2$$

Example 2.7
Converting hexadecimal to binary

CONCEPT CHECK

CC2.1.2 A “googol” is the term for the decimal number 1e100. When written out manually this number is a 1 with 100 0s after it (e.g., 10,000,000,000,000,000,000,000, 000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000, 000,000,000,000,000,000,000,000). This term is more commonly associated with the search engine company Google, which uses a different spelling but is pronounced the same. How many bits does it take to represent a googol in binary?

A) 100 bits

B) 256 bits

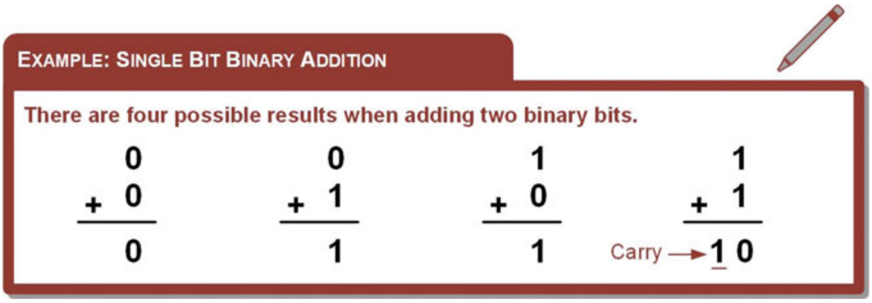
C) 332 bits

D) 333 bits

2.1.3 Binary Arithmetic

2.1.3.1 Addition (Carries)

Binary addition is a straightforward process that mirrors the approach we have learned for longhand decimal addition. The two numbers (or terms) to be added are aligned at the radix point and addition begins at the least significant bit. If the sum of the least significant position yields a value with two bits (e.g., 10_2), then the least significant bit is recorded, and the most significant bit is *carried* to the next higher position. The sum of the next higher position is then performed including the potential *carry bit* from the prior addition. This process continues from the least significant position to the most significant position. Example 2.8 shows how addition is performed on two individual bits.

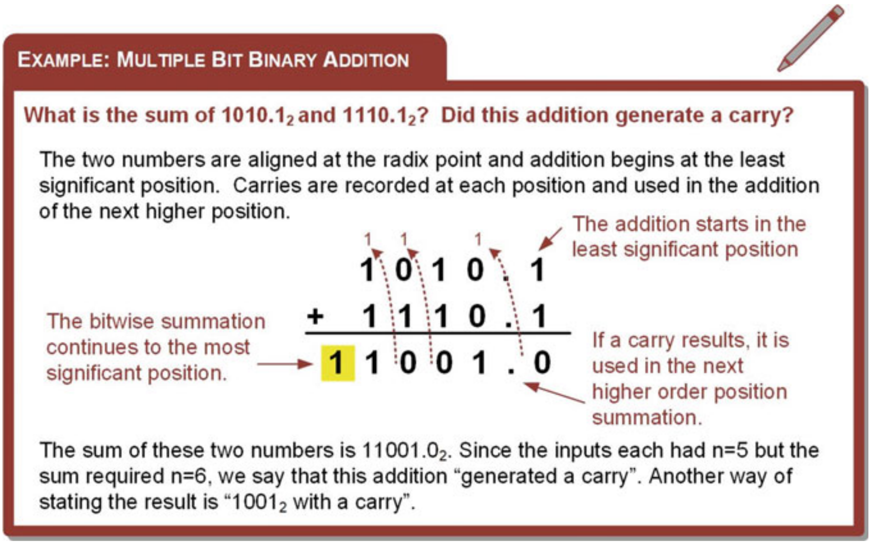


EXAMPLE: SINGLE BIT BINARY ADDITION

There are four possible results when adding two binary bits.

Example 2.8
Single bit binary addition

When performing binary addition, the width of the inputs and output is fixed (i.e., *n*-bits). Carries that exist within the *n*-bits are treated in the normal fashion of including them in the next higher position sum; however, if the highest position summation produces a carry, this is a uniquely named event. This event is called a *carry out* or the sum is said to *generate a carry.* The reason this type of event is given special terminology is because in real circuitry, the number of bits of the inputs and output is fixed in hardware and the carryout is typically handled by a separate circuit. Example 2.9 shows this process when adding two 4-bit numbers.



EXAMPLE: MULTIPLE BIT BINARY ADDITION

What is the sum of 1010.1_2 and 1110.1_2? Did this addition generate a carry?

The two numbers are aligned at the radix point and addition begins at the least significant position. Carries are recorded at each position and used in the addition of the next higher position.

The addition starts in the least significant position

The bitwise summation continues to the most significant position.

If a carry results, it is used in the next higher order position summation.

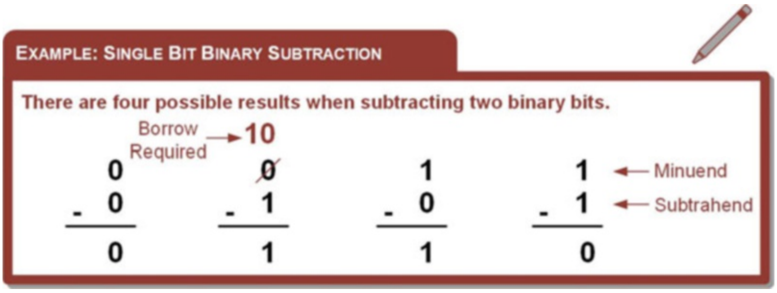
The sum of these two numbers is 11001.0_2. Since the inputs each had n=5 but the sum required n=6, we say that this addition "generated a carry". Another way of stating the result is "1001_2 with a carry".

Example 2.9
Multiple bit binary addition

The largest decimal sum that can result from the addition of two binary numbers is given by $2{\cdot}(2^n{-}1)$. For example, two 8-bit numbers to be added could both represent their highest decimal value of $(2^n{-}1)$ or 255_{10} (i.e., $1111\ 1111_2$). The sum of this number would result in 510_{10} or $(1\ 1111\ 1110_2)$. Notice that the largest sum achievable would only require one additional bit. This means that a single carry bit is sufficient to handle all possible magnitudes for binary addition.

2.1.3.2 Subtraction (Borrows)

Binary subtraction also mirrors longhand decimal subtraction. In subtraction, the formal terms for the two numbers being operated on are *minuend* and *subtrahend*. The subtrahend is subtracted from the minuend to find the *difference*. In longhand subtraction, the minuend is the top number and the subtrahend is the bottom number. For a given position if the minuend is less than the subtrahend, it needs to *borrow* from the next higher-order position to produce a difference that is positive. If the next higher position does not have a value that can be borrowed from (i.e., 0), then it in turn needs to borrow from the next higher position and so forth. Example 2.10 shows how subtraction is performed on two individual bits.

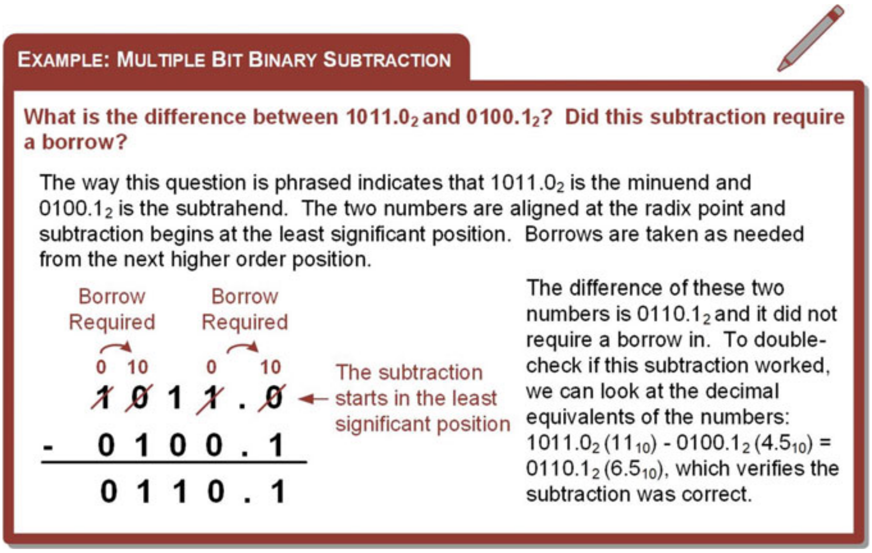


EXAMPLE: SINGLE BIT BINARY SUBTRACTION

There are four possible results when subtracting two binary bits.

Example 2.10
Single bit binary subtraction

As with binary addition, binary subtraction is accomplished on fixed widths of inputs and output (i.e., *n*-bits). The minuend and subtrahend are aligned at the radix point and subtraction begins at the least significant bit position. Borrows are used as necessary as the subtractions move from the least significant position to the most significant position. If the most significant position requires a borrow, this is a uniquely named event. This event is called a *borrow in* or the subtraction is said to *require a borrow.* Again, the reason this event is uniquely named is because in real circuitry, the number of bits of the input and output is fixed in hardware and the borrow in is typically handled by a separate circuit. Example 2.11 shows this process when subtracting two 4-bit numbers.



EXAMPLE: MULTIPLE BIT BINARY SUBTRACTION

What is the difference between 1011.0_2 and 0100.1_2? Did this subtraction require a borrow?

The way this question is phrased indicates that 1011.0_2 is the minuend and 0100.1_2 is the subtrahend. The two numbers are aligned at the radix point and subtraction begins at the least significant position. Borrows are taken as needed from the next higher order position.

The subtraction starts in the least significant position

The difference of these two numbers is 0110.1_2 and it did not require a borrow in. To double-check if this subtraction worked, we can look at the decimal equivalents of the numbers: $1011.0_2\,(11_{10})$ - $0100.1_2\,(4.5_{10})$ = $0110.1_2\,(6.5_{10})$, which verifies the subtraction was correct.

Example 2.11
Multiple bit binary subtraction

Notice that if the minuend is less than the subtrahend, then the difference will be negative. At this point, we need a way to handle negative numbers.

Concept Check

CC2.1.3 If an 8-bit computer system can only perform unsigned addition on 8-bit inputs and produce an 8-bit sum, how is it possible for this computer to perform addition on numbers that are larger than what can be represented with 8-bits (e.g., $1000_{10} + 1000_{10} = 2000_{10}$)?

A) There are multiple 8-bit adders in a computer to handle large numbers.

B) The result is simply rounded to the nearest 8-bit number.

C) The computer returns an error and requires smaller numbers to be entered.

D) The computer keeps track of the carry out and uses it in a subsequent 8-bit addition, which enables larger numbers to be handled.

2.1.4 Unsigned and Signed Numbers

All of the number systems presented in the prior sections were positive. We need to also have a mechanism to indicate negative numbers. When looking at negative numbers, we only focus on the mapping between decimal and binary since octal and hexadecimal are used as just another representation of a binary number. In decimal, we are able to use the negative *sign* in front of a number to indicate it is negative (e.g., -34_{10}). In binary, this notation works fine for writing numbers on paper (e.g., -1010_2), but we need a mechanism that can be implemented using real circuitry. In a real digital circuit, the circuits can only deal with 0s and 1s. There is no "−" in a digital circuit. Since we only have 0s and 1s in the hardware, we use a bit to represent whether a number is positive or negative. This is referred to as the *sign bit*. If a binary number is not going to have any negative values, then it is called an *unsigned* number and it can only represent positive numbers. If a binary number is going to allow negative numbers, it is called a *signed* number. It is important to always keep track of the type of number we are using as the same bit values can represent very different numbers depending on the coding mechanism that is being used.

2.1.4.1 Unsigned Numbers

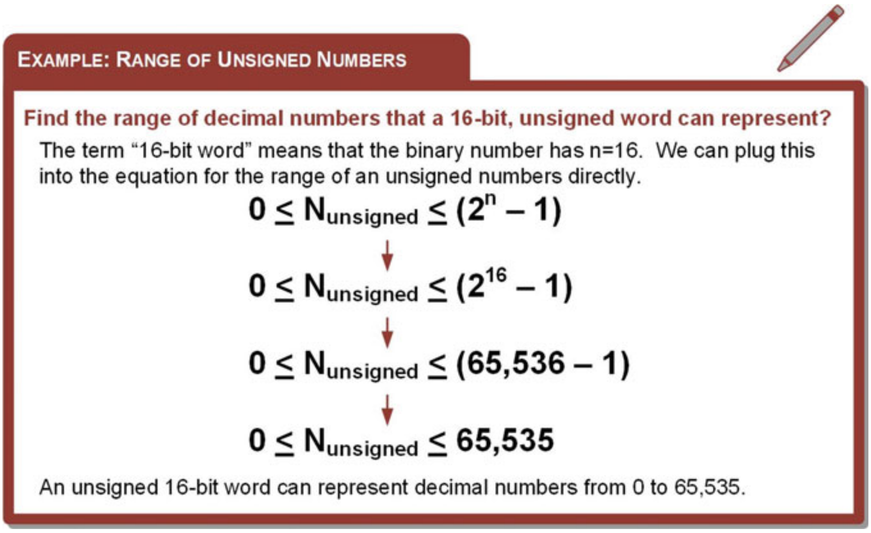
An unsigned number is one that does not allow negative numbers. When talking about this type of code, the number of bits is fixed and stated up front. We use the variable n to represent the number of bits in the number. For example, if we had an 8-bit number, we would say, "This is an 8-bit, unsigned number."

The number of unique codes in an unsigned number is given by 2^n. For example, if we had an 8-bit number, we would have 2^8 or 256 unique codes (e.g., $0000\ 0000_2$ to $1111\ 1111_2$).

The *range* of an unsigned number refers to the decimal values that the binary code can represent. If we use the notation N_{unsigned} to represent any possible value that an n-bit, unsigned number can take on, the range would be defined as: $0 < N_{\text{unsigned}} < (2^n - 1)$

$$\textbf{Range of anUNSIGNED number} \Rightarrow \mathbf{0} < \boldsymbol{N}_{\textbf{unsigned}} < (\mathbf{2}^{\boldsymbol{n}} - \mathbf{1})$$

For example, if we had an unsigned number with $n = 4$, it could take on a range of values from $+0_{10}$ (0000_2) to $+15_{10}$ (1111_2). Notice that while this number has 16 unique possible codes, the highest decimal value it can represent is 15_{10}. This is because one of the unique codes represents 0_{10}. This is the reason that the highest decimal value that can be represented is given by ($2^n - 1$). Example 2.12 shows this process for a 16-bit number.



EXAMPLE: RANGE OF UNSIGNED NUMBERS

Find the range of decimal numbers that a 16-bit, unsigned word can represent?

The term "16-bit word" means that the binary number has n=16. We can plug this into the equation for the range of an unsigned numbers directly.

$$0 \le N_{unsigned} \le (2^n - 1)$$

$$\downarrow$$

$$0 \le N_{unsigned} \le (2^{16} - 1)$$

$$\downarrow$$

$$0 \le N_{unsigned} \le (65{,}536 - 1)$$

$$\downarrow$$

$$0 \le N_{unsigned} \le 65{,}535$$

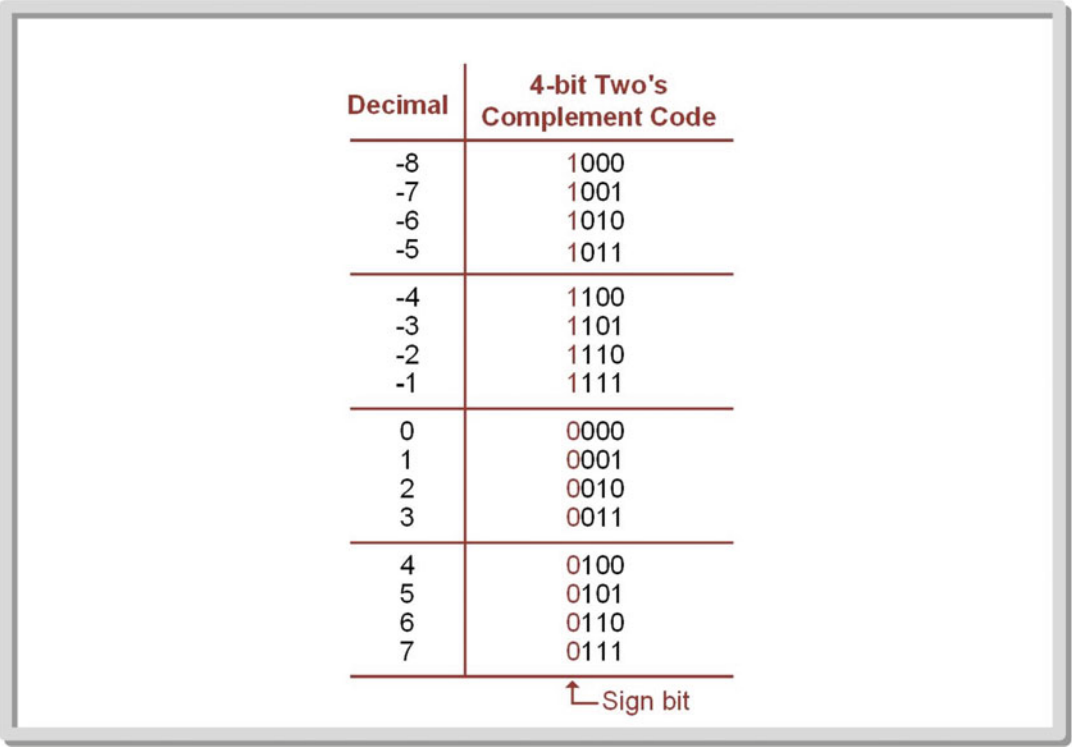
An unsigned 16-bit word can represent decimal numbers from 0 to 65,535.

Example 2.12
Finding the range of an unsigned number

2.1.4.2 Signed Numbers (Two's Complement)

Signed numbers are able to represent both positive and negative numbers. The most significant bit of a signed numbers is always the *sign bit*, which represents whether the number is positive or negative. The sign bit is defined to be a *0 if the number is positive* and *1 if the number is negative*. When using signed numbers, the number of bits is fixed so that the sign bit is always in the same position. There are a variety of ways to encode negative numbers using a sign bit. The encoding method used exclusively in modern computers is called *two's complement*. When talking about a signed number, the number of bits and the type of encoding is always stated. For example, we would say, "This is an 8-bit, two's complement number."

In a two's complement encoding scheme, the negative number is obtained by subtracting its positive equivalent from 2^n. This is identical to performing a complement on the positive equivalent and then adding 1. If a carry is generated, it is discarded. This procedure is called *taking the two's complement of a number* or *performing two's complement negation*. The procedure of complementing each bit and adding 1 is the most common technique to perform a two's complement. In this way, the most significant bit of the number is still the sign bit ($0 =$ positive, $1 =$ negative), but all of the negative numbers are in essence *shifted up* so that there are not duplicate codes for 0. Taking the two's complement of a positive number will give its negative counterpart and vice versa. Table 2.2 shows the values represented by a 4-bit two's complement number.



Decimal	4-bit Two's Complement Code
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

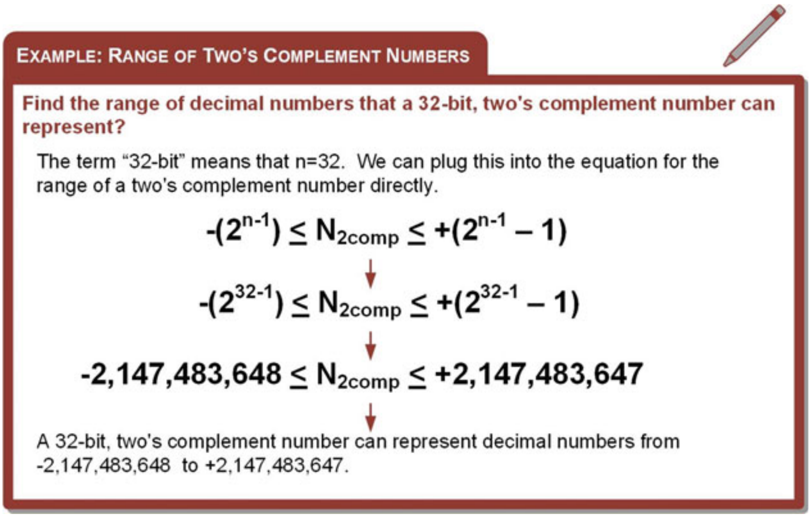
Table 2.2
Decimal values that a 4-bit, two's complement code can represent

There are many advantages of two's complement encoding that make it the prevalent signed number scheme in modern computers. First, it uses all of the possible codes available to represent a number (i.e., there are not two codes representing 0 that come from the notion of a positive 0 and negative 0). Two's complement also provides a sequence of numbers that are each 1 value away from their neighboring codes when applying a traditional binary count. This ordering allows traditional addition operations to be applied to two's complement in the same manner as when using unsigned numbers. Finally, the sequence of codes has a rollover characteristic, meaning that the highest value is only 1 value away from the lowest value. This characteristic means that when incrementing through as set of two's complement numbers, when you reach the maximum value and increment, it will go to the lowest value and continue incrementing up.

The range of two's complement numbers is a critical consideration when using this encoding scheme. If we use the notation $N_{2\text{comp}}$ to represent any possible value that an n-bit, two's complement number can take on, the range of a two's complement number is defined as:

$$\textbf{Range of a TWO'S COMPLEMENT number} \Rightarrow -\left(\mathbf{2}^{n-1}\right) \leq N_{2\text{'s comp}} \leq +\left(\mathbf{2}^{n-1}-\mathbf{1}\right)$$

Example 2.13 shows how to use this expression to find the range of decimal values that a 32-bit, two's complement code can represent.



EXAMPLE: RANGE OF TWO'S COMPLEMENT NUMBERS

Find the range of decimal numbers that a 32-bit, two's complement number can represent?

The term "32-bit" means that n=32. We can plug this into the equation for the range of a two's complement number directly.

$$-(2^{n-1}) \leq N_{2comp} \leq +(2^{n-1} - 1)$$

$$\downarrow$$

$$-(2^{32-1}) \leq N_{2comp} \leq +(2^{32-1} - 1)$$

$$\downarrow$$

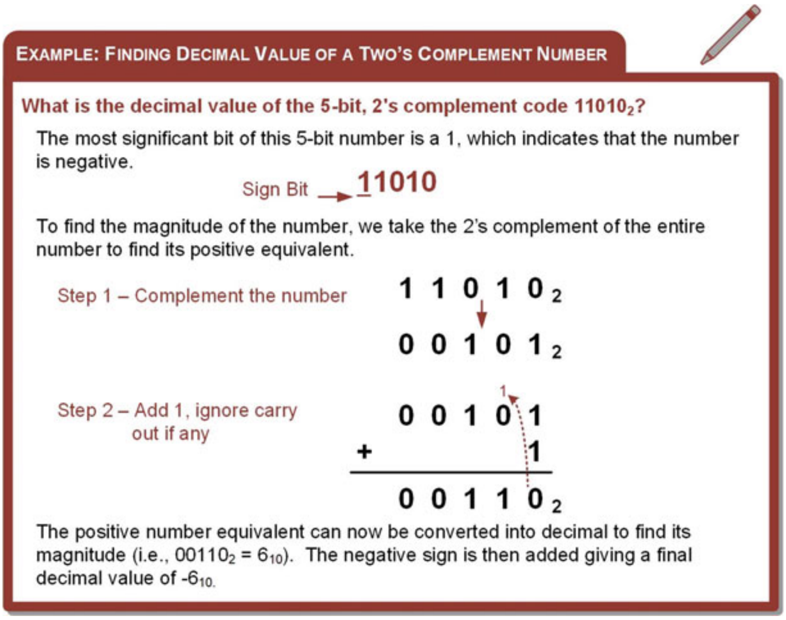
$$-2{,}147{,}483{,}648 \leq N_{2comp} \leq +2{,}147{,}483{,}647$$

$$\downarrow$$

A 32-bit, two's complement number can represent decimal numbers from -2,147,483,648 to +2,147,483,647.

Example 2.13
Finding the range of a two's complement number

The process of finding the decimal value of a two's complement number involves first identifying whether the number is positive or negative by looking at the sign bit. If the number is positive (i.e., the sign bit is 0), then the number is treated as an unsigned code and is converted to decimal using the standard conversion procedure described in prior sections. If the number is negative (i.e., the sign bit is 1), then the number sign is recorded separately, and a *two's complement negation* is performed on the code in order to convert it to its positive magnitude equivalent. This new positive number is then converted to decimal using the standard conversion procedure. The final step is to apply the sign. Example 2.14 shows an example of this process on a negative number.



EXAMPLE: FINDING DECIMAL VALUE OF A TWO'S COMPLEMENT NUMBER

What is the decimal value of the 5-bit, 2's complement code 11010_2?

The most significant bit of this 5-bit number is a 1, which indicates that the number is negative.

Sign Bit → 11010

To find the magnitude of the number, we take the 2's complement of the entire number to find its positive equivalent.

Step 1 – Complement the number

$$\begin{array}{c} 1\ 1\ 0\ 1\ 0_2 \\ \downarrow \\ 0\ 0\ 1\ 0\ 1_2 \end{array}$$

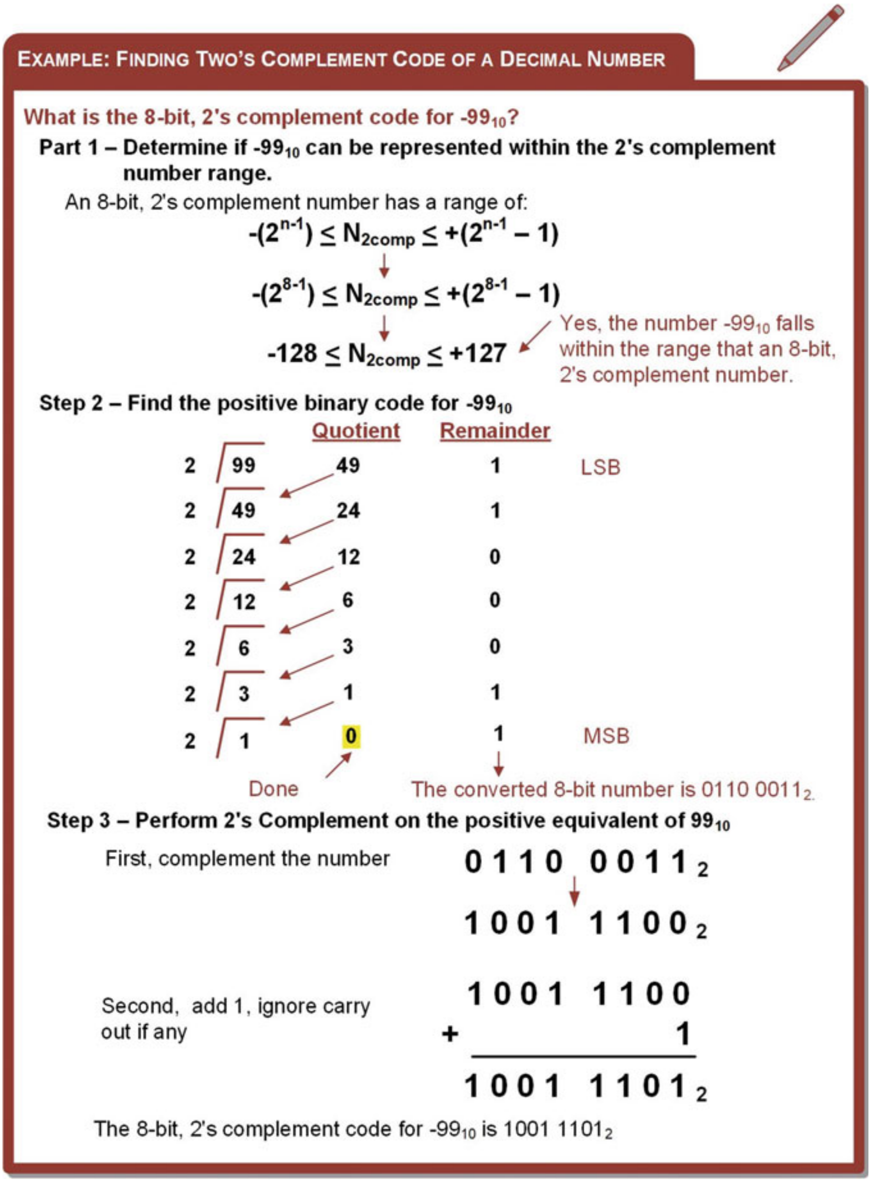
Step 2 – Add 1, ignore carry out if any

$$\begin{array}{r} {}^{1} \\ 0\ 0\ 1\ 0\ 1 \\ +1 \\ \hline 0\ 0\ 1\ 1\ 0_2 \end{array}$$

The positive number equivalent can now be converted into decimal to find its magnitude (i.e., $00110_2 = 6_{10}$). The negative sign is then added giving a final decimal value of -6_{10}.

Example 2.14
Finding the decimal value of a two's complement number

To convert a decimal number into its two's complement code, the range is first checked to determine whether the number can be represented with the allocated number of bits. The next step is to convert the decimal number into unsigned binary. The final step is to apply the sign bit. If the original decimal number was positive, then the conversion is complete. If the original decimal number was negative, then the two's complement is taken on the unsigned binary code to find its negative equivalent. Example 2.15 shows this procedure when converting -99_{10} to its 8-bit, two's complement code.



EXAMPLE: FINDING TWO'S COMPLEMENT CODE OF A DECIMAL NUMBER

What is the 8-bit, 2's complement code for -99_{10}?

Part 1 – Determine if -99_{10} can be represented within the 2's complement number range.

An 8-bit, 2's complement number has a range of:

$$-(2^{n-1}) \le N_{2comp} \le +(2^{n-1} - 1)$$
$$\downarrow$$
$$-(2^{8-1}) \le N_{2comp} \le +(2^{8-1} - 1)$$
$$\downarrow$$
$$-128 \le N_{2comp} \le +127$$

Yes, the number -99_{10} falls within the range that an 8-bit, 2's complement number.

Step 2 – Find the positive binary code for -99_{10}

		Quotient	Remainder	
2	99	49	1	LSB
2	49	24	1	
2	24	12	0	
2	12	6	0	
2	6	3	0	
2	3	1	1	
2	1	0	1	MSB

Done

The converted 8-bit number is $0110\ 0011_2$.

Step 3 – Perform 2's Complement on the positive equivalent of 99_{10}

First, complement the number

$$0110\ 0011_2$$
$$\downarrow$$
$$1001\ 1100_2$$

Second, add 1, ignore carry out if any

$$\begin{array}{r} 1001\ 1100 \\ +\qquad\quad 1 \\ \hline 1001\ 1101_2 \end{array}$$

The 8-bit, 2's complement code for -99_{10} is $1001\ 1101_2$

Example 2.15
Finding the two's complement code of a decimal number

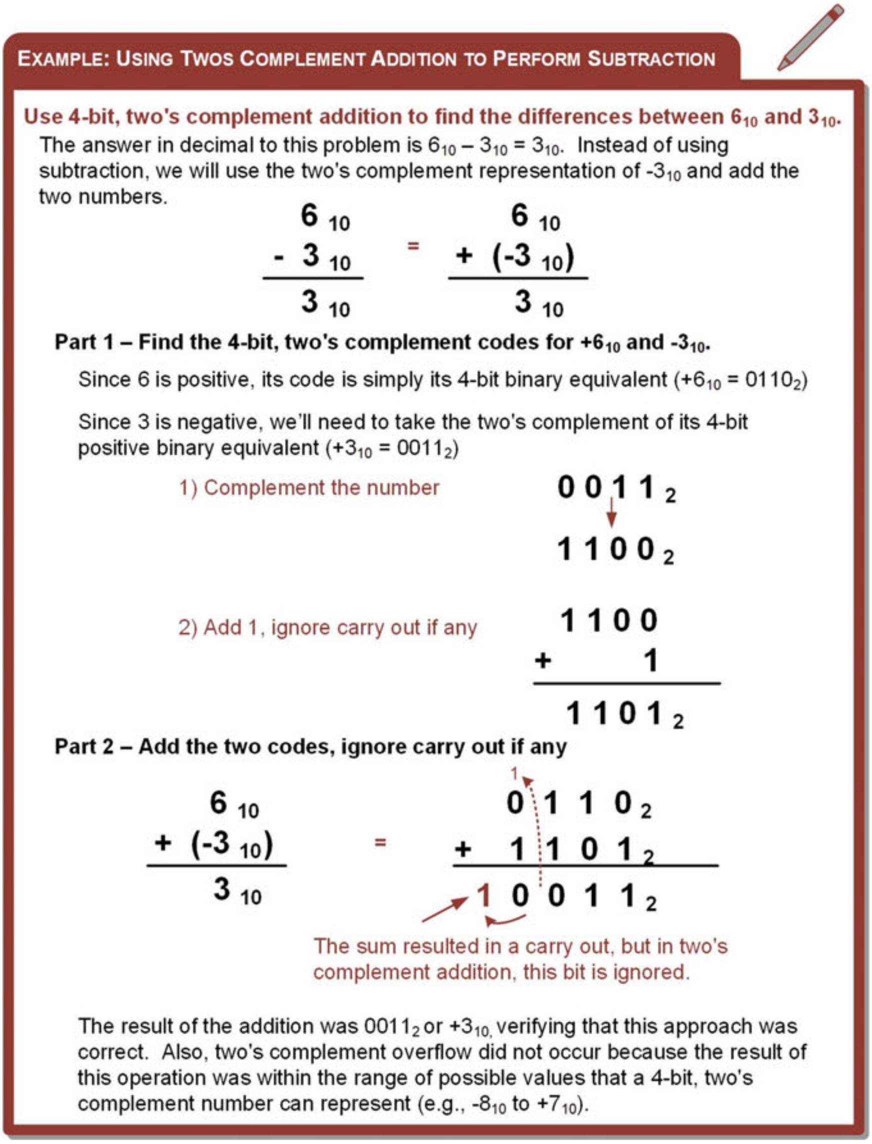
2.1.4.3 Arithmetic with Two's Complement

Two's complement has a variety of arithmetic advantages. First, the operations of addition, subtraction, and multiplication are handled exactly the same as when using unsigned numbers. This means that duplicate circuitry is not needed in a system that uses both number types. Second, the ability to convert a number from positive to its negative representation by performing a *two's complement* means that an adder circuit can be used for subtraction. For example, if we wanted to perform the subtraction $13_{10} - 4_{10} = 9_{10}$, this is the same as performing $13_{10} + (-4_{10}) = 9_{10}$. This allows us to use a single adder circuit to perform both addition and subtraction as long as we have the ability to take the two's complement of a number. Creating a circuit to perform two's complement can be simpler and faster than building a separate subtraction circuit, so this approach can sometimes be advantageous.

There are specific rules for performing two's complement arithmetic that must be followed to ensure proper results. First, any carry or borrow that is generated is *ignored*. The second rule that must be followed is to always check if *two's complement overflow* occurred. Two's complement overflow refers to when the result of the operation falls outside of the range of values that can be represented by the number of bits being used. For example, if you are performing 8-bit, two's complement addition, the range of decimal values that can be represented is -128_{10} to $+127_{10}$. Having two input terms of 127_{10} ($0111\ 1111_2$) is perfectly legal because they can be represented by the 8 bits of the two's complement number; however, the summation of $127_{10} + 127_{10} = 254_{10}$ ($1111\ 1110_2$). This number does *not* fit within the range of values that can be represented and is actually the two's complement code for -2_{10}, which is obviously incorrect. Two's complement overflow occurs if any of the following occurs:

- The sum of like signs results in an answer with opposite sign (i.e., Positive + Positive = Negative or Negative + Negative = Positive).
- The subtraction of a positive number from a negative number results in a positive number (i.e., Negative − Positive = Positive).
- The subtraction of a negative number from a positive number results in a negative number (i.e., Positive − Negative = Negative).

Computer systems that use two's complement have a dedicated logic circuit that monitors for any of these situations and lets the operator know that overflow has occurred. These circuits are straightforward since they simply monitor the sign bits of the input and output codes. Example 2.16 shows how to use two's complement in order to perform subtraction using an addition operation.



EXAMPLE: USING TWOS COMPLEMENT ADDITION TO PERFORM SUBTRACTION

Use 4-bit, two's complement addition to find the differences between 6_{10} and 3_{10}.

The answer in decimal to this problem is $6_{10} - 3_{10} = 3_{10}$. Instead of using subtraction, we will use the two's complement representation of -3_{10} and add the two numbers.

$$\begin{array}{r} 6_{10} \\ -\ 3_{10} \\ \hline 3_{10} \end{array} \quad = \quad \begin{array}{r} 6_{10} \\ +\ (-3_{10}) \\ \hline 3_{10} \end{array}$$

Part 1 – Find the 4-bit, two's complement codes for $+6_{10}$ and -3_{10}.

Since 6 is positive, its code is simply its 4-bit binary equivalent ($+6_{10} = 0110_2$)

Since 3 is negative, we'll need to take the two's complement of its 4-bit positive binary equivalent ($+3_{10} = 0011_2$)

1) Complement the number

$$\begin{array}{c} 0011_2 \\ \downarrow \\ 1100_2 \end{array}$$

2) Add 1, ignore carry out if any

$$\begin{array}{r} 1100 \\ +\quad 1 \\ \hline 1101_2 \end{array}$$

Part 2 – Add the two codes, ignore carry out if any

$$\begin{array}{r} 6_{10} \\ +\ (-3_{10}) \\ \hline 3_{10} \end{array} \quad = \quad \begin{array}{r} {}^{1}\quad \\ 0\,1\,1\,0_2 \\ +\ 1\,1\,0\,1_2 \\ \hline 1\,0\,0\,1\,1_2 \end{array}$$

The sum resulted in a carry out, but in two's complement addition, this bit is ignored.

The result of the addition was 0011_2 or $+3_{10,}$ verifying that this approach was correct. Also, two's complement overflow did not occur because the result of this operation was within the range of possible values that a 4-bit, two's complement number can represent (e.g., -8_{10} to $+7_{10}$).

Example 2.16
Using two's complement addition to perform subtraction

CONCEPT CHECK

CC2.1.4 A 4-bit, two's complement number has 16 unique codes and can represent decimal numbers between -8_{10} to $+7_{10}$. If the number of unique codes is even, why is it that the range of integers it can represent is not symmetrical about 0?

A) One of the positive codes is used to represent 0. This prevents the highest positive number from reaching $+8_{10}$ and being symmetrical.

B) It is asymmetrical because the system allows the numbers to roll over.

C) It isn't asymmetrical if 0 is considered a positive integer. That way there are eight positive numbers and eight negatives numbers.

D) It is asymmetrical because there are duplicate codes for 0.

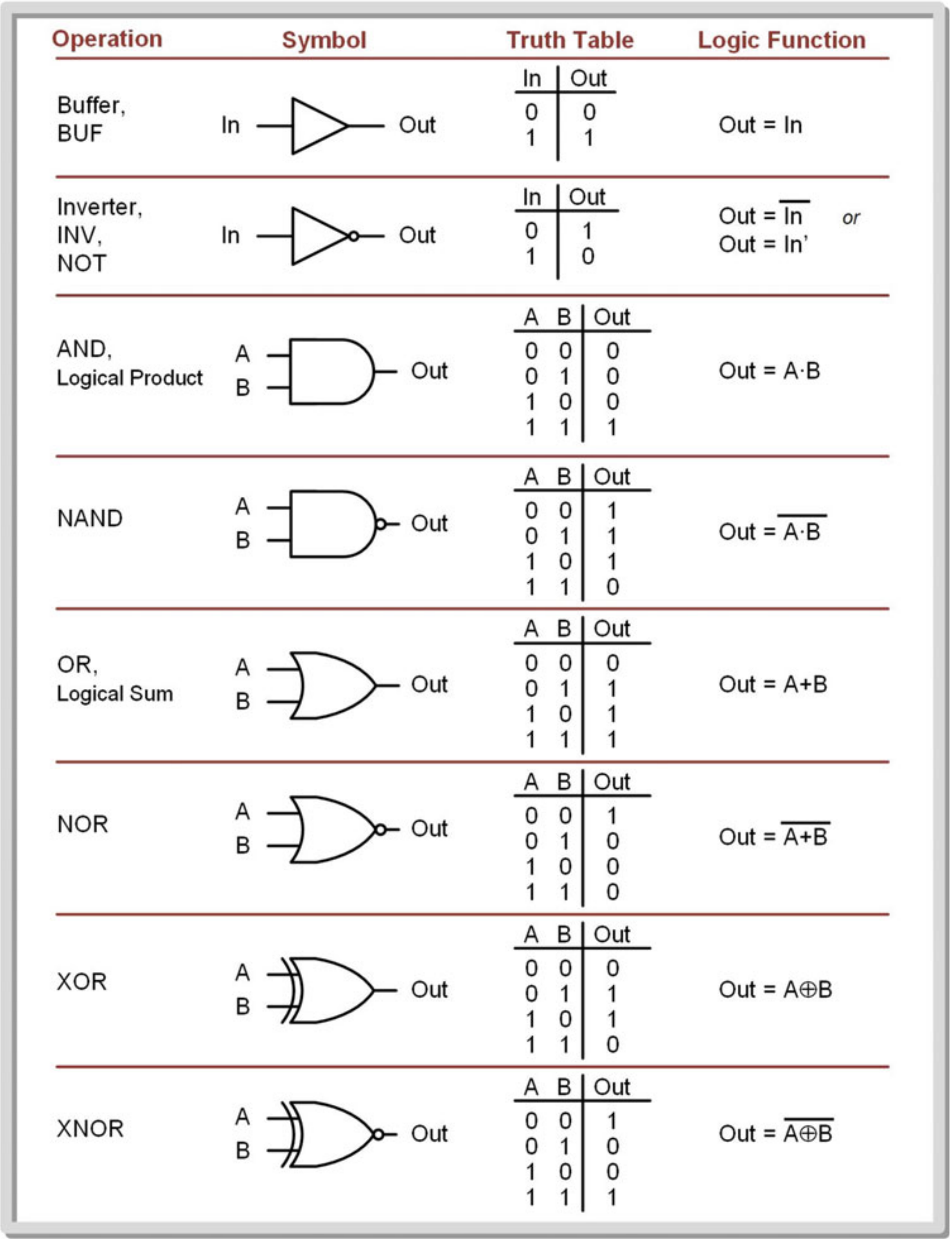
2.2 Combinational Logic

The goal of this section is to provide an understanding of the basic principles of combinational logic design. This includes basic logic functions, Boolean algebra, and common synthesis techniques. The term *combinational logic* refers to circuits where the output depends on the present value(s) of the inputs. This simple definition implies that there is no storage capability in the circuitry and a change on the input immediately impacts the output.

2.2.1 Basic Gates

The term *gate* is used to describe a digital circuit that implements the most basic functions possible on the binary number system. Said another way, the inputs to a basic gate are 0s and 1s and the output is either a 0 or 1. When discussing the operation of a logic gate, we ignore the details of how the 1s and 0s are represented with voltages or currents. We instead treat the inputs and output as simply ideal 1s and 0s. This allows us to design more complex logic circuits without going into the details of the underlying physical hardware. We often use also use the terms *TRUE* and *FALSE* to describe logic states with a TRUE being equivalent to a 1 and FALSE being equivalent to a 0.

There are three common ways to represent the functionality of a basic gate. The first is a unique *symbol*, which can be used in schematics. The second is a *truth table*, in which all possible input values are listed along with the associated output. The third way to represent the gate functionality is with a *logic expression*, which provides a way to express the logic in equation form. Figure 2.6 gives the functionality for the basic gates used in combinational logic.



Operation	Symbol	Truth Table	Logic Function
Buffer, BUF	In → Out	In | Out 0 | 0 1 | 1	Out = In

Inverter,
INV,
NOT
	In → Out	In | Out
		0 | 1
		1 | 0
			Out = $\overline{\text{In}}$ *or*
			Out = In’

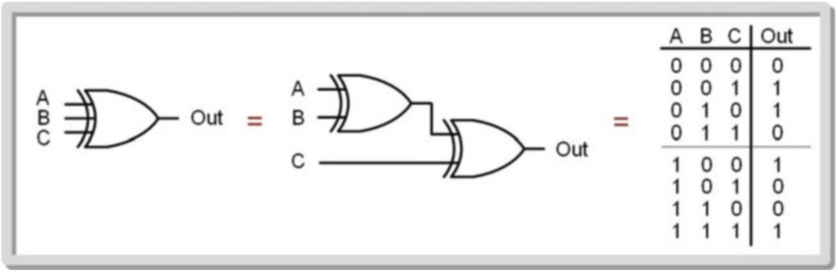
AND, Logical Product	A, B → Out	A B | Out 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1	Out = A·B
NAND	A, B → Out	A B | Out 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0	Out = $\overline{\text{A}\cdot\text{B}}$
OR, Logical Sum	A, B → Out	A B | Out 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1	Out = A+B
NOR	A, B → Out	A B | Out 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0	Out = $\overline{\text{A+B}}$
XOR	A, B → Out	A B | Out 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0	Out = A⊕B
XNOR	A, B → Out	A B | Out 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 1	Out = $\overline{\text{A}\oplus\text{B}}$

Fig. 2.6
Basic gate functionality

It should be noted that each of the 2-input logic operations can be expanded to have a larger number of inputs. For an *n-bit AND* gate (where *n* > 2), the output is a logic 1 only when *all* of the inputs are 1s. A NAND gate is always the inversion of the *n*-bit AND operation, regardless of the number of inputs.

For an *n-bit OR* gate (where *n* > 2), the output is a logic 1 when *any* of the inputs are a 1. A NOR gate is always the inversion of the *n*-bit OR operation, regardless of the number of inputs.

For an *n*-bit XOR gate (where *n* > 2), the functionality is created by cascading 2-input XOR gates. This results in an output that is true when there is *an ODD number of 1s on the inputs*. This functionality is much more useful in modern electronics for error correction codes and arithmetic. As such, this is the functionality that is seen in modern *n*-bit, XOR gates. The 3-bit XOR functionality is also shown in Fig. 2.7. An XNOR gate is always the inversion of the *n*-bit XOR operation, regardless of the number of inputs.

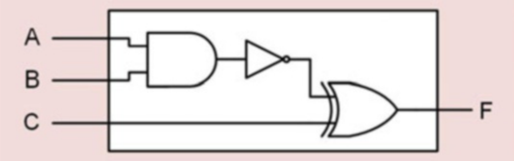


A	B	C	Out
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Fig. 2.7
3-input XOR gate implementation

Concept Check

CC2.2.1 Given the following logic diagram, which is the correct logic expression for F?



A) $F = (A \cdot B)' \oplus C$

B) $F = (A' \cdot B') \oplus C$

C) $F = (A' \cdot B' \oplus C)$

D) $F = A \cdot B' \oplus C$

2.2.2 Boolean Algebra

The term *algebra* refers to the rules of a number system. Algebra defines the operations that are legal to perform on a system. Once we have defined the rules for a system, we can then use the system for more powerful mathematics such as solving for unknowns and manipulating into equivalent forms. The ability to manipulate into equivalent forms allows us to minimize the number of logic operations necessary and also put into a form that can be directly synthesized using modern logic circuits.

In 1854, English mathematician George Boole presented an abstract algebraic framework for a system that contained only two states, true and false. This framework essentially launched the field of computer science even before the existence of the modern integrated circuits that are used to implement digital logic today. In 1930, American mathematician Claude Shannon applied Boole's algebraic framework to his work on switching circuits at Bell Labs, thus launching the field of digital circuit design and information theory. Boole's original framework is still used extensively in modern digital circuit design and thus bears the name *Boolean algebra*. Today, the term Boolean algebra is often used to describe not only George Boole's original work but all of those that contributed to the field after him.

In Boolean algebra there are two valid states (true and false) and three core operations. The operations are conjunction ($\wedge$, equivalent to the AND operation), disjunction ($\vee$, equivalent to the OR operation), and negation ($\neg$, equivalent to the NOT operation). From these three operations, more sophisticated operations can be created including other logic functions (i.e., BUF, NAND, NOR, XOR, and XNOR) and arithmetic. Engineers primarily use the terms AND, OR, and NOT instead of conjunction, disjunction, and negation. Similarly, engineers primarily use the symbols for these operators described in Chap. 3 (e.g. $\cdot$, $+$ and $'$) instead of $\wedge$, $\vee$, and $\neg$.

An *axiom* is a statement of truth about a system that is accepted by the user. Axioms are very simple statements about a system but need to be established before more complicated theorems can be proposed. Axioms are so basic that they do not need to be proved in order to be accepted. Axioms can be thought of as the basic *laws* of the algebraic framework. The terms *axiom* and *postulate* are synonymous and used interchangeably. In Boolean algebra there are five main axioms. These axioms will appear redundant with the description of basic gates in Fig. 2.6 but must be defined in this algebraic context so that more powerful theorems can be proposed.

This axiom states that in Boolean algebra, a variable A can only take on one of two values, 0 or 1. If the variable A is not 0, then it must be a 1, and conversely, if it is not a 1, then it must be a 0.

Axiom #1 – Boolean Values $A = 0$ if $A \neq 1$, conversely $A = 1$ if $A \neq 0$.

This axiom defines logical negation. Negation is also called the NOT operation or taking the *complement*. The negation operation is denoted using either a prime ($'$), an inversion bar, or the negation symbol ($\neg$). If the complement is taken on a 0, it becomes a 1. If the complement is taken on a 1, it becomes a 0.

Axiom #2 – Definition of Logical Negation if $A = 0$ then $A' = 1$, conversely, if $A = 1$ then $A' = 0$.

This axiom defines a logical product or multiplication. Logical multiplication is denoted using either a dot ($\cdot$), an ampersand (&), or the conjunction symbol ($\wedge$). The result of logical multiplication is true when *both* inputs are true and false otherwise.

Axiom #3 – Definition of a Logical Product *$A{\cdot}B = 1$ if $A = B = 1$ and $A{\cdot}B = 0$ otherwise.*

This axiom defines a logical sum or addition. Logical addition is denoted using either a plus sign ($+$) or the disjunction symbol ($\vee$). The result of logical addition is true when *any* of the inputs are true and false otherwise.

Axiom #4 – Definition of a Logical Sum *$A + B = 1$ if $A = 1$ or $B = 1$ and $A + B = 0$ otherwise.*

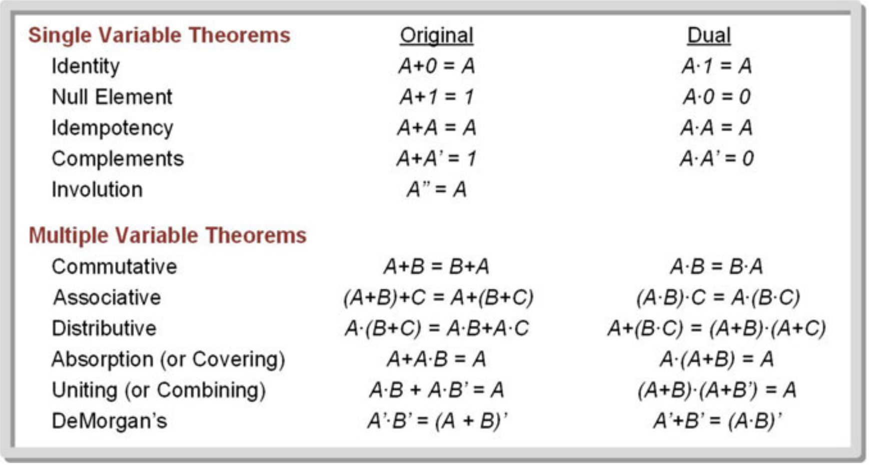
This axiom defines the order of precedence for the three operators. Unless the precedence is explicitly stated using parentheses, negation takes precedence over a logical product and a logical product takes precedence over a logical sum.

Axiom #5 – Definition of Logical Precedence *NOT precedes AND, AND precedes OR.*

To illustrate Axiom #5, consider the logic function $F = A'{\cdot}B + C$. In this function, the first operation that would take place is the NOT operation on A. This would be followed by the AND operation of A$'$ with B. Finally, the result would be OR'd with C. The precedence of any function can also be explicitly stated using parentheses such as $F = (((A') \cdot B) + C)$. In schematic form, the order of operation evaluated moving from the inputs to the output of each gate.

A *theorem* is a more sophisticated truth about a system that is not intuitively obvious. Theorems are proposed and then must be proved. Once proved, they can be accepted as a truth about the system going forward. Proving a theorem in Boolean algebra is much simpler than in our traditional decimal system due to the fact that variables can only take on one of two values, true or false. Since the number of input possibilities is bounded, Boolean algebra theorems can be proved by simply testing the theorem using every possible input code. This is called *proof by exhaustion*.

Table 2.3 gives a summary of the most common Boolean algebra theorems. The theorems are grouped in this table with respect to the number of variables that they contain. For each theorem, the original form in given in addition to the *dual*. *Duality* is its own theorem that says a logic expression will remain true if all 1s and 0s are interchanged and all AND and OR operations are interchanged. The dual provides an additional set of theorems for each original form.



Single Variable Theorems	Original	Dual
Identity	$A+0 = A$	$A\cdot 1 = A$
Null Element	$A+1 = 1$	$A\cdot 0 = 0$
Idempotency	$A+A = A$	$A\cdot A = A$
Complements	$A+A' = 1$	$A\cdot A' = 0$
Involution	$A'' = A$	
Multiple Variable Theorems		
Commutative	$A+B = B+A$	$A\cdot B = B\cdot A$
Associative	$(A+B)+C = A+(B+C)$	$(A\cdot B)\cdot C = A\cdot(B\cdot C)$
Distributive	$A\cdot(B+C) = A\cdot B+A\cdot C$	$A+(B\cdot C) = (A+B)\cdot(A+C)$
Absorption (or Covering)	$A+A\cdot B = A$	$A\cdot(A+B) = A$
Uniting (or Combining)	$A\cdot B + A\cdot B' = A$	$(A+B)\cdot(A+B') = A$
DeMorgan's	$A'\cdot B' = (A + B)'$	$A'+B' = (A\cdot B)'$

Table 2.3
Summary of Boolean algebra theorems

Concept Check

CC2.2.2 If the logic expression $F = A\cdot B\cdot C\cdot D\cdot E\cdot F\cdot G\cdot H$ is implemented with only 2-input AND gates, how many AND gates will it take? Hint: You can consider using the associative property to manipulate the logic expression to use only 2-input AND operations, or, you can sketch out a logic schematic using 2-input AND gates.

A) 4

B) 6

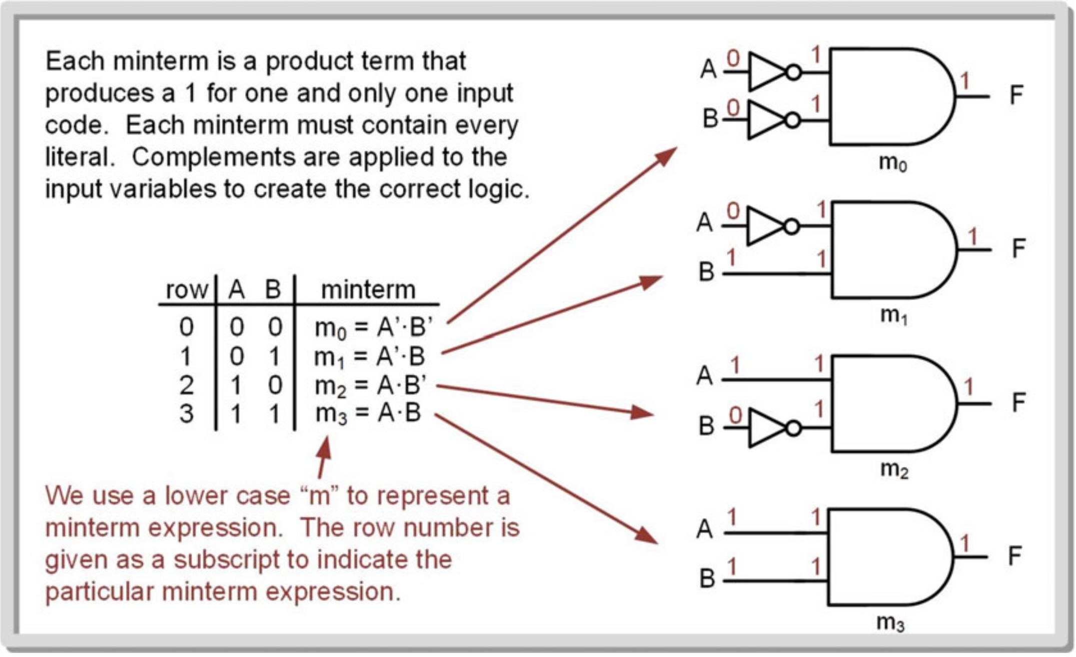
C) 7

D) 8

2.2.3 Combinational Logic Synthesis

2.2.3.1 Canonical Sum of Products

One technique to directly synthesize a logic circuit from a truth table is to use a canonical sum of products (SOP) topology based on *minterms.* The term *canonical* refers to this topology yielding potentially unminimized logic. A minterm is a product term (i.e., an AND operation) that will be true for one and only one input code. The minterm must contain every input variable in its expression. Complements are applied to the input variables as necessary in order to produce a true output for the individual input code. We define the word *literal* to describe an input variable which may or may not be complemented. This is a more useful word because if we say that a minterm "must include all variables," it implies that all variables are included in the term uncomplemented. A more useful statement is that a minterm "must include all literals." This now implies that each variable must be included, but it can be in the form of itself or its complement (e.g., A or A′). Figure 2.8 shows the definition and gate level depiction of a minterm expression. Each minterm can be denoted using the lower case "m" with the row number as a subscript.



Each minterm is a product term that produces a 1 for one and only one input code. Each minterm must contain every literal. Complements are applied to the input variables to create the correct logic.

row	A	B	minterm
0	0	0	$m_0 = A'\cdot B'$
1	0	1	$m_1 = A'\cdot B$
2	1	0	$m_2 = A\cdot B'$
3	1	1	$m_3 = A\cdot B$

We use a lower case "m" to represent a minterm expression. The row number is given as a subscript to indicate the particular minterm expression.

Fig. 2.8
Definition and gate level depiction of a minterm

For an arbitrary truth table, a minterm can be used for each row corresponding to a true output. If each of these minterms' outputs are fed into a single OR gate, then a sum of products logic circuit is formed that will produce the logic listed in the truth table. In this topology, any input code that corresponds to an output of 1 will cause its corresponding minterm to output a 1. Since a 1 on any input of an OR gate will cause the output to go to a 1, the output of the minterm is passed to the final result. Example 2.17 shows this process. One important consideration of this approach is that no effort has been taken to minimize the logic expression. This unminimized logic expression is also called the *canonical sum*. The canonical sum is logically correct but uses the most amount of circuitry possible for a given truth table. This canonical sum can be the starting point for minimization using Boolean algebra.

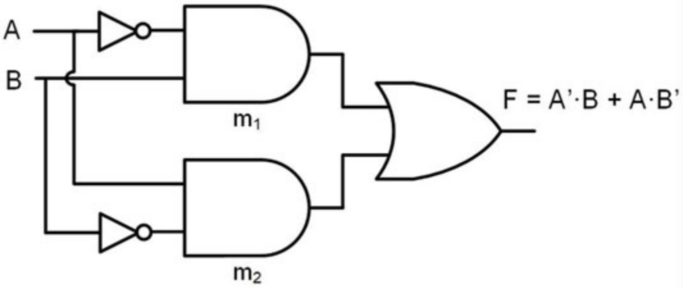
Example: Creating a Canonical SOP Circuit using Minterms

For the following truth table, find the canonical sum of products logic expression & logic circuit.

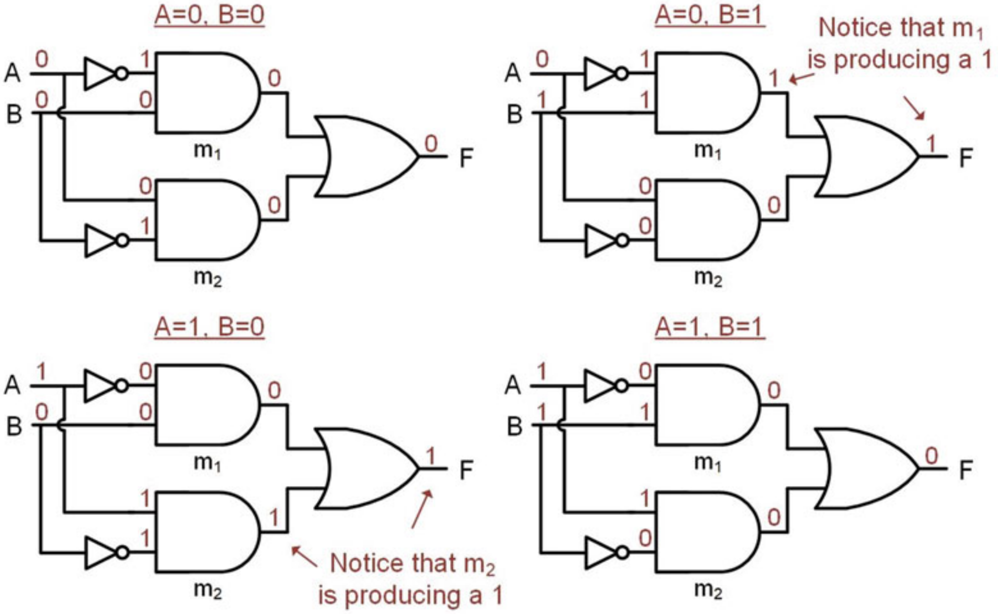
A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

Let's first start by writing the minterms for the rows that correspond to a 1 on the output. These can then be implemented using inverters and AND gates. The final step is to feed the outputs of each minterm circuit into a single OR gate.

row	A	B	minterm
0	0	0	-
1	0	1	$m_1 = A'{\cdot}B$
2	1	0	$m_2 = A{\cdot}B'$
3	1	1	-



Let's now check that this circuit performs as intended by testing it under each input code for A and B and observing the output F.

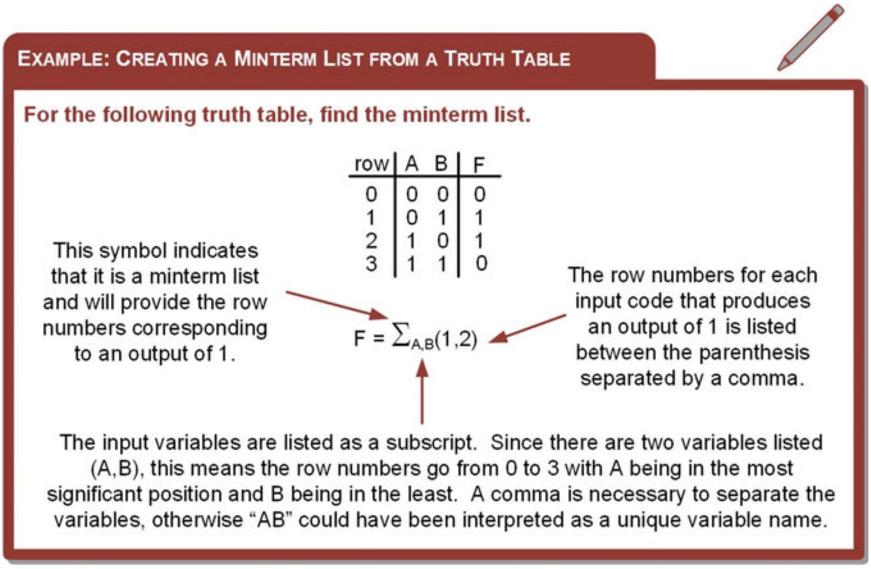


Notice that m_1 is producing a 1

Notice that m_2 is producing a 1

Example 2.17
Creating a canonical sum of products logic circuit using minterms

A *minterm list* is a compact way to describe the functionality of a logic circuit by simply listing the row numbers that correspond to an output of 1 in the truth table. The ∑ symbol is used to denote a minterm list. All input variables must be listed in the order they appear in the truth table. This is necessary because since a minterm list uses only the row numbers to indicate which input codes result in an output of 1, the minterm list must indicate how many variables comprise the row number, which variable is in the most significant position, and which is in the least significant position. After the ∑ symbol, the row numbers corresponding to a true output are listed in a comma-delimited format within parentheses. Example 2.18 shows the process for creating a minterm list from a truth table.



EXAMPLE: CREATING A MINTERM LIST FROM A TRUTH TABLE

For the following truth table, find the minterm list.

row	A	B	F
0	0	0	0
1	0	1	1
2	1	0	1
3	1	1	0

This symbol indicates that it is a minterm list and will provide the row numbers corresponding to an output of 1.

$$F = \Sigma_{A,B}(1,2)$$

The row numbers for each input code that produces an output of 1 is listed between the parenthesis separated by a comma.

The input variables are listed as a subscript. Since there are two variables listed (A,B), this means the row numbers go from 0 to 3 with A being in the most significant position and B being in the least. A comma is necessary to separate the variables, otherwise "AB" could have been interpreted as a unique variable name.

Example 2.18
Creating a minterm list from a truth table

A minterm list contains the same information as the truth table, the canonical sum, and the canonical sum of products logic diagram. Since the minterms themselves are formally defined for an input code, it is trivial to go back and forth between the minterm list and these other forms. Example 2.19 shows how a minterm list can be used to generate an equivalent truth table, canonical sum, and canonical sum of products logic diagram.

Example: Creating Equivalent Forms from a Minterm List

For the following minterm list:

$F = \sum_{A,B,C}(0,3,7)$

(a) find the truth table;
(b) find the canonical sum of products logic expression;
(c) find the canonical sum of products logic diagram.

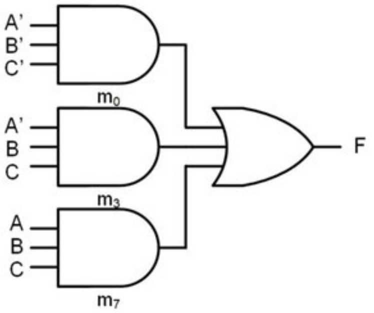
(a) Truth Table: From the minterm list subscripts, we know that there are three input variables named A, B and C. These will be listed in the truth table with A in the most significant position and C in the least significant position. We can fill in the input codes as a binary count and insert the row numbers. We can then list the output values that are true. From the minterm list we know that the true outputs are on rows 0, 3 and 7. Since we know we will need the minterm expressions for these rows in the canonical sum, we can also list them in the truth table.

row	A	B	C	F	minterm
0	0	0	0	1	$m_0 = A'\cdot B'\cdot C'$
1	0	0	1	0	-
2	0	1	0	0	-
3	0	1	1	1	$m_3 = A'\cdot B\cdot C$
4	1	0	0	0	-
5	1	0	1	0	-
6	1	1	0	0	-
7	1	1	1	1	$m_7 = A\cdot B\cdot C$

(b) Canonical SOP Logic Expression: This is simply the minterm expressions corresponding to a true output OR'd together. Since we already wrote the minterm expressions for rows 0, 3 and 7 (e.g., m_0, m_3 and m_7) in the truth table, we can write the canonical sum directly.

$$F = A'\cdot B'\cdot C' + A'\cdot B\cdot C + A\cdot B\cdot C$$

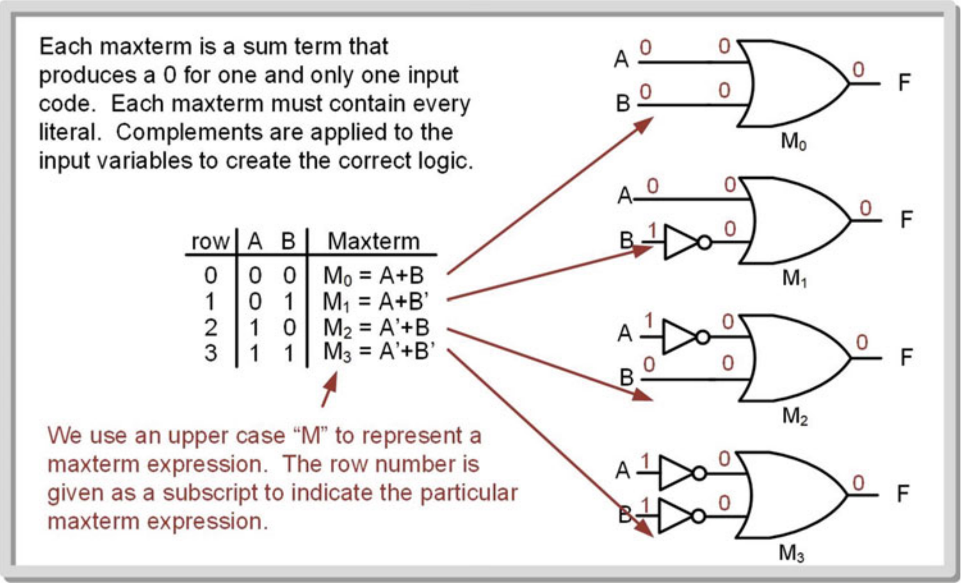
(c) Canonical SOP Logic Diagram: This is simply the gate level depiction of the canonical sum. When logic diagrams get larger, it is acceptable to indicate a variable's complement as a prime instead of placing individual inverters and drawing connection wires that cross each other. It is implied that multiple listings of a variable's complement (e.g., A' in m_0 and m_3) will come from the same inverter.



Example 2.19
Creating equivalent functional representations from a minterm list

2.2.3.2 Canonical Product of Sums

Another technique to directly synthesize a logic circuit from a truth table is to use a canonical product of sums (POS) topology based on *maxterms.* A maxterm is a sum term (i.e., an OR operation) that will be false for one and only one input code. The maxterm must contain every literal in its expression. Complements are applied to the input variables as necessary in order to produce a false output for the individual input code. Figure 2.9 shows the definition and gate level depiction of a maxterm expression. Each maxterm can be denoted using the upper case "M" with the row number as a subscript.



Each maxterm is a sum term that produces a 0 for one and only one input code. Each maxterm must contain every literal. Complements are applied to the input variables to create the correct logic.

row	A	B	Maxterm
0	0	0	M_0 = A+B
1	0	1	M_1 = A+B'
2	1	0	M_2 = A'+B
3	1	1	M_3 = A'+B'

We use an upper case "M" to represent a maxterm expression. The row number is given as a subscript to indicate the particular maxterm expression.

Fig. 2.9
Definition and gate level depiction of a maxterm

For an arbitrary truth table, a maxterm can be used for each row corresponding to a false output. If each of these maxterms outputs are fed into a single AND gate, then a product of sums logic circuit is formed that will produce the logic listed in the truth table. In this topology, any input code that corresponds to an output of 0 will cause its corresponding maxterm to output a 0. Since a 0 on any input of an AND gate will cause the output to go to a 0, the output of the maxterm is passed to the final result. Example 2.20 shows this process. This approach is complementary to the sum of products approach. In the sum of products approach based on minterms, the circuit operates by producing 1s that are passed to the output for the rows that require a true output. For all other rows, the output is false. A product of sums approach based on maxterms operates by producing 0s that are passed to the output for the rows that require a false output. For all other rows, the output is true. These two approaches produce the equivalent logic functionality. Again, at this point no effort has been taken to minimize the logic expression. This unminimized form is called a *canonical product*. The canonical product is logically correct but uses the most amount of circuitry possible for a given truth table. This canonical product can be the starting point for minimization using the Boolean algebra theorems.

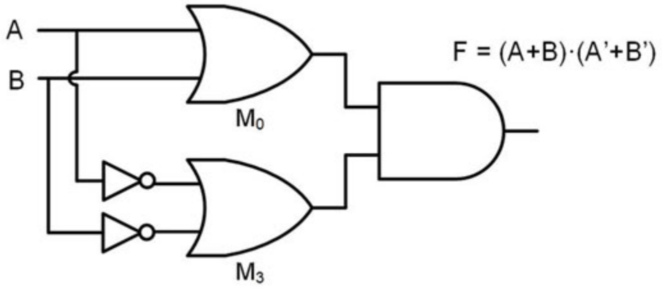
Example: Creating a Canonical POS Circuit using Maxterms

For the following truth table, find the canonical product of sums logic expression & logic circuit.

A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

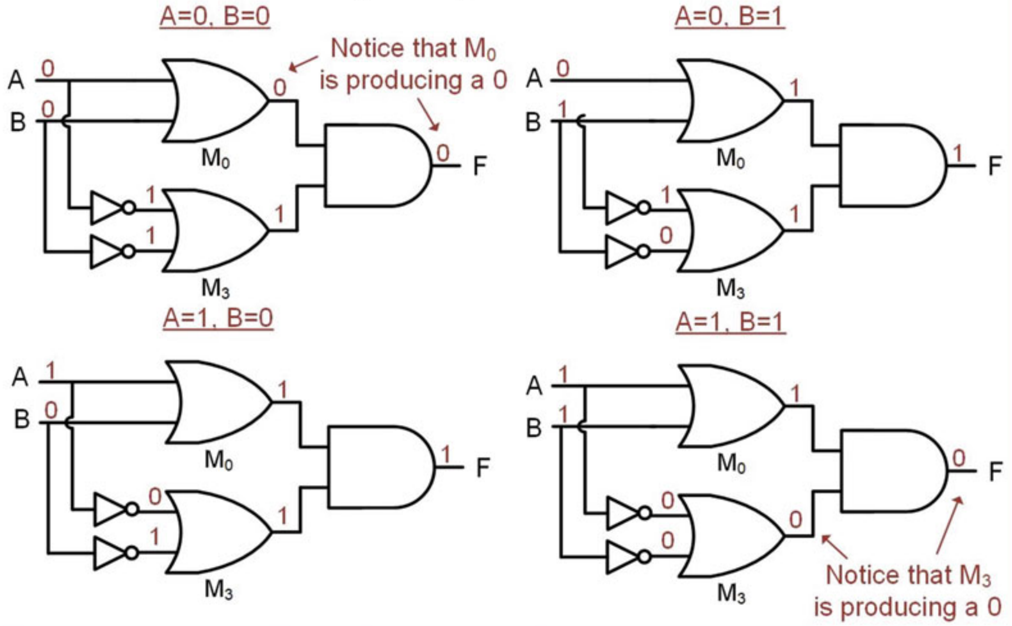
Let's first start by writing the maxterms for the rows that correspond to a 0 on the output. These can then be implemented using inverters and OR gates. The final step is to feed the outputs of each maxterm circuit into a single AND gate.

row	A	B	Maxterm
0	0	0	M_0=A+B
1	0	1	-
2	1	0	-
3	1	1	M_3=A'+B'



F = (A+B)·(A'+B')

Let's now check that this circuit performs as intended by testing it under each input code for A and B and observing the output F.


A=0, B=0

A=0, B=1

A=1, B=0

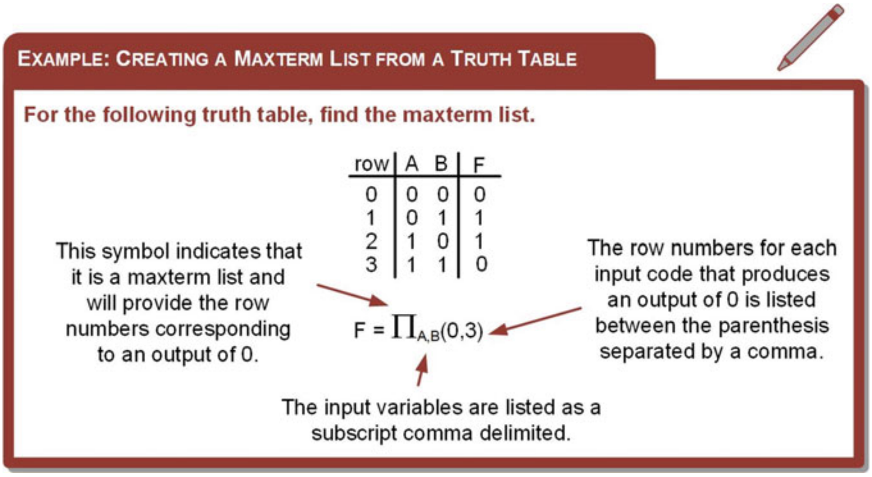
A=1, B=1

Notice that M_0 is producing a 0

Notice that M_3 is producing a 0

Example 2.20
Creating a product of sums logic circuit using maxterms

A *maxterm list* is a compact way to describe the functionality of a logic circuit by simply listing the row numbers that correspond to an output of 0 in the truth table. The Π symbol is used to denote a maxterm list. All literals used in the logic expression must be listed in the order they appear in the truth table. After the Π symbol, the row numbers corresponding to a false output are listed in a comma-delimited format within parentheses. Example 2.21 shows the process for creating a maxterm list from a truth table.



EXAMPLE: CREATING A MAXTERM LIST FROM A TRUTH TABLE

For the following truth table, find the maxterm list.

row	A	B	F
0	0	0	0
1	0	1	1
2	1	0	1
3	1	1	0

$F = \prod_{A,B}(0,3)$

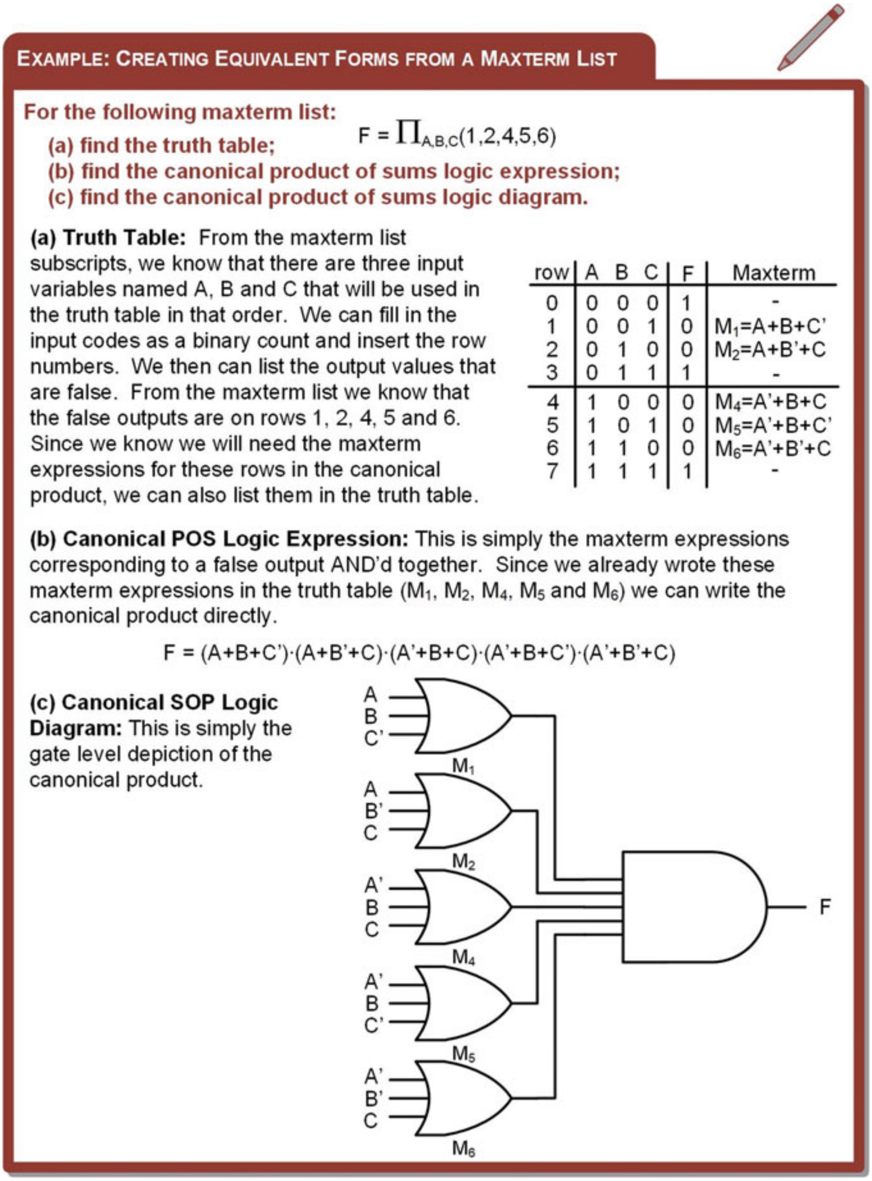
This symbol indicates that it is a maxterm list and will provide the row numbers corresponding to an output of 0.

The row numbers for each input code that produces an output of 0 is listed between the parenthesis separated by a comma.

The input variables are listed as a subscript comma delimited.

Example 2.21
Creating a maxterm list from a truth table

A maxterm list contains the same information as the truth table, the canonical product, and the canonical product of sums logic diagram. Example 2.22 shows how a maxterm list can be used to generate these equivalent forms.



EXAMPLE: CREATING EQUIVALENT FORMS FROM A MAXTERM LIST

For the following maxterm list: $F = \prod_{A,B,C}(1,2,4,5,6)$

(a) find the truth table;

(b) find the canonical product of sums logic expression;

(c) find the canonical product of sums logic diagram.

(a) Truth Table: From the maxterm list subscripts, we know that there are three input variables named A, B and C that will be used in the truth table in that order. We can fill in the input codes as a binary count and insert the row numbers. We then can list the output values that are false. From the maxterm list we know that the false outputs are on rows 1, 2, 4, 5 and 6. Since we know we will need the maxterm expressions for these rows in the canonical product, we can also list them in the truth table.

row	A	B	C	F	Maxterm
0	0	0	0	1	-
1	0	0	1	0	M_1=A+B+C’
2	0	1	0	0	M_2=A+B’+C
3	0	1	1	1	-
4	1	0	0	0	M_4=A’+B+C
5	1	0	1	0	M_5=A’+B+C’
6	1	1	0	0	M_6=A’+B’+C
7	1	1	1	1	-

(b) Canonical POS Logic Expression: This is simply the maxterm expressions corresponding to a false output AND’d together. Since we already wrote these maxterm expressions in the truth table (M_1, M_2, M_4, M_5 and M_6) we can write the canonical product directly.

F = (A+B+C’)·(A+B’+C)·(A’+B+C)·(A’+B+C’)·(A’+B’+C)

(c) Canonical SOP Logic Diagram: This is simply the gate level depiction of the canonical product.

Example 2.22
Creating equivalent functional representations from a maxterm list

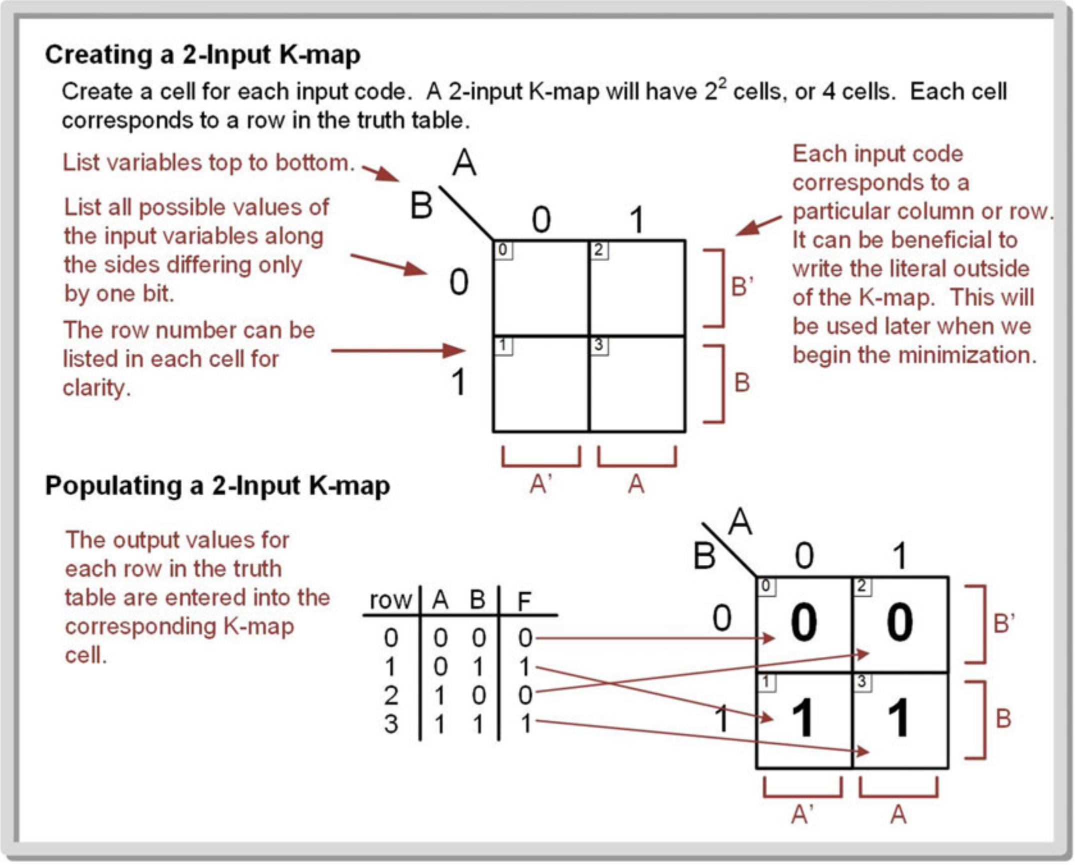
The examples in Examples 2.19 and 2.22 illustrate how minterm and maxterm lists produce the exact same logic functionality but in a complementary fashion.

2.2.3.3 Logic Minimization in SOP Form

We now look at how to reduce the canonical expressions into equivalent forms that use less logic. This minimization is key to reducing the complexity of the logic prior to implementing in real circuitry. This reduces the number of gates needed, placement area, wiring, and power consumption of the logic circuit.

One of the earliest ways to manually minimize a logic expression was using a Karnaugh map, or K-map. A K-map is a graphical way to minimize logic expressions. This technique is named after Maurice Karnaugh, American physicist, who introduced the map in its latest form in 1953 while working at Bell Labs. The K-map is a way to put a truth table into a form that allows logic minimization through a graphical process. This technique provides a graphical process that accomplishes the same result as factoring variables via the distributive property and removing variables via the Complements and Identity Theorems. K-maps present a truth table in a form that allows variables to be removed from the final logic expression in a graphical manner.

A K-map is constructed as a two-dimensional grid. Each *cell* within the map corresponds to the output for a specific input code. The cells are positioned such that neighboring cells only differ by one bit in their input codes. Neighboring cells are defined as cells immediately adjacent horizontally and immediately adjacent vertically. Two cells positioned diagonally next to each other are not considered neighbors. The input codes for each variable are listed along the top and side of the K-map. Consider the construction of a 2-input K-map shown in Fig. 2.10.



Creating a 2-Input K-map

Create a cell for each input code. A 2-input K-map will have 2^2 cells, or 4 cells. Each cell corresponds to a row in the truth table.

List variables top to bottom.

List all possible values of the input variables along the sides differing only by one bit.

The row number can be listed in each cell for clarity.

Each input code corresponds to a particular column or row. It can be beneficial to write the literal outside of the K-map. This will be used later when we begin the minimization.

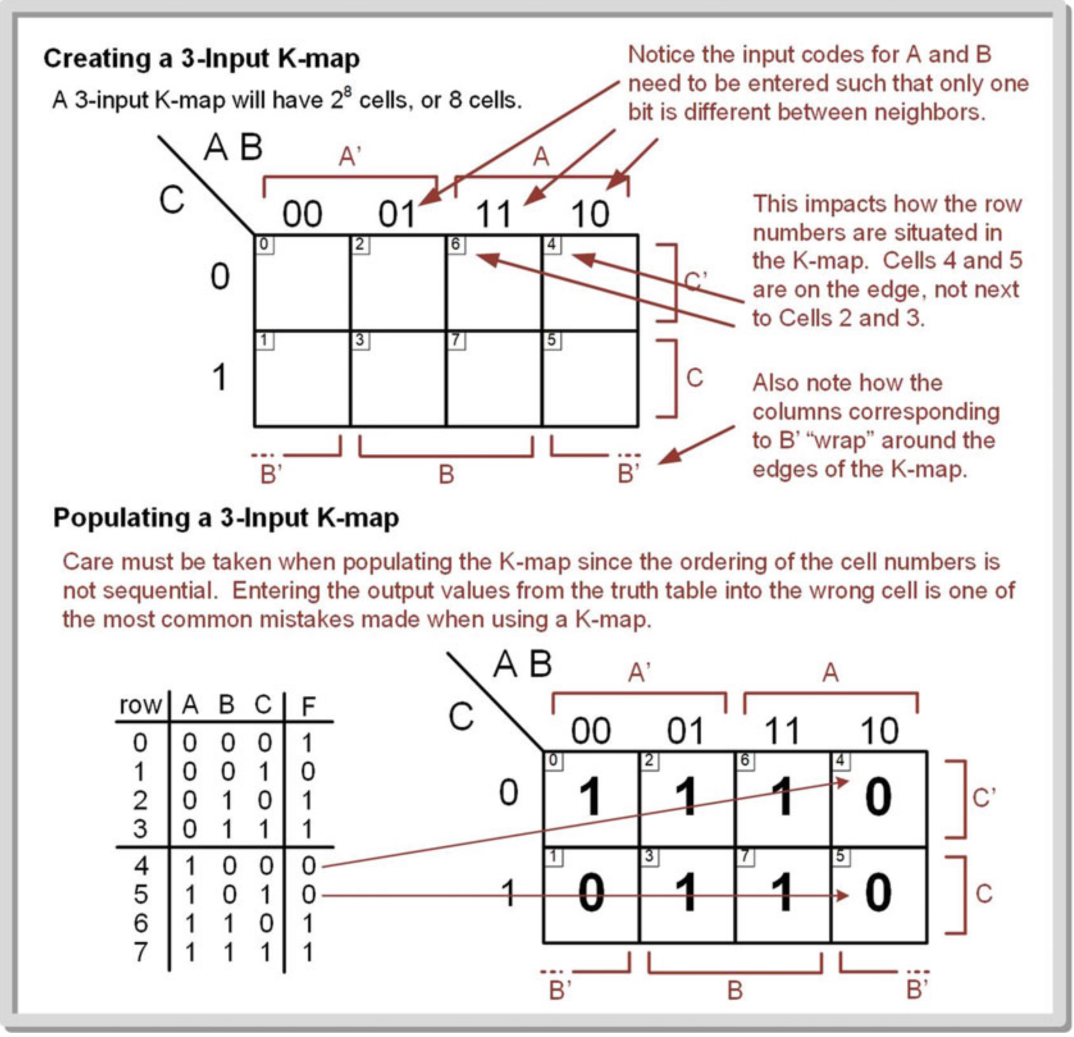
Populating a 2-Input K-map

The output values for each row in the truth table are entered into the corresponding K-map cell.

row	A	B	F
0	0	0	0
1	0	1	1
2	1	0	0
3	1	1	1

Fig. 2.10
Formation of a 2-input K-map

When constructing a 3-input K-map, it is important to remember that each input code can only differ from its neighbor by 1 bit. For example, the two codes 01 and 10 differ by two bits (i.e., the MSB is different and the LSB is different); thus, they could not be neighbors; however, the codes 01-11 and 11-10 can be neighbors. As such, the input codes along the top of the 3-input K-map must be ordered accordingly (i.e., 00-01-11-10). Consider the construction of a 3-input K-map shown in Fig. 2.11. The rows and columns that correspond to the input literals can now span multiple rows and columns. Notice how in this 3-input K-map, the literals A, A′, B, and B′ all correspond to two columns. Also, notice that B′ spans two columns, but the columns are on different edges of the K-map. The side edges of the 3-input K-map are still considered neighbors because the input codes for these columns only differ by one bit. This is an important attribute once we get to the minimization of variables because it allows us to examine an input literal's impact not only within the obvious adjacent cells but also when the variables *wrap* around the edges of the K-map.



Creating a 3-Input K-map

A 3-input K-map will have 2^8 cells, or 8 cells.

Notice the input codes for A and B need to be entered such that only one bit is different between neighbors.

This impacts how the row numbers are situated in the K-map. Cells 4 and 5 are on the edge, not next to Cells 2 and 3.

Also note how the columns corresponding to B' "wrap" around the edges of the K-map.

Populating a 3-Input K-map

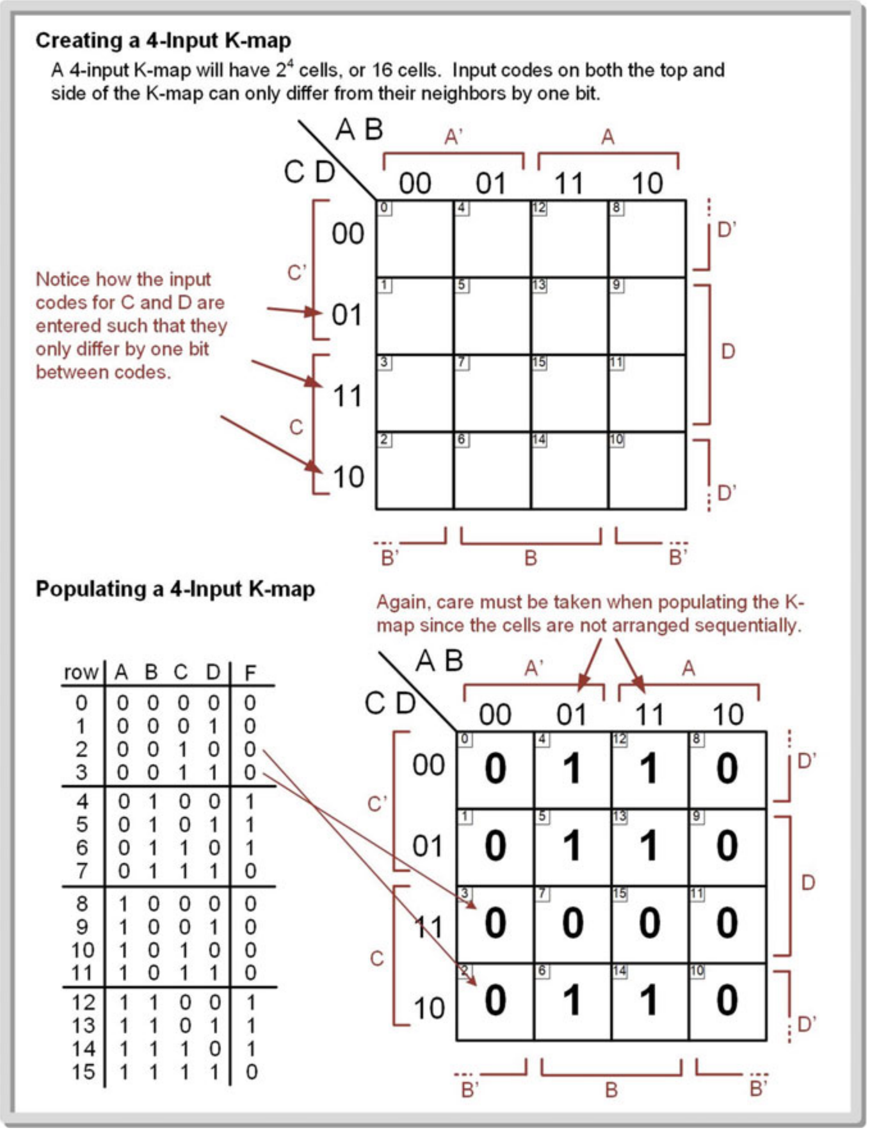
Care must be taken when populating the K-map since the ordering of the cell numbers is not sequential. Entering the output values from the truth table into the wrong cell is one of the most common mistakes made when using a K-map.

row	A	B	C	F
0	0	0	0	1
1	0	0	1	0
2	0	1	0	1
3	0	1	1	1
4	1	0	0	0
5	1	0	1	0
6	1	1	0	1
7	1	1	1	1

C \ AB	00	01	11	10
0	1	1	1	0
1	0	1	1	0

Fig. 2.11
Formation of a 3-input K-map

When constructing a 4-input K-map, the same rules apply that the input codes can only differ from their neighbors by one bit. Consider the construction of a 4-input K-map in Fig. 2.12. In a 4-input K-map, neighboring cells can wrap around both the top-to-bottom edges in addition to the side-to-side edges. Notice that all 16 cells are positioned within the map so that their neighbors on the top, bottom, and sides only differ by one bit in their input codes.



Creating a 4-Input K-map

A 4-input K-map will have 2^4 cells, or 16 cells. Input codes on both the top and side of the K-map can only differ from their neighbors by one bit.

Notice how the input codes for C and D are entered such that they only differ by one bit between codes.

CD \ AB	00	01	11	10
00	0	4	12	8
01	1	5	13	9
11	3	7	15	11
10	2	6	14	10

Populating a 4-Input K-map

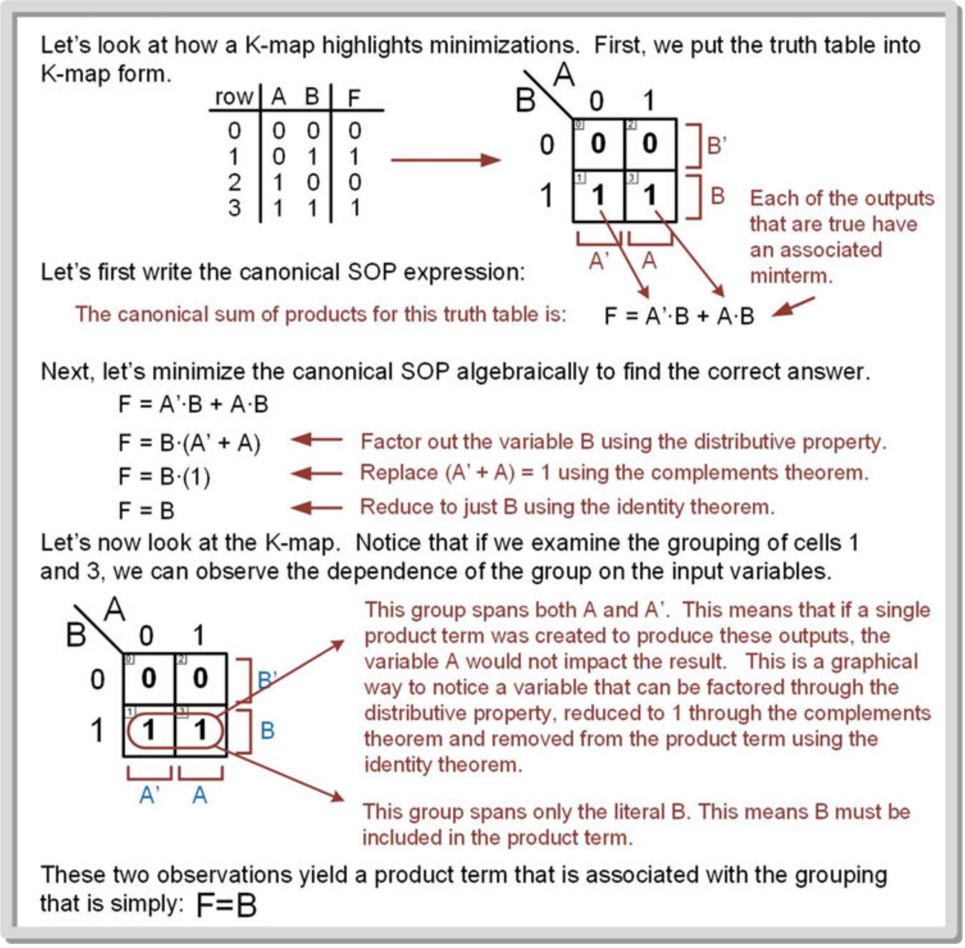
Again, care must be taken when populating the K-map since the cells are not arranged sequentially.

row	A	B	C	D	F
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	1
5	0	1	0	1	1
6	0	1	1	0	1
7	0	1	1	1	0
8	1	0	0	0	0
9	1	0	0	1	0
10	1	0	1	0	0
11	1	0	1	1	0
12	1	1	0	0	1
13	1	1	0	1	1
14	1	1	1	0	1
15	1	1	1	1	0

CD \ AB	00	01	11	10
00	0	1	1	0
01	0	1	1	0
11	0	0	0	0
10	0	1	1	0

Fig. 2.12
Formation of a 4-input K-map

Now we look at using a K-map to create a minimized logic expression in a SOP form. Remember that each cell with an output of 1 has a minterm associated with it, just as in the truth table. When two neighboring cells have outputs of 1, it graphically indicates that the two minterms can be reduced into a minimized product term that will cover both outputs. Consider the example given in Fig. 2.13.



Let's look at how a K-map highlights minimizations. First, we put the truth table into K-map form.

row	A	B	F
0	0	0	0
1	0	1	1
2	1	0	0
3	1	1	1

Each of the outputs that are true have an associated minterm.

Let's first write the canonical SOP expression:

The canonical sum of products for this truth table is: $F = A' \cdot B + A \cdot B$

Next, let's minimize the canonical SOP algebraically to find the correct answer.

$F = A' \cdot B + A \cdot B$

$F = B \cdot (A' + A)$ ← Factor out the variable B using the distributive property.

$F = B \cdot (1)$ ← Replace $(A' + A) = 1$ using the complements theorem.

$F = B$ ← Reduce to just B using the identity theorem.

Let's now look at the K-map. Notice that if we examine the grouping of cells 1 and 3, we can observe the dependence of the group on the input variables.

This group spans both A and A'. This means that if a single product term was created to produce these outputs, the variable A would not impact the result. This is a graphical way to notice a variable that can be factored through the distributive property, reduced to 1 through the complements theorem and removed from the product term using the identity theorem.

This group spans only the literal B. This means B must be included in the product term.

These two observations yield a product term that is associated with the grouping that is simply: $F=B$

Fig. 2.13
Observing how K-maps visually highlight logic minimizations

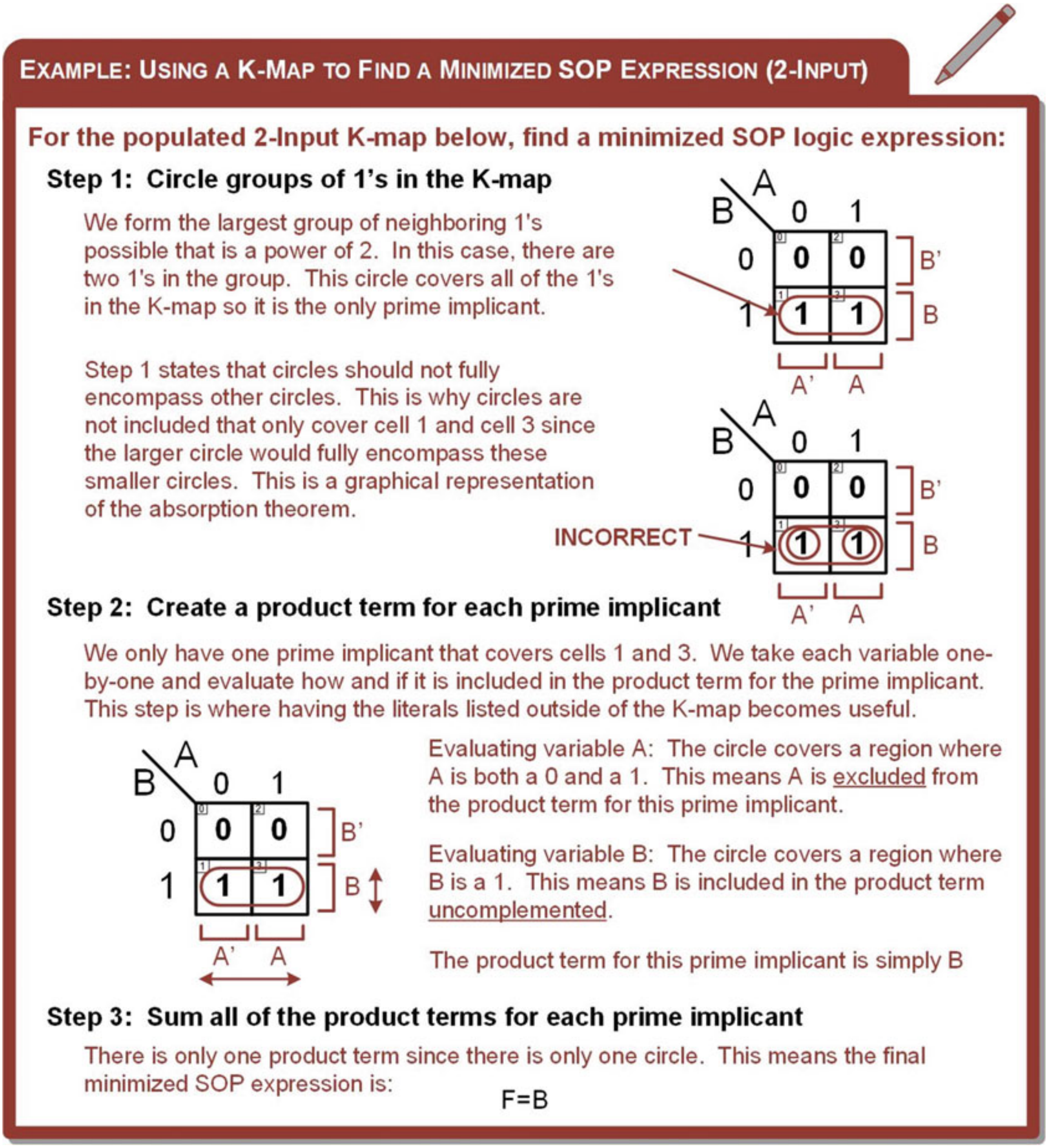
These observations can be put into a formal process to produce a minimized SOP logic expression using a K-map. The steps are as follows:

1. Circle groups of 1s in the K-map following the rules:
   - Each circle should contain the largest number of 1s possible.
   - The circles encompass only neighboring cells (i.e., side-to-side sides and/or top and bottom).
   - The circles must contain a number of 1s that is a power of 2 (i.e., 1, 2, 4, 8, or 16).
   - Enter as many circles as possible without having any circles fully cover another circle.
   - Each circle is called a *prime implicant*.
2. Create a product term for each prime implicant following the rules:
   - Each variable in the K-map is evaluated one by one.
   - If the circle covers a region where the input variable is a 1, then include it in the product term *uncomplemented.*

- If the circle covers a region where the input variable is a 0, then include it in the product term *complemented*.
- If the circle covers a region where the input variable is both a 0 and 1, then the variable is *excluded* from the product term.

3. Sum all of the product terms for each prime implicant.

Let's apply this approach to our 2-input K-map example. Example 2.23 shows the process of finding a minimized sum of products logic expression for a 2-input logic circuit using a K-map. This process yielded the same SOP expression as the algebraic minimization and observations shown in Fig. 2.13, but with a formalized process.



EXAMPLE: USING A K-MAP TO FIND A MINIMIZED SOP EXPRESSION (2-INPUT)

For the populated 2-Input K-map below, find a minimized SOP logic expression:

Step 1: Circle groups of 1's in the K-map

We form the largest group of neighboring 1's possible that is a power of 2. In this case, there are two 1's in the group. This circle covers all of the 1's in the K-map so it is the only prime implicant.

Step 1 states that circles should not fully encompass other circles. This is why circles are not included that only cover cell 1 and cell 3 since the larger circle would fully encompass these smaller circles. This is a graphical representation of the absorption theorem.

INCORRECT

Step 2: Create a product term for each prime implicant

We only have one prime implicant that covers cells 1 and 3. We take each variable one-by-one and evaluate how and if it is included in the product term for the prime implicant. This step is where having the literals listed outside of the K-map becomes useful.

Evaluating variable A: The circle covers a region where A is both a 0 and a 1. This means A is excluded from the product term for this prime implicant.

Evaluating variable B: The circle covers a region where B is a 1. This means B is included in the product term uncomplemented.

The product term for this prime implicant is simply B

Step 3: Sum all of the product terms for each prime implicant

There is only one product term since there is only one circle. This means the final minimized SOP expression is:

F=B

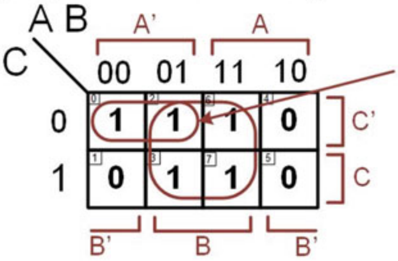
Example 2.23
Using a K-map to find a minimized sum of products expression (2-input)

Let's now apply this process to our 3-input K-map example. Example 2.24 shows the process of finding a minimized sum of products logic expression for a 3-input logic circuit using a K-map. This example shows circles that overlap. This is legal as long as one circle does not fully encompass another. Overlapping circles are common since the K-map process dictates that circles should be drawn that group the largest number of 1s possible as long as they are in powers of 2. Forming groups of 1s using 1s that have already been circled is perfectly legal to accomplish larger groupings. The larger the grouping of 1s, the more chance there is for a variable to be excluded from the product term. This results in better minimization of the logic.

Example: Using a K-Map to Find a Minimized SOP Expression (3-Input)

For the populated 3-Input K-map below, find a minimized SOP logic expression:

Step 1: Circle groups of 1's in the K-map



The two prime implicants overlap in cell 2, but this is legal because the larger circle does not fully encompass the smaller circle.

Step 2: Create a product term for each prime implicant

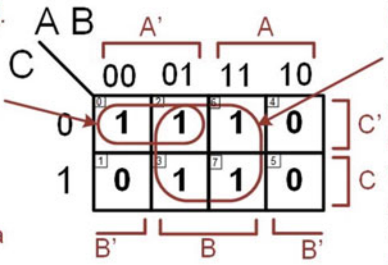
First Prime Implicant

Variable A: The circle covers a region where A is a 0 so it is included in the product term complemented.

Variable B: The circle covers a region where B is both a 0 and 1, so it is excluded from the product term.

Variable C: The circle covers a region where C is a 0, so it is included in the product term complemented.

The product term for this prime implicant is: A'·C'



Second Prime Implicant

Variable A: The circle covers a region where A is both a 0 and 1, so it is excluded from the product term.

Variable B: The circle covers a region where B is a 1, so it is included in the product term uncomplemented.

Variable C: The circle covers a region where C is both a 0 and 1, so it is excluded from the product term.

The product term for this prime implicant is: B

Step 3: Sum all of the product terms for each prime implicant

There are two product terms, one for each circle. The final minimized SOP expression is:

F = A'·C' + B

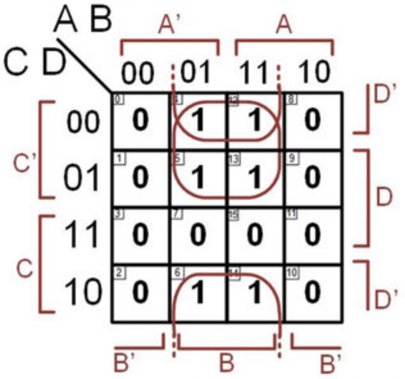
Example 2.24
Using a K-map to find a minimized sum of products expression (3-input)

Let's now apply this process to our 4-input K-map example. Example 2.25 shows the process of finding a minimized sum of products logic expression for a 4-input logic circuit using a K-map.

Example: Using a K-Map to Find a Minimized SOP Expression (4-Input)

For the populated 4-Input K-map below, find a minimized SOP logic expression:

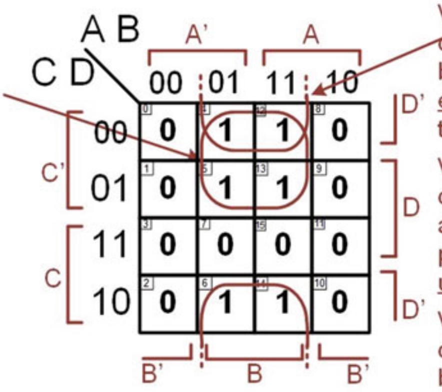
Step 1: Circle groups of 1's in the K-map



Circles can be drawn that "wrap" around the edges. Notice that the input codes for cells 4 and 12 only differ by 1 bit from cells 6 and 14. This makes them neighbors and grouping these 4 cells together is legal.

Again, circles that overlap are legal as long as one circle does not fully encompass another.

Step 2: Create a product term for each prime implicant



First Prime Implicant

Variable A: The circle covers a region where A is both a 0 and 1, so it is excluded from the product term.

Variable B: The circle covers a region where B is a 1, so it is included in the product term uncomplemented.

Variable C: The circle covers a region where C is a 0, so it is included in the product term complemented.

Variable D: The circle covers a region where D is both a 0 and 1, so it is excluded from the product term.

The product term for this prime implicant is: B·C'

Second Prime Implicant

Variable A: The circle covers a region where A is both a 0 and 1, so it is excluded from the product term.

Variable B: The circle covers a region where B is a 1, so it is included in the product term uncomplemented.

Variable C: The circle covers a region where C is both a 0 and 1, so it is excluded from the product term.

Variable D: The circle covers a region where D is a 0, so it is included in the product term complemented.

The product term for this prime implicant is: B·D'

Step 3: Sum all of the product terms for each prime implicant

There are two product terms, one for each circle. The final minimized SOP expression is:

$$F = B \cdot C' + B \cdot D'$$

This expression could be further factored using the distributive property to F = B·(C' + D') to eliminate one more logic operation; however, since the problem asked for an SOP form, this last step was not necessary.

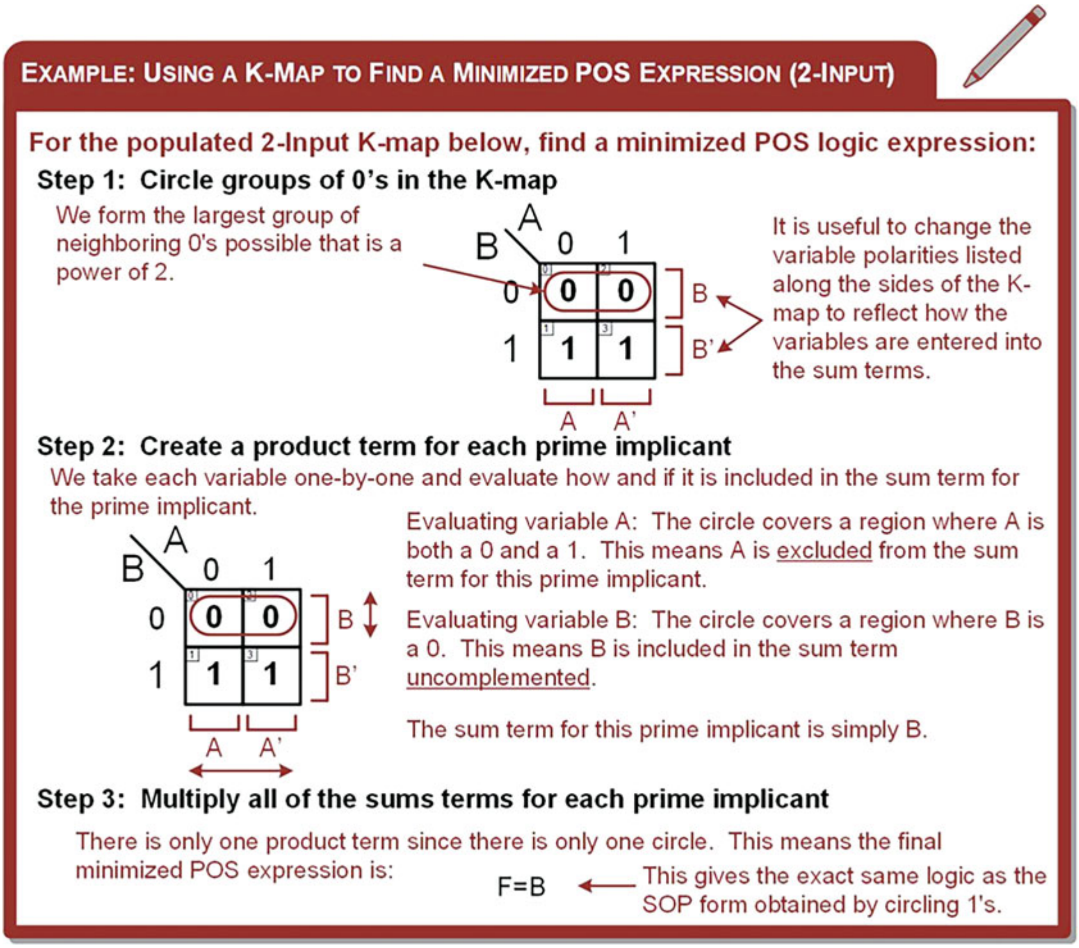
Example 2.25
Using a K-map to find a minimized sum of products expression (4-input)

2.2.3.4 Logic Minimization in POS Form

K-maps can also be used to create minimized product of sums logic expressions. This is the same concept as how a minterm list and maxterm list each produce the same logic function, but in complementary fashions. When creating a product of sums expression from a K-map, groups of 0s are circled. For each circle, a sum term is derived with a negation of variables similar to when forming a maxterm (i.e., in the input variable is a 0, then it is included uncomplemented in the sum term and vice versa). The final step in forming the minimized POS expression is to AND all of the sum terms together. The formal process is as follows:

1. Circle groups of 0s in the K-map following the rules:
   - Each circle should contain the largest number of 0s possible.
   - The circles encompass only neighboring cells (i.e., side-to-side sides and/or top and bottom).
   - The circles must contain a number of 0s that is a power of 2 (i.e., 1, 2, 4, 8, or 16).
   - Enter as many circles as possible without having any circles fully cover another circle.
   - Each circle is called a *prime implicant*.
2. Create a sum term for each prime implicant following the rules:
   - Each variable in the K-map is evaluated one by one.
   - If the circle covers a region where the input variable is a 1, then include it in the sum term *complemented*.
   - If the circle covers a region where the input variable is a 0, then include it in the sum term *uncomplemented*.
   - If the circles cover a region where the input variable is both a 0 and 1, then the variable is *excluded* from the sum term.
3. Multiply all of the sum terms for each prime implicant.

Let's apply this approach to our 2-input K-map example. Example 2.26 shows the process of finding a minimized product of sums logic expression for a 2-input logic circuit using a K-map. Notice that this process yielded the same logic expression as the SOP approach shown in Example 2.23. This illustrates that both the POS and SOP expressions produce the correct logic for the circuit.



EXAMPLE: USING A K-MAP TO FIND A MINIMIZED POS EXPRESSION (2-INPUT)

For the populated 2-Input K-map below, find a minimized POS logic expression:

Step 1: Circle groups of 0's in the K-map

We form the largest group of neighboring 0's possible that is a power of 2.

It is useful to change the variable polarities listed along the sides of the K-map to reflect how the variables are entered into the sum terms.

Step 2: Create a product term for each prime implicant

We take each variable one-by-one and evaluate how and if it is included in the sum term for the prime implicant.

Evaluating variable A: The circle covers a region where A is both a 0 and a 1. This means A is excluded from the sum term for this prime implicant.

Evaluating variable B: The circle covers a region where B is a 0. This means B is included in the sum term uncomplemented.

The sum term for this prime implicant is simply B.

Step 3: Multiply all of the sums terms for each prime implicant

There is only one product term since there is only one circle. This means the final minimized POS expression is:

F=B

This gives the exact same logic as the SOP form obtained by circling 1's.

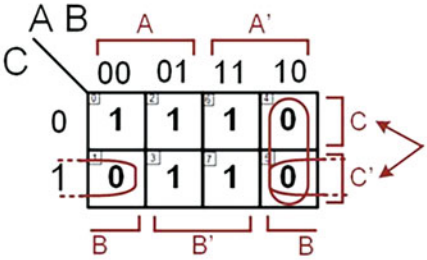
Example 2.26
Using a K-map to find a minimized product of sums expression (2-input)

Let's now apply this process to our 3-input K-map example. Example 2.27 shows the process of finding a minimized product of sums logic expression for a 3-input logic circuit using a K-map. Notice that the logic expression in POS form is not identical to the SOP expression found in Example 2.24; however, using a few steps of algebraic manipulation shows that the POS expression can be put into a form that *is* identical to the prior SOP expression. This illustrates that both the POS and SOP produce equivalent functionality for the circuit.

Example: Using a K-Map to Find a Minimized POS Expression (3-Input)

For the populated 3-Input K-map below, find a minimized POS logic expression:

Step 1: Circle groups of 0's in the K-map



Again, the polarities of the variables along K-map are changed to reflect how the variables are entered into the sum terms.

Step 2: Create a sum term for each prime implicant

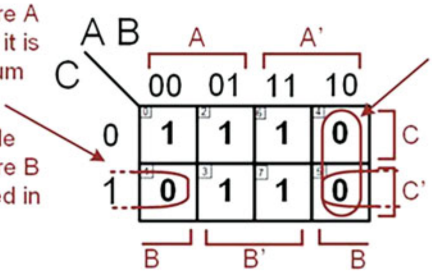
First Prime Implicant

Variable A: The circle covers a region where A is both a 0 and 1, so it is excluded from the sum term.

Variable B: The circle covers a region where B is a 0, so it is included in the sum term uncomplemented.

Variable C: The circle covers a region where C is a 1, so it is included in the sum term complemented.

The sum term for this prime implicant is: B+C'



Second Prime Implicant

Variable A: The circle covers a region where A is a 1, so it is included in the sum term complemented.

Variable B: The circle covers a region where B is a 0, so it is included in the sum term uncomplemented.

Variable C: The circle covers a region where C is both a 0 and 1, so it is excluded from the sum term.

The sum term for this prime implicant is: A'+B

Step 3: Multiply all of the sum terms for each prime implicant

There are two sum terms, one for each circle. The final minimized POS expression is:

F = (B+C')·(A'+B)

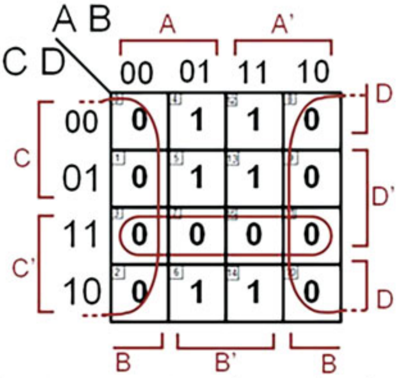
Example 2.27
Using a K-map to find a minimized product of sums expression (3-input)

Let's now apply this process to our 4-input K-map example. Example 2.28 shows the process of finding a minimized product of sums logic expression for a 4-input logic circuit using a K-map.

Example: Using a K-Map to Find a Minimized POS Expression (4-Input)

For the populated 4-Input K-map below, find a minimized POS logic expression:

Step 1: Circle groups of 0's in the K-map



CD \ AB	00	01	11	10
00	0	1	1	0
01	0	1	1	0
11	0	0	0	0
10	0	1	1	0

Again, the polarities of the variables along K-map are changed to reflect how the variables are entered into the sum terms.

Step 2: Create a sum term for each prime implicant

First Prime Implicant

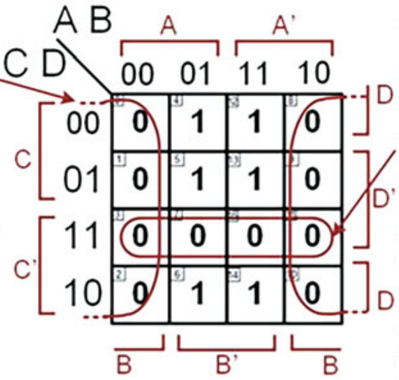
Variable A: The circle covers a region where A is both a 0 and 1, so it is excluded from the sum term.

Variable B: The circle covers a region where B is a 0, so it is included in the sum term uncomplemented.

Variable C: The circle covers a region where C is both a 0 and 1, so it is excluded from the sum term.

Variable D: The circle covers a region where D is both a 0 and 1, so it is excluded from the sum term.

The sum term for this prime implicant is: B



CD \ AB	00	01	11	10
00	0	1	1	0
01	0	1	1	0
11	0	0	0	0
10	0	1	1	0

Second Prime Implicant

Variable A: The circle covers a region where A is both a 0 and 1, so it is excluded from the sum term.

Variable B: The circle covers a region where B is both a 0 and 1, so it is excluded from the sum term.

Variable C: The circle covers a region where C is a 1, so it is included in the sum term complemented.

Variable D: The circle covers a region where D is a 1, so it is included in the sum term complemented.

The sum term for this prime implicant is: C'+D'

Step 3: Multiply all of the sum terms for each prime implicant

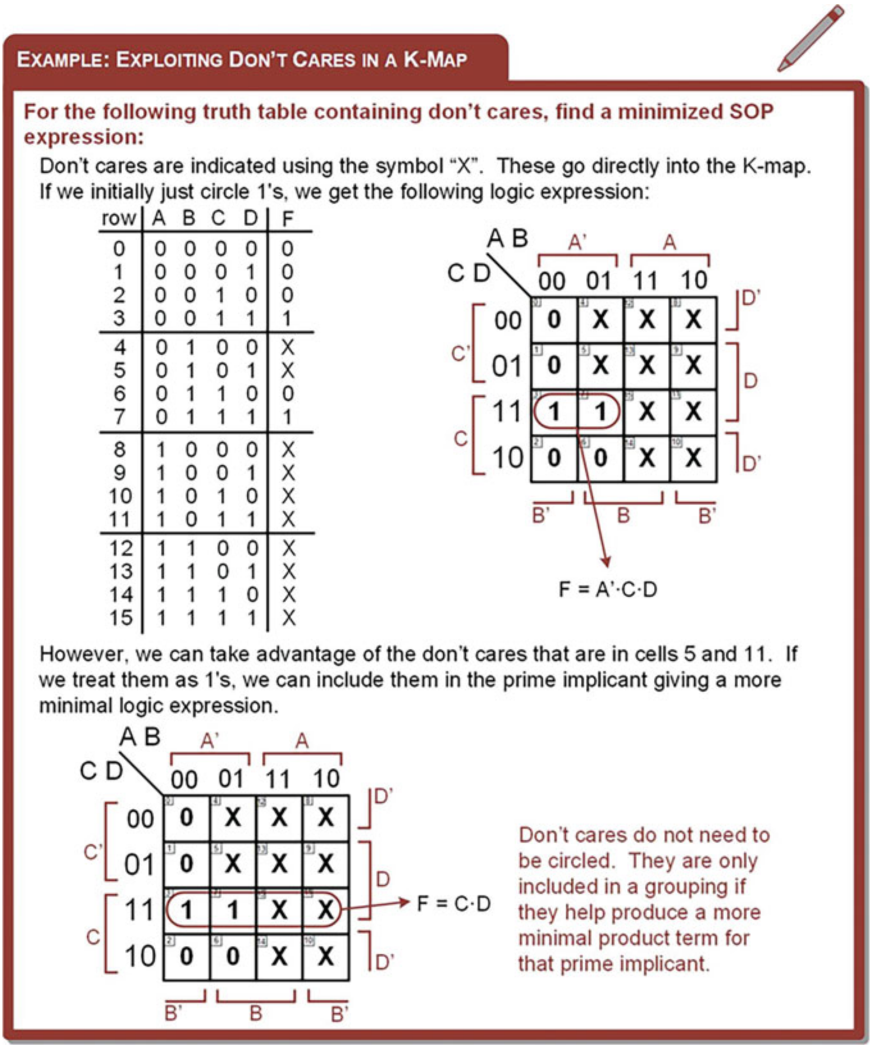
There are two sum terms, one for each circle. The final minimized POS expression is:

$$F = (B)\cdot(C'+D')$$

Example 2.28
Using a K-map to find a minimized product of sums expression (4-input)

2.2.3.5 Don't Cares

There are often times when framing a design problem that there are specific input codes that require exact output values, but there are other codes where the output value doesn't matter. This can occur for a variety of reasons, such as knowing that certain input codes will never occur due to the nature of the problem or that the output of the circuit will only be used under certain input codes. We can take advantage of this situation to produce a more minimal logic circuit. We define an output as a *don't care* when it doesn't matter whether it is a 1 or 0 for the particular input code. The symbol for a don't care is "X." We take advantage of don't cares when performing logic minimization by treating them as whatever output value will produce a minimal logic expression. Example 2.29 shows how to use this process.



EXAMPLE: EXPLOITING DON'T CARES IN A K-MAP

For the following truth table containing don't cares, find a minimized SOP expression:

Don't cares are indicated using the symbol "X". These go directly into the K-map. If we initially just circle 1's, we get the following logic expression:

row	A	B	C	D	F
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	1
4	0	1	0	0	X
5	0	1	0	1	X
6	0	1	1	0	0
7	0	1	1	1	1
8	1	0	0	0	X
9	1	0	0	1	X
10	1	0	1	0	X
11	1	0	1	1	X
12	1	1	0	0	X
13	1	1	0	1	X
14	1	1	1	0	X
15	1	1	1	1	X

CD \ AB	00	01	11	10
00	0	X	X	X
01	0	X	X	X
11	1	1	X	X
10	0	0	X	X

F = A'·C·D

However, we can take advantage of the don't cares that are in cells 5 and 11. If we treat them as 1's, we can include them in the prime implicant giving a more minimal logic expression.

CD \ AB	00	01	11	10
00	0	X	X	X
01	0	X	X	X
11	1	1	X	X
10	0	0	X	X

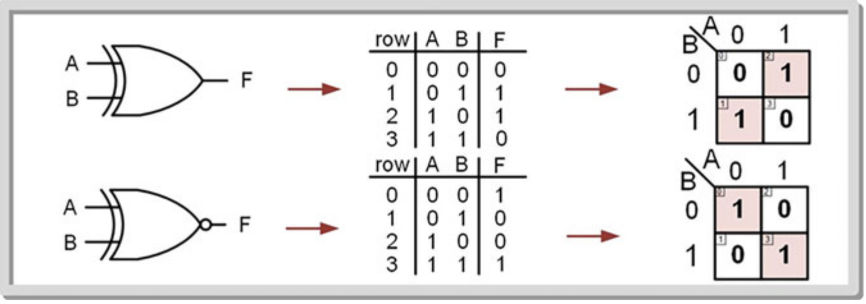
F = C·D

Don't cares do not need to be circled. They are only included in a grouping if they help produce a more minimal product term for that prime implicant.

Example 2.29
Exploiting don't cares in a K-map

2.2.3.6 Identifying XOR Gates in K-maps

While Boolean algebra does not include the exclusive-OR and exclusive-NOR operations, XOR and XNOR gates do indeed exist in modern electronics. They can be a useful tool to provide logic circuitry with less operations, sometimes even compared to a minimal sum or product synthesized using the techniques just described. An XOR can be used to replace a canonical SOP expression such as $F = A'B + AB' = A \oplus B$. An XNOR can be used to replace another canonical SOP expression such as $F = A'B' + AB = (A \oplus B)'$. An XOR/XNOR operation can be identified by putting the values from a truth table into a K-map. The XOR/XNOR operations will result in a characteristic checkerboard pattern in the K-map. Consider the following patterns for XOR and XNOR gates in Figs. 2.14, 2.15, and 2.16. Anytime these patterns are observed, it indicates an XOR/XNOR gate. To find the logic expression, it usually involves writing out the canonical SOP and identifying groups of terms that can be replaced by XOR/XNOR gates.



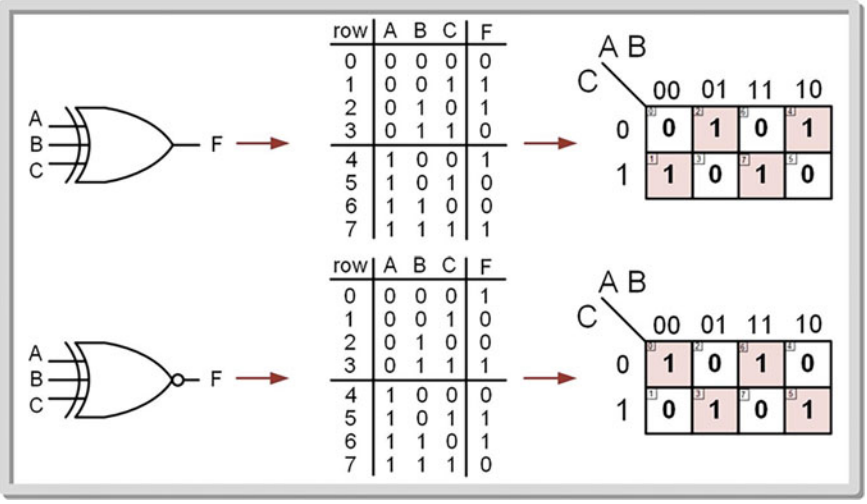
row	A	B	F
0	0	0	0
1	0	1	1
2	1	0	1
3	1	1	0

B \ A	0	1
0	0	1
1	1	0

row	A	B	F
0	0	0	1
1	0	1	0
2	1	0	0
3	1	1	1

B \ A	0	1
0	1	0
1	0	1

Fig. 2.14
XOR and XNOR checkerboard patterns observed in K-maps (2 input)



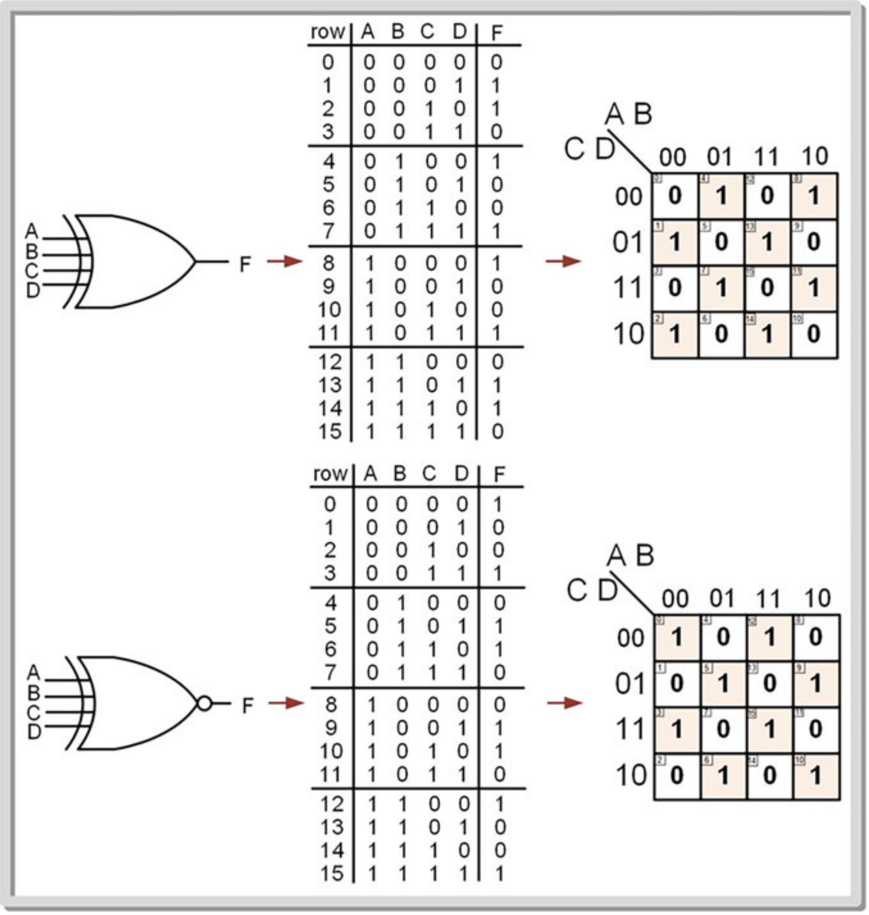
row	A	B	C	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	0
6	1	1	0	0
7	1	1	1	1

C \ AB	00	01	11	10
0	0	1	0	1
1	1	0	1	0

row	A	B	C	F
0	0	0	0	1
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

C \ AB	00	01	11	10
0	1	0	1	0
1	0	1	0	1

Fig. 2.15
XOR and XNOR checkerboard patterns observed in K-maps (3-input)



row	A	B	C	D	F
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	1
3	0	0	1	1	0
4	0	1	0	0	1
5	0	1	0	1	0
6	0	1	1	0	0
7	0	1	1	1	1
8	1	0	0	0	1
9	1	0	0	1	0
10	1	0	1	0	0
11	1	0	1	1	1
12	1	1	0	0	0
13	1	1	0	1	1
14	1	1	1	0	1
15	1	1	1	1	0

CD \ AB	00	01	11	10
00	0	1	0	1
01	1	0	1	0
11	0	1	0	1
10	1	0	1	0

row	A	B	C	D	F
0	0	0	0	0	1
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	1
4	0	1	0	0	0
5	0	1	0	1	1
6	0	1	1	0	1
7	0	1	1	1	0
8	1	0	0	0	0
9	1	0	0	1	1
10	1	0	1	0	1
11	1	0	1	1	0
12	1	1	0	0	1
13	1	1	0	1	0
14	1	1	1	0	0
15	1	1	1	1	1

CD \ AB	00	01	11	10
00	1	0	1	0
01	0	1	0	1
11	1	0	1	0
10	0	1	0	1

Fig. 2.16
XOR and XNOR checkerboard pattern observed in K-maps (4-input)

Concept Check

CC2.2.3(a) All logic functions can be implemented equivalently using either a *canonical* sum of products (SOP) or *canonical* product of sums (POS) topology. Which of these statements is true with respect to selecting a topology that requires the least number of gates?

A) Since a minterm list and a maxterm list can both be written to describe the same logic functionality, the number of gates in an SOP and POS will always be the same.

B) If a minterm list has over half of its row numbers listed, an SOP topology will require fewer gates than a POS.

C) A POS topology always requires more gates because it needs additional logic to convert the inputs from positive to negative logic.

D) If a minterm list has over half of its row numbers listed, a POS topology will require fewer gates than SOP.

CC2.2.3(b) Logic minimization is accomplished by removing variables from the original canonical logic expression that don't impact the result. How does a Karnaugh map graphically show what variables can be removed?

A) K-maps contain the same information as a truth table but the data is formatted as a grid. This allows variables to be removed by inspection.

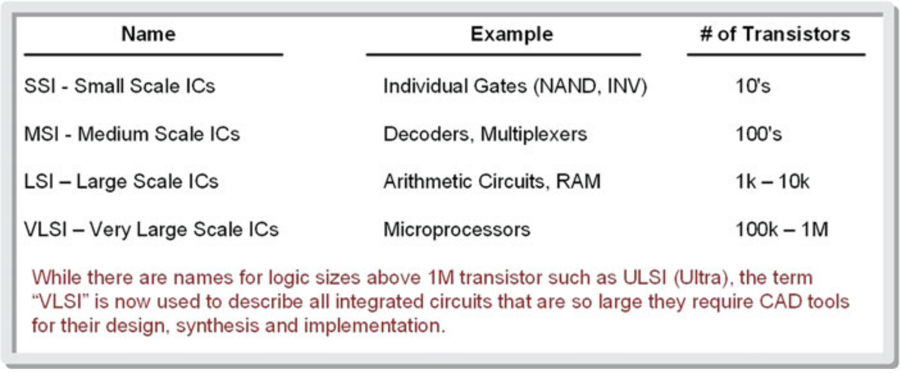
B) K-maps rearrange a truth table so that adjacent cells have one and only one input variable changing at a time. If adjacent cells have the same output value when an input variable is both a 0 and a 1, that variable has no impact on the interim result and can be eliminated.

C) K-maps list both the rows with outputs of 1s and 0s simultaneously. This allows minimization to occur for a SOP and POS topology that each have the same, but minimal, number of gates.

D) K-maps display the truth table information in a grid format, which is a more compact way of presenting the behavior of a circuit.

2.2.4 MSI Logic

This section introduces a group of combinational logic building blocks that are commonly used in digital design. As we move into systems that are larger than individual gates, there are naming conventions that are used to describe the size of the logic. Table 2.4 gives these naming conventions. In this chapter we will look at *medium-scale integrated circuit* (MSI) logic. Each of these building blocks can be implemented using the combinational logic design steps covered in Chaps. 4 and 5. The goal of this chapter is to provide an understanding of the basic principles of MSI logic.



Name	Example	# of Transistors
SSI - Small Scale ICs	Individual Gates (NAND, INV)	10's
MSI - Medium Scale ICs	Decoders, Multiplexers	100's
LSI – Large Scale ICs	Arithmetic Circuits, RAM	1k – 10k
VLSI – Very Large Scale ICs	Microprocessors	100k – 1M

While there are names for logic sizes above 1M transistor such as ULSI (Ultra), the term "VLSI" is now used to describe all integrated circuits that are so large they require CAD tools for their design, synthesis and implementation.

Table 2.4
Naming convention for the size of digital systems

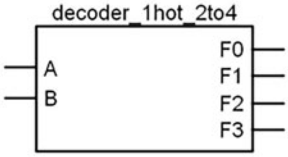
2.2.4.1 Decoders

A decoder is a circuit that takes in a binary *code* and has outputs that are asserted for specific values of that code. The code can be of any type or size (e.g., unsigned, two's complement). Each output will assert for only specific input codes. Since combinational logic circuits only produce a single output, this means that within a decoder, there will be a separate combinational logic circuit for each output.

A one-hot decoder is a circuit that has n inputs and 2^n outputs. Each output will assert for one and only one input code. Since there are 2^n outputs, there will always be one and only one output asserted at any given time. Example 2.30 shows the process of designing a 2-to-4 one-hot decoder by hand (i.e., using the classical digital design approach).

Example: Design of a 2-to-4 One-Hot Decoder

The block diagram and truth table for this system are as follows:



A	B	F3	F2	F1	F0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Each output asserts for a specific input code. This is where the term "one-hot" comes from. Each output is only "hot" for one input code.

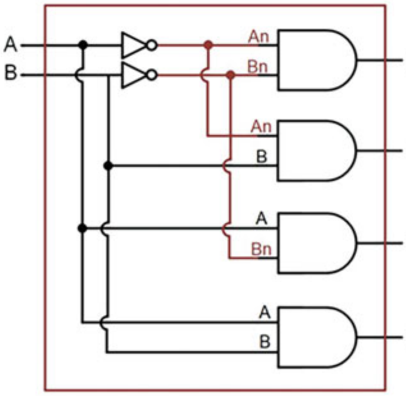
When designing this circuit, each output needs to have its own separate combinational logic circuit. This is the same as if there were four separate truth tables. This design could be implemented using 4x, 2-input K-maps to form the logic expressions for these outputs; however, by inspection a minterm list for each output will be the most minimal circuit.

$$F0 = \sum_{A,B}(0) = A'{\cdot}B' \qquad F2 = \sum_{A,B}(2) = A{\cdot}B'$$

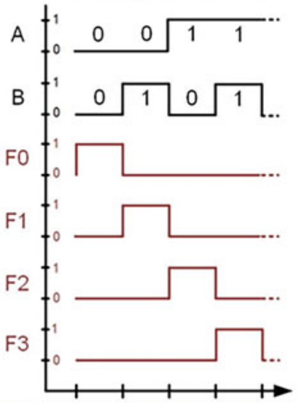
$$F1 = \sum_{A,B}(1) = A'{\cdot}B \qquad F3 = \sum_{A,B}(3) = A{\cdot}B$$

When implementing the final decoder, the input inversions for A and B can be shared across all of the AND gates.

Logic Diagram



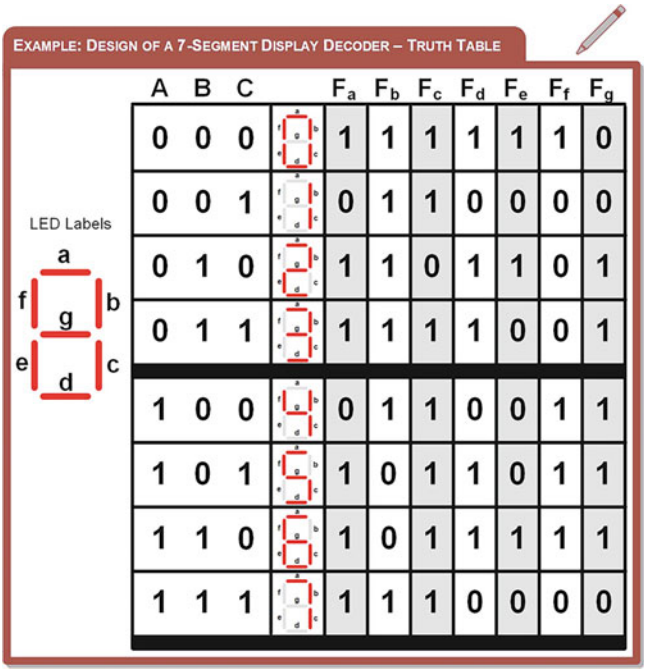
Timing Waveform



Example 2.30
Design of a 2-to-4 one-hot decoder

Another common decoder found in digital logic is a 7-segment display decoder. This decoder is a circuit used to drive character displays that are commonly found in applications such as digital clocks and household appliances. A character display is made up of seven individual LEDs, typically labeled a–g. The input to the decoder is the binary equivalent of the decimal or hex character that is to be displayed. The output of the decoder is the arrangement of LEDs that will form the character. Decoders with 2-inputs can drive characters "0" to "3." Decoders with 3-inputs can drive characters "0" to "7." Decoders with 4-inputs can drive characters "0" to "F" with the case of the hex characters being "A, b, c or C, d, E, and F."

Let's look at an example of how to design a 3-input, 7-segment decoder by hand. The first step in the process is to create the truth table for the outputs that will drive the LEDs in the display. We'll call these outputs F_a, F_b, . . ., F_g. Example 2.31 shows how to construct the truth table for the 7-segment display decoder. In this table, a logic 1 corresponds to the LED being ON.



EXAMPLE: DESIGN OF A 7-SEGMENT DISPLAY DECODER – TRUTH TABLE

LED Labels
a
b
c
d
e
f
g

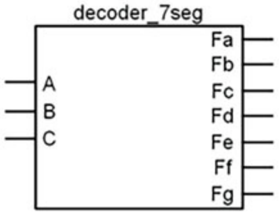
A	B	C		F_a	F_b	F_c	F_d	F_e	F_f	F_g
0	0	0		1	1	1	1	1	1	0
0	0	1		0	1	1	0	0	0	0
0	1	0		1	1	0	1	1	0	1
0	1	1		1	1	1	1	0	0	1
1	0	0		0	1	1	0	0	1	1
1	0	1		1	0	1	1	0	1	1
1	1	0		1	0	1	1	1	1	1
1	1	1		1	1	1	0	0	0	0

Example 2.31
Design of a 7-segment display decoder: truth table

If we wish to design this decoder by hand, we need to create seven separate combinational logic circuits. Each of the outputs (F_a – F_g) can be put into a 3-input K-map to find the minimized logic expression. Example 2.32 shows the design of the decoder from the truth table in Example 2.31.

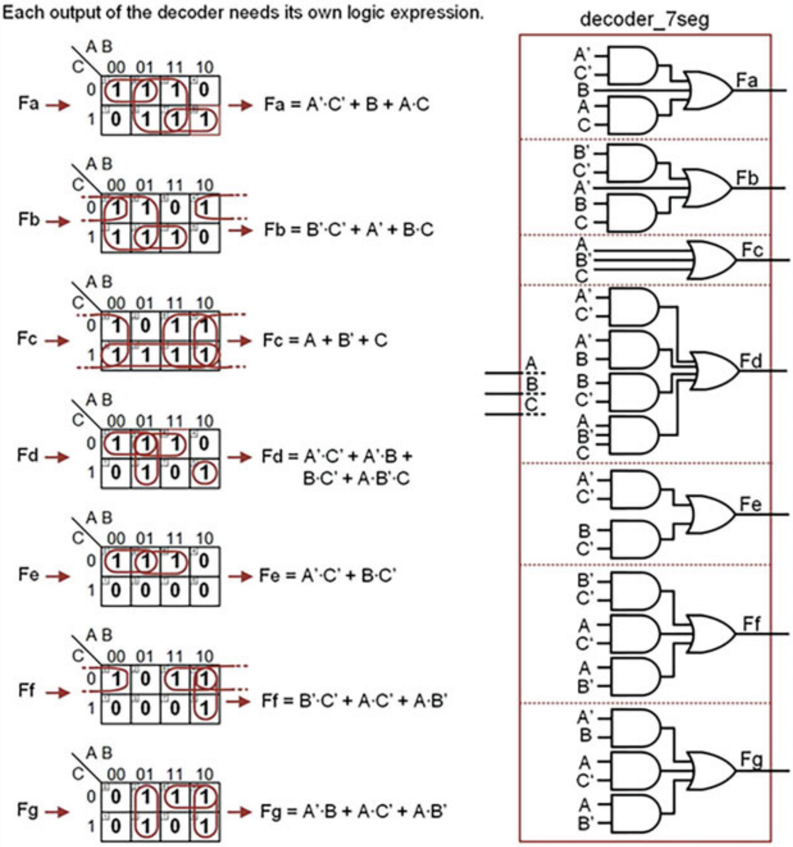
Example: Design of a 7-Segment Display Decoder – Synthesis

The block diagram and truth table for this system are as follows:



A	B	C	Fa	Fb	Fc	Fd	Fe	Ff	Fg
0	0	0	1	1	1	1	1	1	0
0	0	1	0	1	1	0	0	0	0
0	1	0	1	1	0	1	1	0	1
0	1	1	1	1	1	1	0	0	1
1	0	0	0	1	1	0	0	1	1
1	0	1	1	0	1	1	0	1	1
1	1	0	1	0	1	1	1	1	1
1	1	1	1	1	1	0	0	0	0

Each output of the decoder needs its own logic expression.



$F_a = A' \cdot C' + B + A \cdot C$

$F_b = B' \cdot C' + A' + B \cdot C$

$F_c = A + B' + C$

$F_d = A' \cdot C' + A' \cdot B + B \cdot C' + A \cdot B' \cdot C$

$F_e = A' \cdot C' + B \cdot C'$

$F_f = B' \cdot C' + A \cdot C' + A \cdot B'$

$F_g = A' \cdot B + A \cdot C' + A \cdot B'$

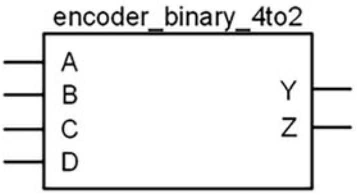
Example 2.32
Design of a 7-segment display decoder: synthesis

2.2.4.2 Encoders

An encoder works in the opposite manner as a decoder. An assertion on a specific input port corresponds to a unique code on the output port. A one-hot *binary* encoder has n outputs and 2^n inputs. The output will be an *n*-bit, binary code which corresponds to an assertion on one and only one of the inputs. Example 2.33 shows the process of designing a 4-to-2 binary encoder.

Example: Design of a 4-to-2 Binary Encoder

The block diagram and truth table for this system are as follows:

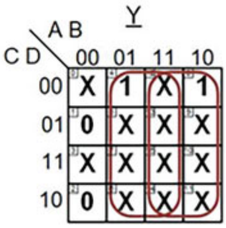


A	B	C	D	Y	Z
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

←

Binary Code Mapping
D→"00"
C→"01"
B→"10"
A→"11"

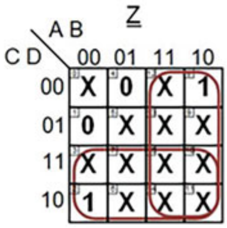
When designing this circuit, each output needs to have its own separate combinational logic circuit. When constructing the K-maps for Y and Z, each will have 4-inputs (A, B, C, D). The output values for many of the input codes are not specified in the above truth table. As such, we can use Don't Cares (X) to simplify the logic.



Y

CD \ AB	00	01	11	10
00	X	1	X	1
01	0	X	X	X
11	X	X	X	X
10	0	X	X	X

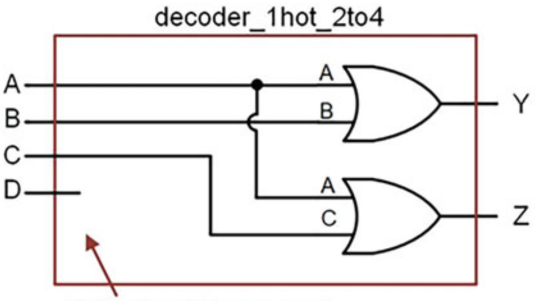
→ Y = A + B



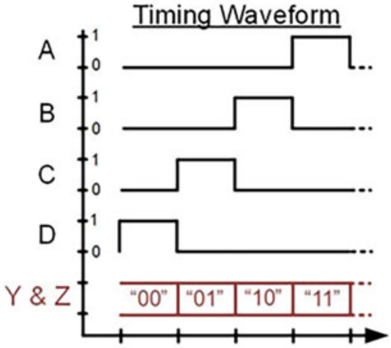
Z

CD \ AB	00	01	11	10
00	X	0	X	1
01	0	X	X	X
11	X	X	X	X
10	1	X	X	X

→ Z = A + C



Notice that D is not used.



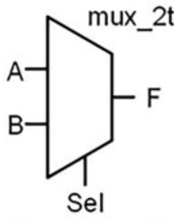
Example 2.33
Design of a 4-to-2 binary encoder

2.2.4.3 Multiplexers

A multiplexer is a circuit that passes one of its multiple inputs to a single output based on a select input. This can be thought of as a digital switch. The multiplexer has n select lines, 2^n inputs, and one output. Example 2.34 shows the process of designing a 2-to-1 multiplexer by hand (i.e., using the classical digital design approach).

EXAMPLE: DESIGN OF A 2-TO-1 MULTIPLEXER

The symbol and truth table for the 2-to-1 multiplexer are as follows:



Sel	F
0	A
1	B

In order to design the multiplexer, it is helpful to list all possible values for A, B and Sel in a truth table form.

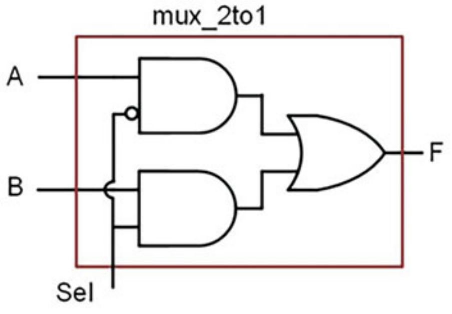
Sel	A	B	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

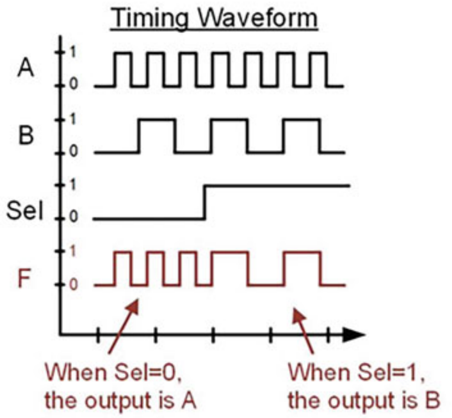
When Sel=0, the output is A

When Sel=1, the output is B

B \ Sel A	00	01	11	10
0	0	1	0	0
1	0	1	1	1

F = Sel'·A + Sel·B





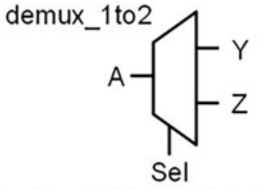
Example 2.34
Design of a 2-to-1 multiplexer

2.2.4.4 Demultiplexers

A demultiplexer works in a complementary fashion to a multiplexer. A demultiplexer has one input that is routed to one of its multiple outputs. The output that is active is dictated by a select input. A demux has *n* select lines that chooses to route the input to one of its 2^n outputs. When an output is not selected, it outputs a logic 0. Example 2.35 shows the process of designing a 1-to-2 demultiplexer by hand (i.e., using the classical digital design approach).

Example: Design of a 1-to-2 Demultiplexer

The symbol and truth table for the 1-to-2 multiplexer are as follows:



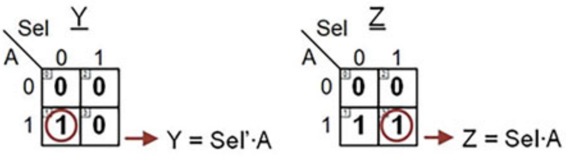
Sel	Y	Z
0	A	0
1	0	A

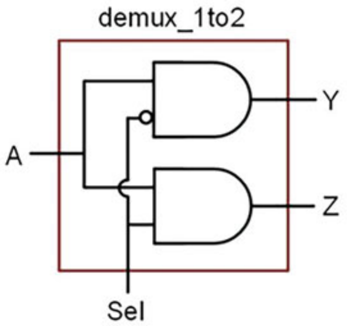
In order to design the demultiplexer, it is helpful to list all possible values for A and Sel and the corresponding outputs on Y and Z. A separate circuit is needed for both Y and Z.

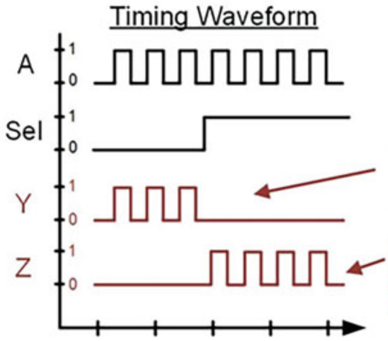
When Sel=0, the Y = A →

When Sel=1, the Y = B →

Sel	A	Y	Z
0	0	0	0
0	1	1	0
1	0	0	0
1	1	0	1







When Sel=0, Y=A. Y=0 otherwise.

When Sel=1, Z=A. Z=0 otherwise.

Example 2.35
Design of a 1-to-2 demultiplexer

2.2.4.5 Adders

Binary addition is performed in a similar manner to performing decimal addition by hand. The addition begins in the least significant position of the number ($p = 0$). The addition produces the sum for this position. In the event that this positional sum cannot be represented by a single symbol, then the higher-order symbol is *carried* to the subsequent position ($p = 1$). The addition in the next higher position must include the number that was carried in from the lower positional sum. This process continues until all of the symbols in the number have been operated on. The final positional sum can also produce a carry, which needs to be accounted for in a separate system.

Designing a binary adder involves creating a combinational logic circuit to perform the positional additions. Since a combinational logic circuit can only produce a scalar output, circuitry is needed to produce the sum and the carry at each position. The binary adder size is predetermined and fixed prior to implementing the logic (i.e., an *n*-bit adder). Both inputs to the adder must adhere to the fixed size, regardless of their value. Smaller numbers simply contain leading 0s in their higher-order positions. For an *n*-bit adder, the largest sum that can be produced will require $n + 1$ bits. To illustrate this, consider a 4-bit adder. The largest numbers that the adder will operate on are $1111_2 + 1111_2$. (or $15_{10} + 15_{10}$). The result of this addition is 11110_2 (or 30_{10}). Notice that the largest sum produced fits within 5 bits, or $n + 1$. When constructing an adder circuit, the sum is always recorded using *n*-bits with a separate carryout bit. In our 4-bit example, the sum would be expressed as "1110" with a carryout. The carryout bit can be used in multiple word additions, used as part of the number when being decoded for a display, or simply discarded as in the case when using two's complement numbers.

When creating an adder, it is desirable to design incremental sub-systems that can be reused. This reduces design effort and minimizes troubleshooting complexity. The most basic component in the adder is called a *half adder*. This circuit computes the sum and carryout on two input arguments. The reason it is called a half adder instead of a full adder is because it does not accommodate a *carry in* during the computation; thus, it does not provide all of the necessary functionality required for the positional adder. Example 2.36 shows the design of a half adder. Notice that two combinational logic circuits are required in order to produce the sum (the XOR gate) and the carryout (the AND gate). These two gates are in parallel to each other; thus, the delay through the half adder is due to only one level of logic.

EXAMPLE: DESIGN OF A HALF ADDER

Recall in binary addition, the output consists of a sum and a carry bit.

$$\begin{array}{r} 0 \\ +\ 0 \\ \hline \underline{0} \end{array} \leftarrow \text{Sum} \qquad \begin{array}{r} 0 \\ +\ 1 \\ \hline 1 \end{array} \qquad \begin{array}{r} 1 \\ +\ 0 \\ \hline 1 \end{array} \qquad \text{Carry} \rightarrow \begin{array}{r} 1 \\ +\ 1 \\ \hline \underline{1}\,0 \end{array}$$

We can build a simple circuit callled a "Half Adder" to compute these outputs.

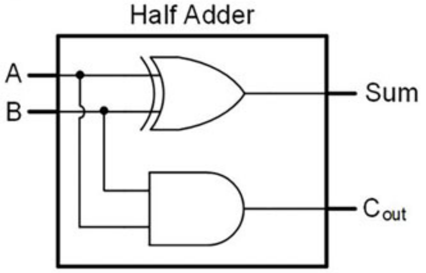
A	B	C_{out}	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

→ By inpsection

$\text{Sum} = A \oplus B$

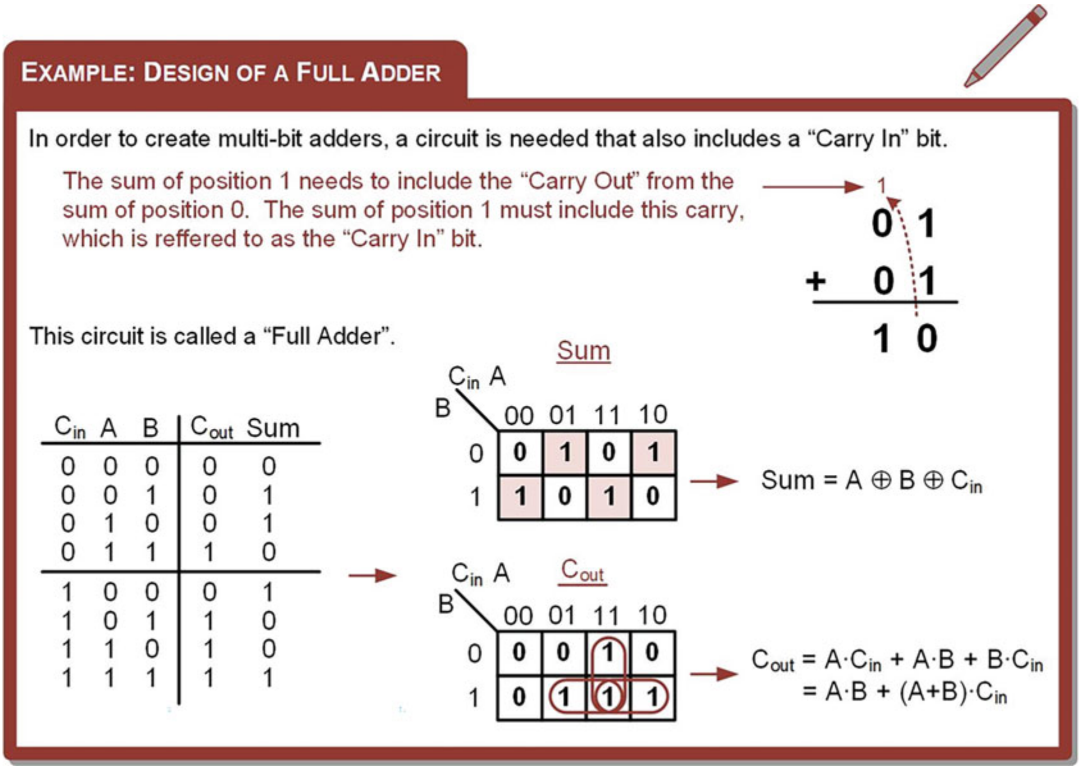
$C_{out} = A \cdot B$

→



Example 2.36
Design of a half adder

A full adder is a circuit that still produces a sum and carryout, but considers three inputs in the computations (A, B, and C_{in}). Example 2.37 shows the design of a full adder.



EXAMPLE: DESIGN OF A FULL ADDER

In order to create multi-bit adders, a circuit is needed that also includes a "Carry In" bit.

The sum of position 1 needs to include the "Carry Out" from the sum of position 0. The sum of position 1 must include this carry, which is reffered to as the "Carry In" bit.

$$\begin{array}{rcc} & 1 & \\ & 0 & 1 \\ + & 0 & 1 \\ \hline & 1 & 0 \end{array}$$

This circuit is called a "Full Adder".

C_{in}	A	B	C_{out}	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Sum

B \ C_{in} A	00	01	11	10
0	0	1	0	1
1	1	0	1	0

$\text{Sum} = A \oplus B \oplus C_{in}$

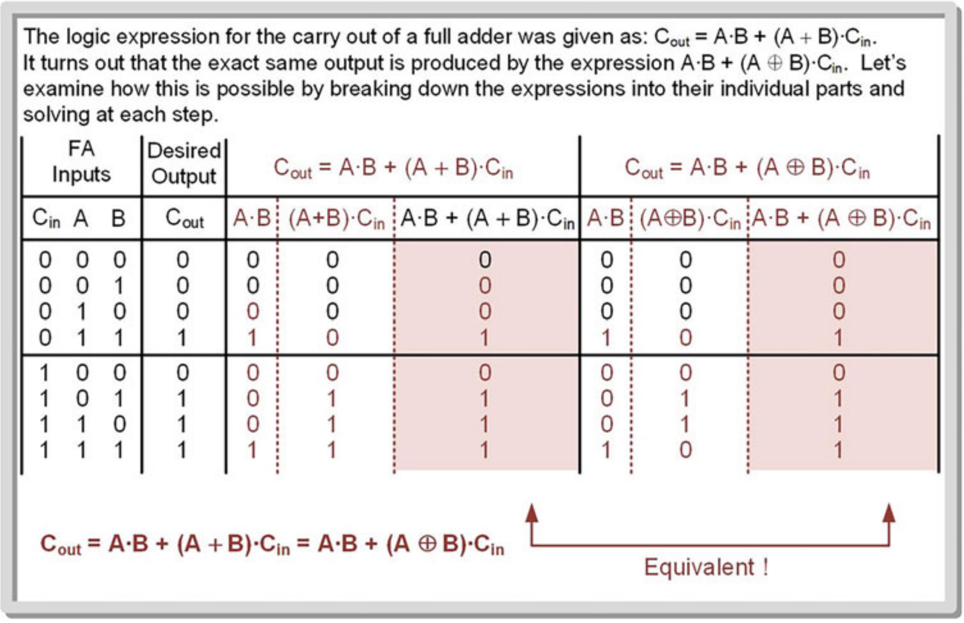
C_{out}

B \ C_{in} A	00	01	11	10
0	0	0	1	0
1	0	1	1	1

$$C_{out} = A \cdot C_{in} + A \cdot B + B \cdot C_{in}$$
$$= A \cdot B + (A+B) \cdot C_{in}$$

Example 2.37
Design of a full adder

As mentioned before, it is desirable to reuse design components as we construct more complex systems. One such design reuse approach is to create a full adder using two half adders. This is straightforward for the sum output since the logic is simply two cascaded XOR gates ($\text{Sum} = A \oplus B \oplus C_{in}$). The carryout is not as straightforward. Notice that the expression for Cout derived in Example 2.37 contains the term $(A + B)$. If this term could be manipulated to use an XOR gate instead, it would allow the full adder to take advantage of existing circuitry in the system. Figure 2.17 shows a derivation of an equivalency that allows $(A + B)$ to be replaced with $(A \oplus B)$ in the Cout logic expression.



The logic expression for the carry out of a full adder was given as: $C_{out} = A{\cdot}B + (A + B){\cdot}C_{in}$. It turns out that the exact same output is produced by the expression $A{\cdot}B + (A \oplus B){\cdot}C_{in}$. Let's examine how this is possible by breaking down the expressions into their individual parts and solving at each step.

FA Inputs			Desired Output	$C_{out} = A{\cdot}B + (A + B){\cdot}C_{in}$			$C_{out} = A{\cdot}B + (A \oplus B){\cdot}C_{in}$		
C_{in}	A	B	C_{out}	$A{\cdot}B$	$(A+B){\cdot}C_{in}$	$A{\cdot}B + (A + B){\cdot}C_{in}$	$A{\cdot}B$	$(A \oplus B){\cdot}C_{in}$	$A{\cdot}B + (A \oplus B){\cdot}C_{in}$
0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0
0	1	1	1	1	0	1	1	0	1
1	0	0	0	0	0	0	0	0	0
1	0	1	1	0	1	1	0	1	1
1	1	0	1	0	1	1	0	1	1
1	1	1	1	1	1	1	1	0	1

$$C_{out} = A{\cdot}B + (A + B){\cdot}C_{in} = A{\cdot}B + (A \oplus B){\cdot}C_{in}$$

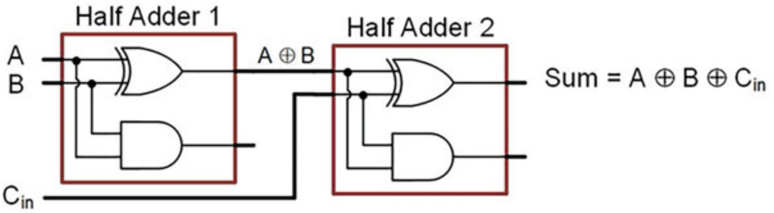
Equivalent !

Fig. 2.17
A useful equivalency that can be exploited in arithmetic circuits

The ability to implement the carryout logic using the expression $C_{out} = A{\cdot}B + (A \oplus B){\cdot}C_{in}$ allows us to implement a full adder with two half adders and the addition of a single OR gate. Example 2.38 shows this approach. In this new configuration, the sum is produced in two levels of logic while the carryout is produced in three levels of logic.

Example: Design of a Full Adder using two Half Adders

It is often desirable to create a full adder out of two half adders in order to re-use existing design components. The "Sum" of the full adder can be created by using two cascaded XOR gates provided by the half adders.



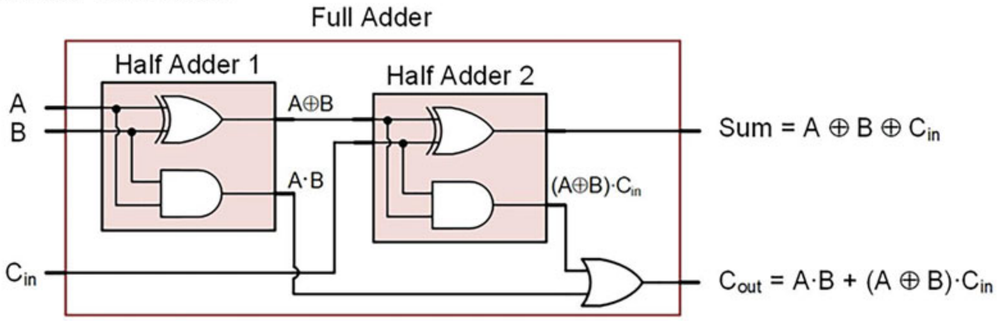
The expression for the "Carry Out" of the full adder is:

$$C_{out} = A \cdot B + (A + B) \cdot C_{in}$$

or

$$C_{out} = A \cdot B + (A \oplus B) \cdot C_{in}$$

Notice that the carry out of Half Adder 1 produces the A·B term in this expression. Also notice that the carry out of Half Adder 2 produces the $(A \oplus B) \cdot C_{in}$ term. The only remaining logic needed to create the carry out of the full adder is an OR gate. The final logic diagram for the full adder is as follows:

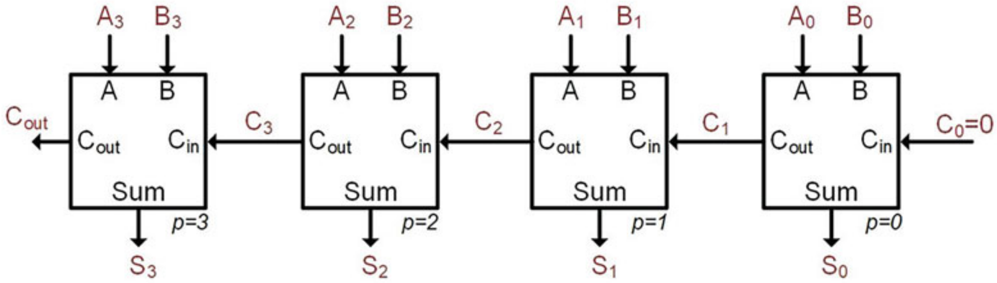


Example 2.38
Design of a full adder using two half adders

The full adder can now be used in the creation of multi-bit adders. The simplest architecture exploiting the full adder is called a *ripple carry adder* (RCA). In this approach, full adders are used to create the sum and carry out of each bit position. The carryout of each full adder is used as the carry in for the next higher position. Since each subsequent full adder needs to wait for the carry to be produced by the preceding stage, the carry is said to *ripple* through the circuit, thus giving this approach its name. Example 2.39 shows how to design a 4-bit ripple carry adder using a chain of full adders. Notice that the carry in for the full adder in position 0 is tied to a logic 0. The 0 input has no impact on the result of the sum but enables a full adder to be used in the 0th position.

Example: Design of a 4-Bit Ripple Carry Adder (RCA)

Full adders can be cascaded together to form a multi-bit adder. The symbols are typically drawn in the following fashion to mirror a positional number system.

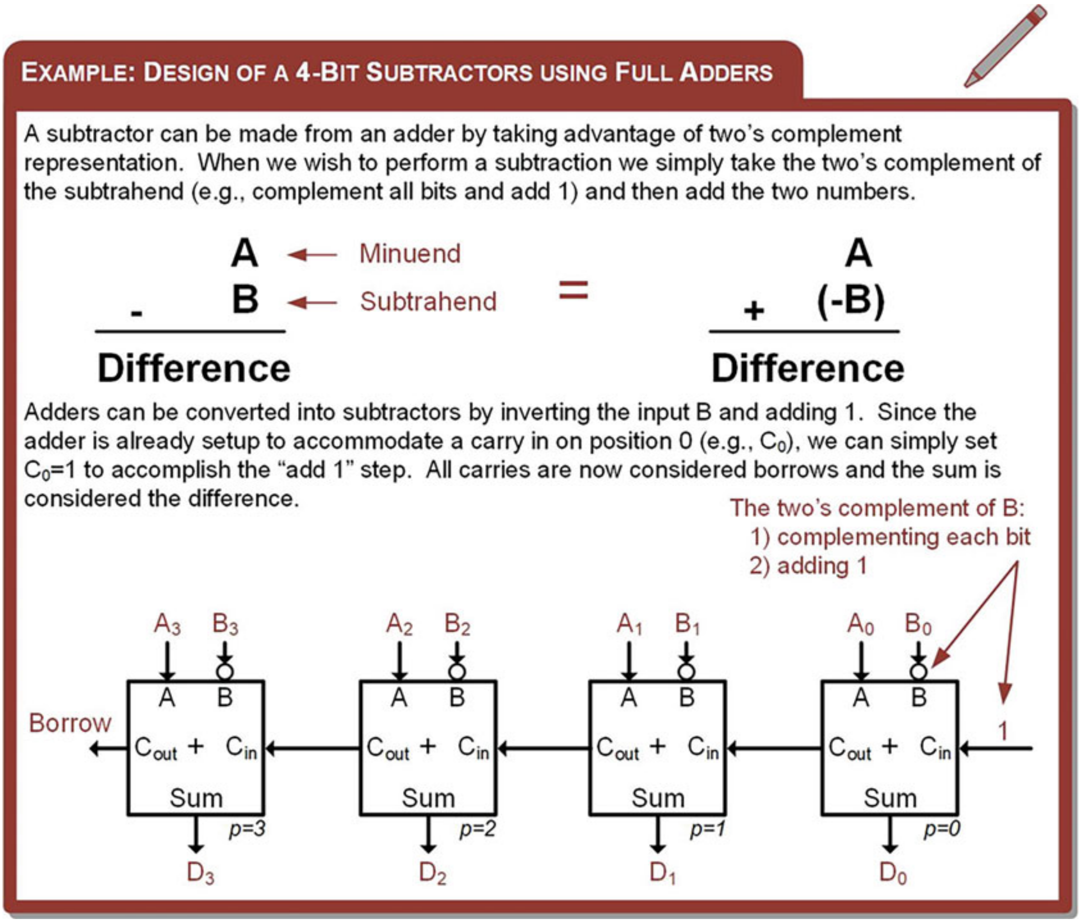


The sum of position 1 cannot complete until it receives the carry in (C_1) from the sum in position 0. The position 2 sum cannot complete until it receives the carry in (C_2) from the sum in position 1, etc. In this way, the carry "ripples" through the circuit from right to left. This configuration is known as a Ripple Carry Adder (RCA).

Example 2.39
Design of a 4-bit ripple carry adder (RCA)

2.2.4.6 Subtractors

Binary subtraction can be accomplished by building a dedicated circuit using a similar design approach as just described for adders. A more effective approach is to take advantage of two's complement representation in order to reuse existing adder circuitry. Recall that taking the two's complement of a number will produce an equivalent magnitude number, but with the opposite sign (i.e., positive to negative or negative to positive). This means that all that is required to create a subtractor from an adder is to first take the two's complement of the subtrahend input. Since the steps to take the two's complement of a number involve complementing each of the bits in the number and then adding 1, the logic required is relatively simple. Example 2.40 shows a 4-bit subtractor using full adders. The subtrahend B is inverted prior to entering the full adders. Also, the carry in bit C_0 is set to 1. This handles the "adding 1" step of the two's complement. All of the carries in the circuit are now treated as *borrows* and the sum is now treated as the *difference*.



EXAMPLE: DESIGN OF A 4-BIT SUBTRACTORS USING FULL ADDERS

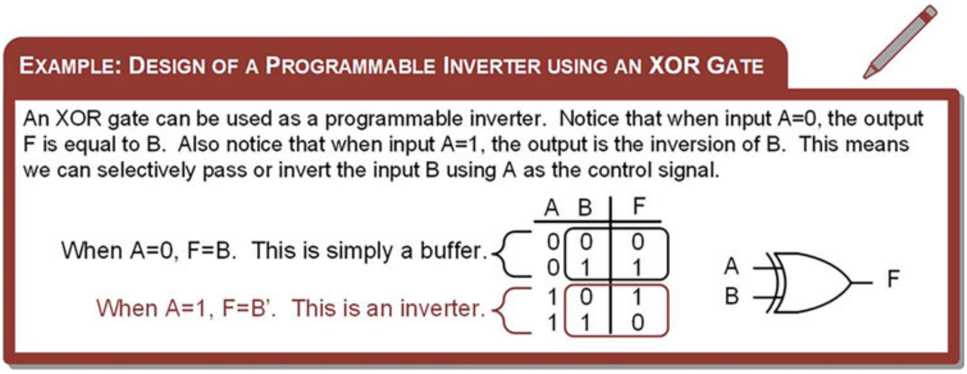
A subtractor can be made from an adder by taking advantage of two's complement representation. When we wish to perform a subtraction we simply take the two's complement of the subtrahend (e.g., complement all bits and add 1) and then add the two numbers.

Adders can be converted into subtractors by inverting the input B and adding 1. Since the adder is already setup to accommodate a carry in on position 0 (e.g., C_0), we can simply set C_0=1 to accomplish the "add 1" step. All carries are now considered borrows and the sum is considered the difference.

The two's complement of B:
1) complementing each bit
2) adding 1

Example 2.40
Design of a 4-bit subtractor using full adders

A programmable adder/subtractor can be created with the use of a programmable inverter and a control signal. The control signal will selectively invert B and also change the C_0 bit between a 0 (for adding) and a 1 (for subtracting). Example 2.41 shows how an XOR gate can be used to create a programmable inverter for use in a programmable adder/subtractor circuit.



EXAMPLE: DESIGN OF A PROGRAMMABLE INVERTER USING AN XOR GATE

An XOR gate can be used as a programmable inverter. Notice that when input A=0, the output F is equal to B. Also notice that when input A=1, the output is the inversion of B. This means we can selectively pass or invert the input B using A as the control signal.

When A=0, F=B. This is simply a buffer.

When A=1, F=B'. This is an inverter.

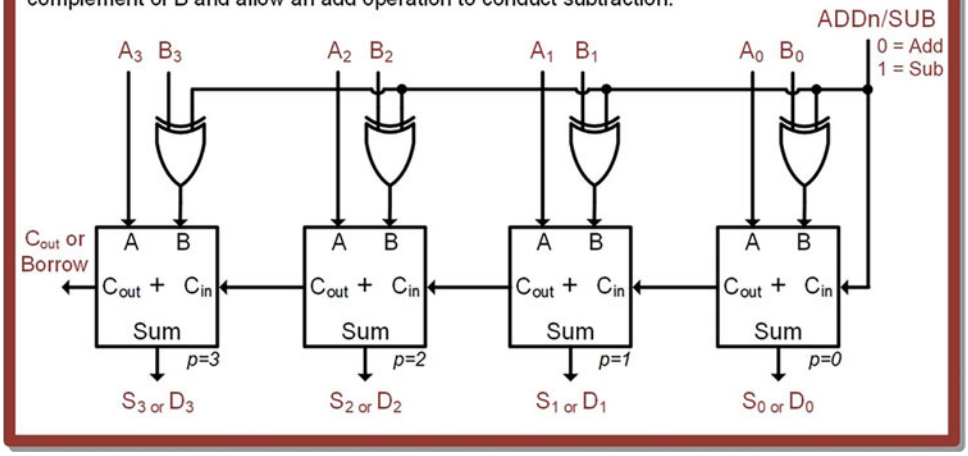
A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

Example 2.41
Design of a programmable inverter using an XOR gate

We can now define a control signal called (ADDn/SUB) that will control whether the circuit performs addition or subtraction. Example 2.42 shows the architecture of a 4-bit programmable adder/subtractor. It should be noted that this programmability adds another level of logic to the circuit, thus increasing its delay.

Example: Design of a 4-Bit Programmable Adder/Subtractor

The control signal "ADDn/SUB" is used to select whether the circuit performs addition (ADDn/SUB=0) or subtraction (ADDn/SUB=1). When in subtraction mode, the XOR gates invert the subtrahend B and add 1 to the first adder stage. These steps take the two's complement of B and allow an add operation to conduct subtraction.



Example 2.42
Design of a 4-bit programmable adder/subtractor

Concept Check

CC2.2.4(a) In a decoder, a logic expression is created for each output. Once all of the output logic expressions are found, how can the decoder logic be further minimized?

A) By using K-maps to find the output logic expressions.

B) By buffering the inputs so that they can drive a large number of other gates.

C) By identifying any logic terms that are used in multiple locations (inversions, product terms, and sum terms) and sharing the interim results among multiple circuits in the decoder.

D) By using a canonical product of sums (POS) architecture.

CC2.2.4(b) How many select lines are needed in a 1-to-64 demultiplexer?

A) 1

B) 4

C) 6

D) 64

CC2.2.4(c) Does a binary adder behave differently when it's operating on unsigned vs. two's complement numbers? Why or why not?

A) Yes. The adder needs to keep track of the sign bit; thus, extra circuitry is needed.

B) No. The binary addition is identical. It is up to the designer to handle how the two's complement codes are interpreted and whether two's complement overflow occurred using a separate system.

2.3 Sequential Logic

In this section, we look at sequential logic design. Sequential logic design differs from combinational logic design in that the outputs of the circuit depend not only on the current values of the inputs but also on the *past* values of the inputs. This is different from the combinational logic design where the output of the circuitry depends only on the current values of the inputs. The ability of a sequential logic circuit to base its outputs on both the current and past inputs allows more sophisticated and intelligent systems to be created. We begin by looking at sequential logic storage devices, which are used to hold the past values of a system. This is followed by an investigation of timing considerations of sequential logic circuits. Finally, we look at one of the most important logic circuits in digital systems, the finite state machine. The goal of this section is to provide an understanding of the basic operation of sequential logic circuits.

2.3.1 Sequential Logic Storage Devices

2.3.1.1 The Cross-Coupled Inverter Pair

The first thing that is needed in sequential logic is a storage device. The fundamental storage device in sequential logic is based on a positive feedback configuration. Consider the circuit in Fig. 2.18. This circuit configuration is called the *cross-coupled inverter pair*. In this circuit if the input of U1 starts with a value of 1, it will produce an output of $Q = 0$. This output is fed back to the input of U2, thus producing an output of $Q_n = 1$. Qn is fed back to the original input of U1, thus reinforcing the initial condition. This circuit will *hold*, or *store,* a logic 0 without being driven by any other inputs. This circuit operates in a complementary manner when the initial value of U1 is a 0. With this input condition, the circuit will store a logic 1 without being driven by any other inputs.

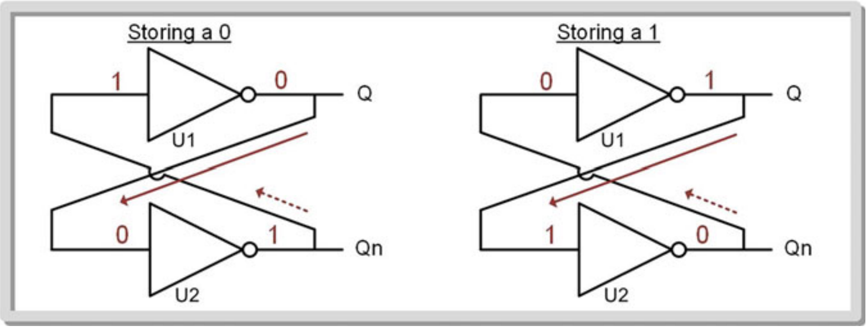
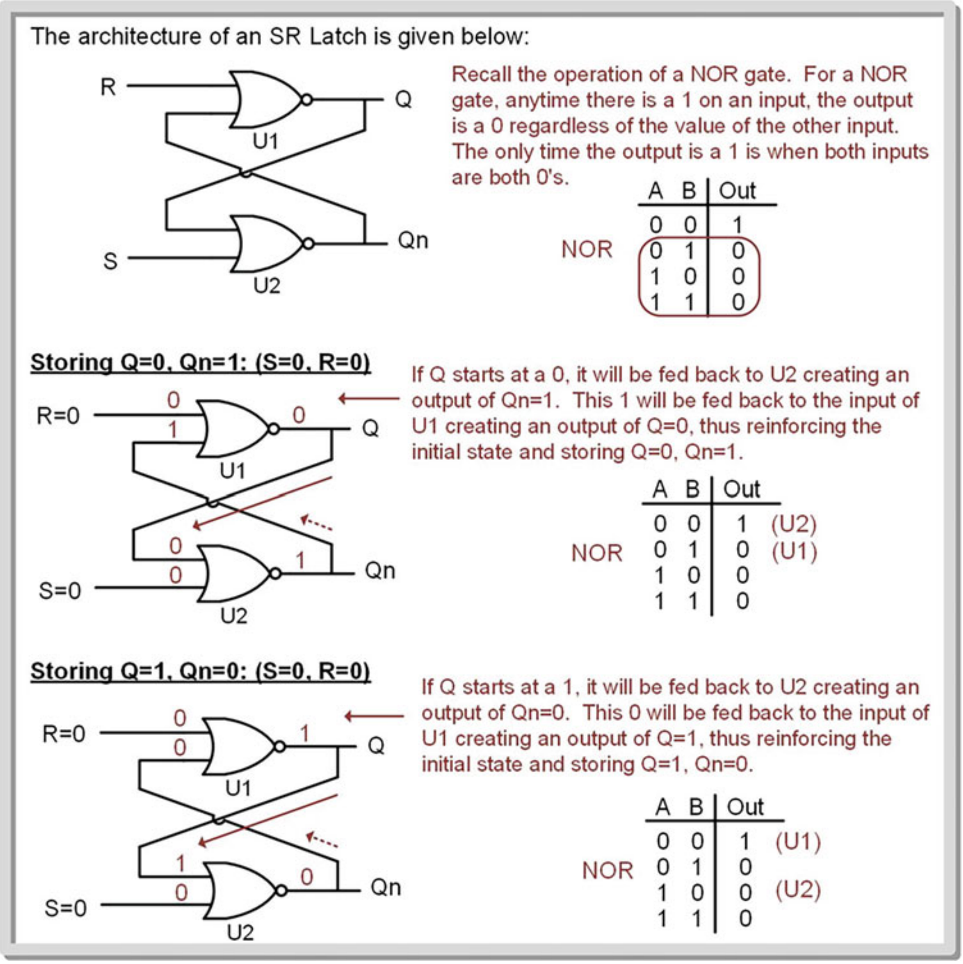


Fig. 2.18
Storage using a cross-coupled inverter pair

The cross-coupled inverter pair in Fig. 2.18 exhibits what is called *metastable* behavior due to its positive feedback configuration. Metastability refers to when a system can exist in a state of equilibrium when undisturbed but can be moved to a different, more stable state of equilibrium when sufficiently disturbed. Systems that exhibit *high levels* of metastability have an equilibrium state that is highly unstable, meaning that if disturbed even slightly the system will move rapidly to a more stable point of equilibrium. The cross-coupled inverter pair is a highly metastable system. This system actually contains three equilibrium states. The first is when the input of U1 is exactly between a logic 0 and logic 1 (i.e., $V_{CC}/2$). In this state, the output of U1 is also exactly $V_{CC}/2$. This voltage is fed back to the input of U2, thus producing an output of exactly $V_{CC}/2$ on U2. This in turn is fed back to the original input on U1 reinforcing the initial state. Despite this system being at equilibrium in this condition, this state is highly unstable. With minimal disturbance to any of the nodes within the system, it will move rapidly to one of two more stable states. The two stable states for this system are when $Q = 0$ or when $Q = 1$ (see Fig. 2.18). Once the transition begins between the unstable equilibrium state toward one of the two more stable states, the positive feedback in the system continually reinforces the transition until the system reaches its final state. In electrical systems, this initial disturbance is caused by the presence of *noise*, or unwanted voltage in the system. Noise can come from many sources including random thermal motion of charge carriers in the semiconductor materials, electromagnetic energy, or naturally occurring ionizing particles. Noise is present in every electrical system so the cross-coupled inverter pair will never be able to stay in the unstable equilibrium state where all nodes are at $\mathrm{V}_{\mathrm{CC}}/2$. The cross-coupled inverter pair does however have two highly stable states. Thus, this circuit is called a *bistable* element. The two stable states are shown in Fig. 2.18 and allow the circuit to store either a 1 or a 0.

2.3.1.2 The SR Latch

While the cross-coupled inverter pair is the fundamental storage concept for sequential logic, there is no mechanism to set the initial value of Q. All that is guaranteed is that the circuit will store a value in one of two stable states ($Q = 0$ or $Q = 1$). The *SR Latch* provides a means to control the initial values in this positive feedback configuration by replacing the inverters with NOR gates. In this circuit, S stands for *set* and indicates when the output is forced to a logic 1 ($Q = 1$), and R stands for *reset* and indicates when the output is forced to a logic 0 ($Q = 0$). When both $S = 0$ and $R = 0$, the SR Latch is put into a *store* mode and it will hold the last value of Q. In all of these input conditions, Qn is the complement of Q. Consider the behavior of the SR Latch during its store state shown in Fig. 2.19.



The architecture of an SR Latch is given below:

Recall the operation of a NOR gate. For a NOR gate, anytime there is a 1 on an input, the output is a 0 regardless of the value of the other input. The only time the output is a 1 is when both inputs are both 0's.

NOR

A	B	Out
0	0	1
0	1	0
1	0	0
1	1	0

Storing Q=0, Qn=1: (S=0, R=0)

If Q starts at a 0, it will be fed back to U2 creating an output of Qn=1. This 1 will be fed back to the input of U1 creating an output of Q=0, thus reinforcing the initial state and storing Q=0, Qn=1.

NOR

A	B	Out	
0	0	1	(U2)
0	1	0	(U1)
1	0	0	
1	1	0	

Storing Q=1, Qn=0: (S=0, R=0)

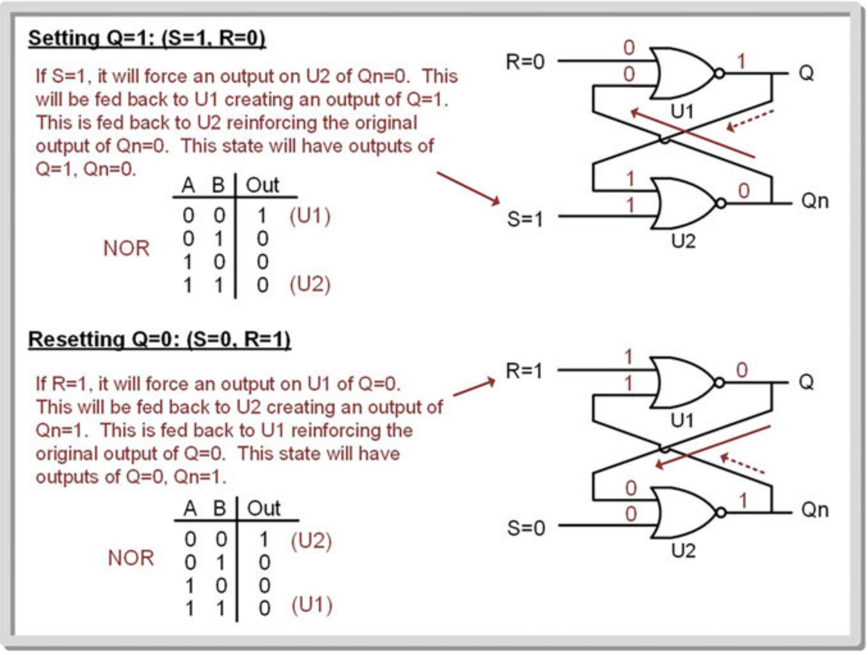
If Q starts at a 1, it will be fed back to U2 creating an output of Qn=0. This 0 will be fed back to the input of U1 creating an output of Q=1, thus reinforcing the initial state and storing Q=1, Qn=0.

NOR

A	B	Out	
0	0	1	(U1)
0	1	0	
1	0	0	(U2)
1	1	0	

Fig. 2.19
SR Latch behavior – store state ($S = 0$, $R = 0$)

The SR Latch has two input conditions that will force the outputs to known values. The first condition is called the *set* state. In this state, the inputs are configured as $S = 1$ and $R = 0$. This input condition will force the outputs to $Q = 1$ (e.g., setting Q) and $Q_n = 0$. The second input condition is called the *reset* state. In this state the inputs are configured as $S = 0$ and $R = 1$. This input condition will force the outputs to $Q = 0$ (i.e., resetting Q) and $Q_n = 1$. Consider the behavior of the SR Latch during its set and reset states shown in Fig. 2.20.



Setting Q=1: (S=1, R=0)

If S=1, it will force an output on U2 of Qn=0. This will be fed back to U1 creating an output of Q=1. This is fed back to U2 reinforcing the original output of Qn=0. This state will have outputs of Q=1, Qn=0.

NOR

A	B	Out	
0	0	1	(U1)
0	1	0	
1	0	0	
1	1	0	(U2)

Resetting Q=0: (S=0, R=1)

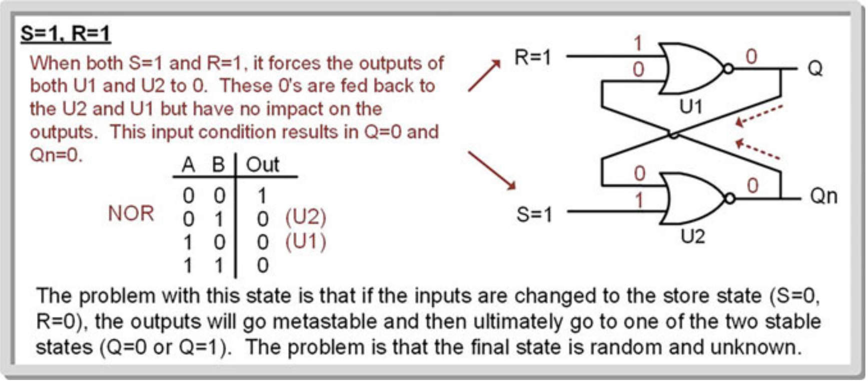
If R=1, it will force an output on U1 of Q=0. This will be fed back to U2 creating an output of Qn=1. This is fed back to U1 reinforcing the original output of Q=0. This state will have outputs of Q=0, Qn=1.

NOR

A	B	Out	
0	0	1	(U2)
0	1	0	
1	0	0	
1	1	0	(U1)

Fig. 2.20
SR Latch behavior – set ($S = 1$, $R = 0$) and reset ($S = 0$, $R = 1$) states

The final input condition for the SR Latch leads to potential metastability and should be avoided. When $S = 1$ and $R = 1$, the outputs of the SR Latch will both go to logic 0s. The problem with this state is that if the inputs subsequently change to the store state ($S = 0$, $R = 0$), the outputs will go metastable and then settle in one of the two stable states ($Q = 0$ or $Q = 1$). The reason this state is avoided is because the final resting state of the SR Latch is random and unknown. Consider this operation shown in Fig. 2.21.



S=1, R=1

When both S=1 and R=1, it forces the outputs of both U1 and U2 to 0. These 0's are fed back to the U2 and U1 but have no impact on the outputs. This input condition results in Q=0 and Qn=0.

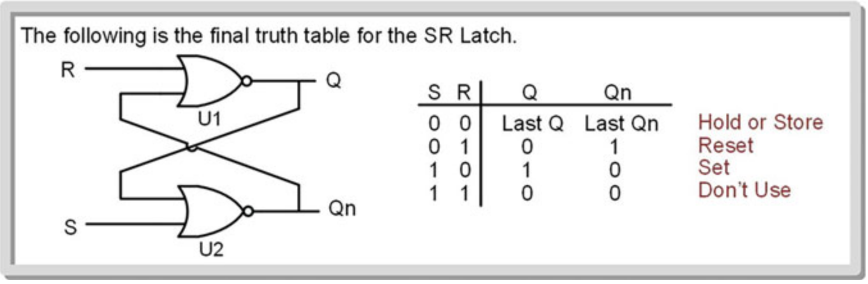
NOR

A	B	Out	
0	0	1	
0	1	0	(U2)
1	0	0	(U1)
1	1	0	

The problem with this state is that if the inputs are changed to the store state (S=0, R=0), the outputs will go metastable and then ultimately go to one of the two stable states (Q=0 or Q=1). The problem is that the final state is random and unknown.

Fig. 2.21
SR Latch behavior – don’t use state ($S = 1$ and $R = 1$)

Figure 2.22 shows the final truth table for the SR Latch.



The following is the final truth table for the SR Latch.

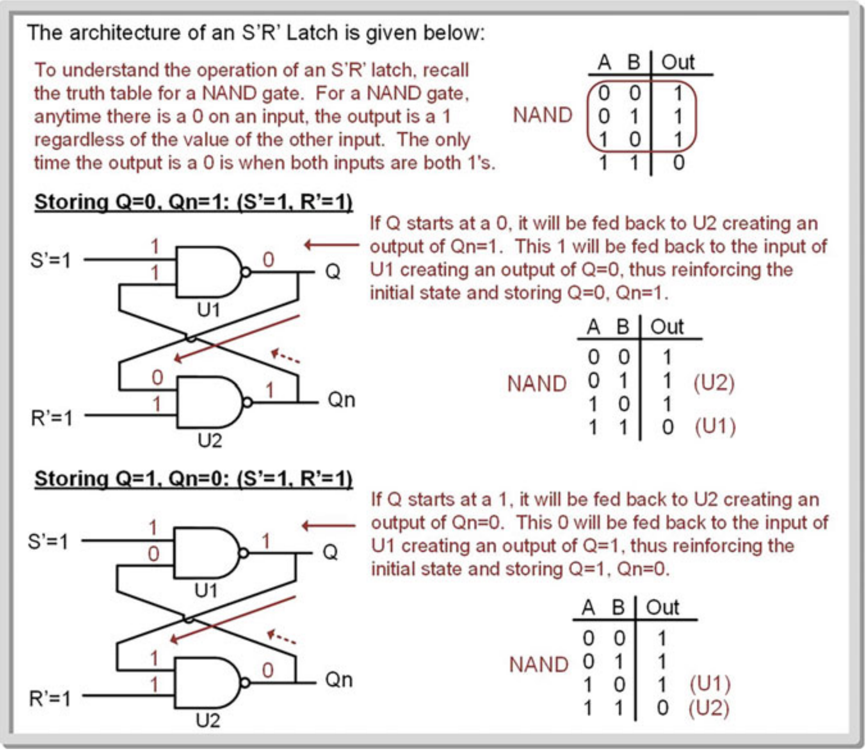
S	R	Q	Qn	
0	0	Last Q	Last Qn	Hold or Store
0	1	0	1	Reset
1	0	1	0	Set
1	1	0	0	Don't Use

Fig. 2.22
SR Latch truth table

The SR Latch has some drawbacks when it comes to implementation with real circuitry. First, it takes two independent inputs to control the outputs. Second, the state where $S = 1$ and $R = 1$ causes problems when real propagation delays are considered through the gates. Since it is impossible to match the delays exactly between U1 and U2, the SR Latch may occasionally enter this state and experience momentary metastable behavior. In order to address these issues, a number of improvements can be made to this circuit to create two of the most commonly used storage devices in sequential logic, the *D-Latch* and the *D-Flip-Flop*. In order to understand the operation of these storage devices, two incremental modifications are made to the SR Latch. The first is called the *S′R′ Latch* and the second is the *SR Latch with enable*. These two circuits are rarely implemented and are only explained to understand how the SR Latch is modified to create a D-Latch and ultimately a D-Flip-Flop.

2.3.1.3 The S′R′ Latch

The S′R′ Latch operates in a similar manner as the SR Latch with the exception that the input codes corresponding to the store, set, and reset states are complemented. To accomplish this complementary behavior, the S′R′ Latch is implemented with NAND gates configured in a positive feedback configuration. In this configuration, the S′R′ Latch will store the last output when $S' = 1$, $R' = 1$. It will set the output ($Q = 1$) when $S' = 0$, $R' = 1$. Finally, it will reset the output ($Q = 0$) when $S' = 1$, $R' = 0$. Consider the behavior of the S′R′ Latch during its store state shown in Fig. 2.23.



The architecture of an S'R' Latch is given below:

To understand the operation of an S'R' latch, recall the truth table for a NAND gate. For a NAND gate, anytime there is a 0 on an input, the output is a 1 regardless of the value of the other input. The only time the output is a 0 is when both inputs are both 1's.

NAND

A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0

Storing Q=0, Qn=1: (S'=1, R'=1)

If Q starts at a 0, it will be fed back to U2 creating an output of Qn=1. This 1 will be fed back to the input of U1 creating an output of Q=0, thus reinforcing the initial state and storing Q=0, Qn=1.

NAND

A	B	Out	
0	0	1	
0	1	1	(U2)
1	0	1	
1	1	0	(U1)

Storing Q=1, Qn=0: (S'=1, R'=1)

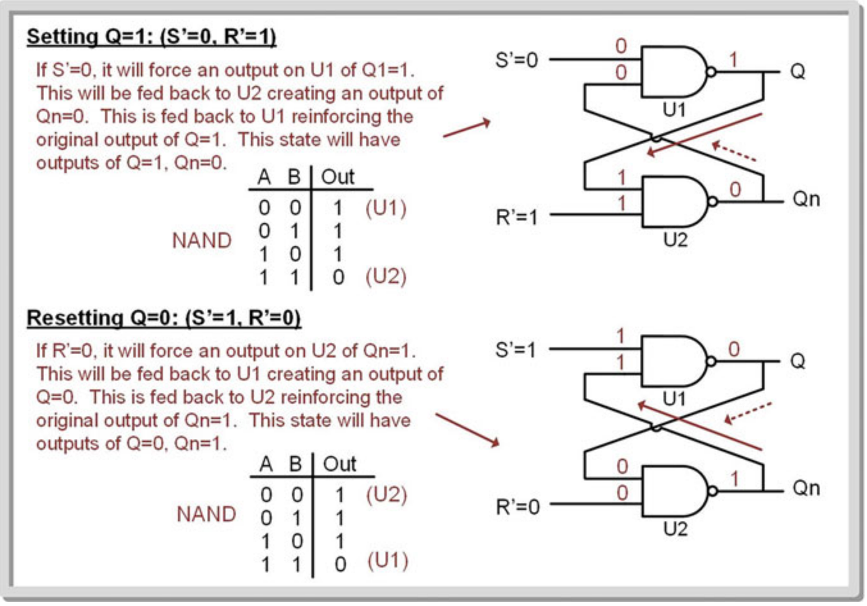
If Q starts at a 1, it will be fed back to U2 creating an output of Qn=0. This 0 will be fed back to the input of U1 creating an output of Q=1, thus reinforcing the initial state and storing Q=1, Qn=0.

NAND

A	B	Out	
0	0	1	
0	1	1	
1	0	1	(U1)
1	1	0	(U2)

Fig. 2.23
S′R′ Latch behavior – store state ($S' = 1$, $R' = 1$)

Just as with the SR Latch, the S′R′ Latch has two input configurations to control the values of the outputs. Consider the behavior of the S′R′ Latch during its set and reset states shown in Fig. 2.24.



Setting Q=1: (S'=0, R'=1)

If S'=0, it will force an output on U1 of Q1=1. This will be fed back to U2 creating an output of Qn=0. This is fed back to U1 reinforcing the original output of Q=1. This state will have outputs of Q=1, Qn=0.

NAND

A	B	Out	
0	0	1	(U1)
0	1	1	
1	0	1	
1	1	0	(U2)

Resetting Q=0: (S'=1, R'=0)

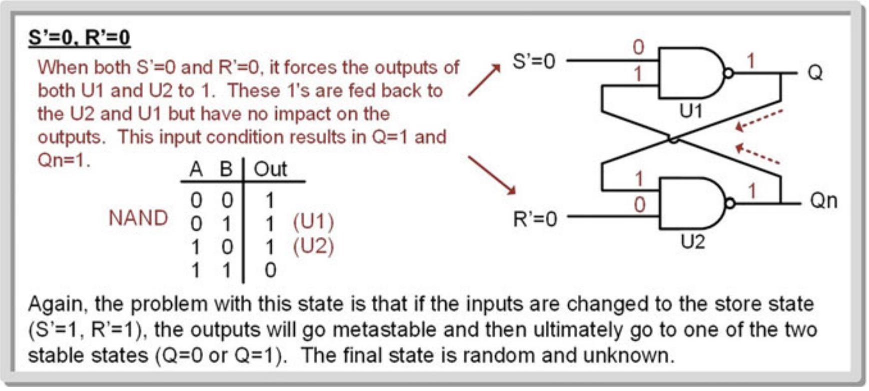
If R'=0, it will force an output on U2 of Qn=1. This will be fed back to U1 creating an output of Q=0. This is fed back to U2 reinforcing the original output of Qn=1. This state will have outputs of Q=0, Qn=1.

NAND

A	B	Out	
0	0	1	(U2)
0	1	1	
1	0	1	
1	1	0	(U1)

Fig. 2.24
S′R′ Latch behavior – set ($S' = 0$, $R' = 1$) and reset ($S' = 1$, $R' = 0$) states

And finally, just as with the SR Latch, the S′R′ Latch has a state that leads to potential metastability and should be avoided. Consider the operation of the S′R′ Latch when the inputs are configured as $S' = 0$ and $R' = 0$ shown in Fig. 2.25.



S'=0, R'=0

When both S'=0 and R'=0, it forces the outputs of both U1 and U2 to 1. These 1's are fed back to the U2 and U1 but have no impact on the outputs. This input condition results in Q=1 and Qn=1.

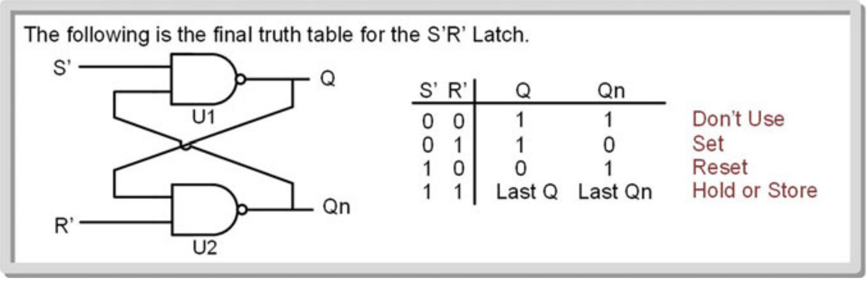
NAND

A	B	Out	
0	0	1	
0	1	1	(U1)
1	0	1	(U2)
1	1	0	

Again, the problem with this state is that if the inputs are changed to the store state (S'=1, R'=1), the outputs will go metastable and then ultimately go to one of the two stable states (Q=0 or Q=1). The final state is random and unknown.

Fig. 2.25
S′R′ Latch behavior – don’t use state ($S' = 0$ and $R' = 0$)

The final truth table for the S′R′ Latch is given in Fig. 2.26.



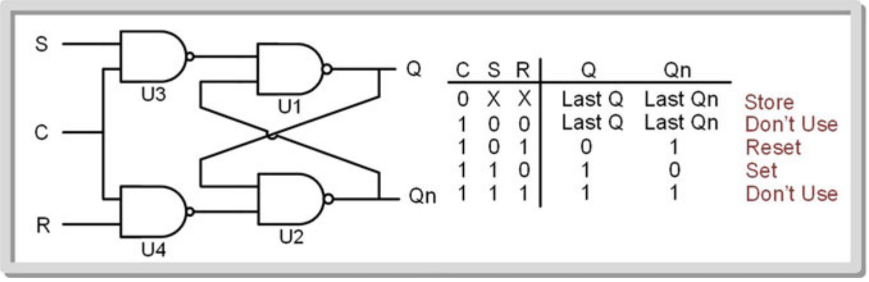
The following is the final truth table for the S'R' Latch.

S'	R'	Q	Qn	
0	0	1	1	Don't Use
0	1	1	0	Set
1	0	0	1	Reset
1	1	Last Q	Last Qn	Hold or Store

Fig. 2.26
S′R′ Latch truth table

2.3.1.4 SR Latch with Enable

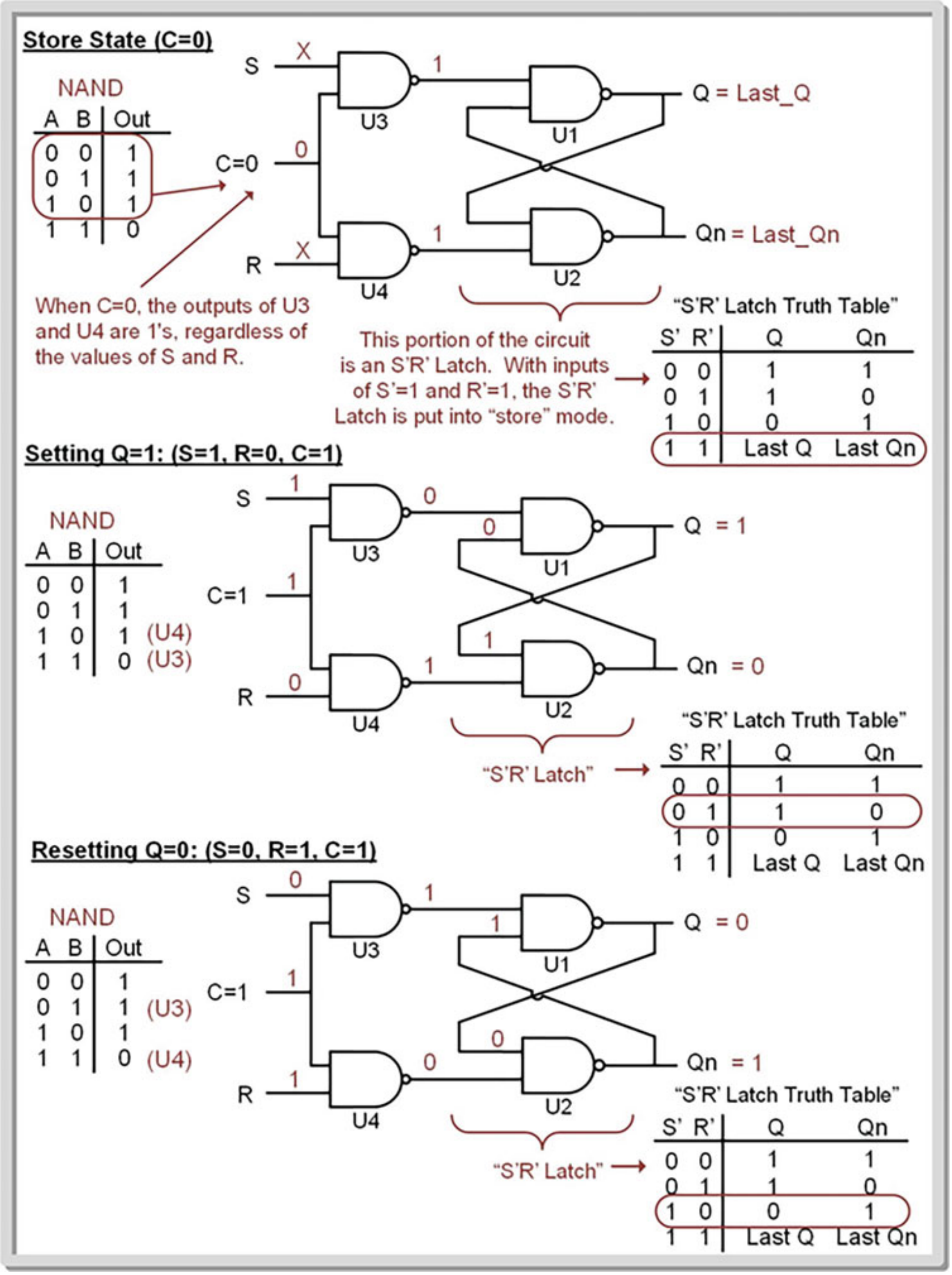
The next modification that is made in order to move toward a D-Latch and ultimately a D-Flip-Flop is to add an *enable* line to the S′R′ Latch. The enable is implemented by adding two NAND gates on the input stage of the S′R′ Latch. The SR Latch with enable is shown in Fig. 2.27. In this topology, the use of NAND gates changes the polarity of the inputs, so this circuit once again has a set state where $S = 1$, $R = 0$ and a reset state of $S = 0$, $R = 1$. The enable line is labeled C, which stands for *clock*. The rationale for this will be demonstrated upon moving through the explanation of the D-Latch.



C	S	R	Q	Qn	
0	X	X	Last Q	Last Qn	Store
1	0	0	Last Q	Last Qn	Don't Use
1	0	1	0	1	Reset
1	1	0	1	0	Set
1	1	1	1	1	Don't Use

Fig. 2.27
SR Latch with enable schematic

Recall that any time a 0 is present on one of the inputs to a NAND gate, the output will always be a 1 regardless of the value of the other inputs. In the SR Latch with enable configuration, any time $C = 0$, the outputs of U3 and U4 will be 1s and will be fed into the inputs of the cross-coupled NAND gate configuration (U1 and U2). Recall that the cross-coupled configuration of U1 and U2 is an S′R′ Latch and will be put into a store state when $S' = 1$ and $R' = 1$. This is the *store state* ($C = 0$). When $C = 1$, it has the effect of inverting the values of the S and R inputs before they reach U1 and U2. This condition allows the *set state* to be entered when $S = 1$, $R = 0$, $C = 1$ and the *reset state* to be entered when $S = 0$, $R = 1$, $C = 1$. Consider this operation in Fig. 2.28.



Store State (C=0)

NAND

A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0

When C=0, the outputs of U3 and U4 are 1's, regardless of the values of S and R.

This portion of the circuit is an S'R' Latch. With inputs of S'=1 and R'=1, the S'R' Latch is put into "store" mode.

"S'R' Latch Truth Table"

S'	R'	Q	Qn
0	0	1	1
0	1	1	0
1	0	0	1
1	1	Last Q	Last Qn

Setting Q=1: (S=1, R=0, C=1)

NAND

A	B	Out	
0	0	1	
0	1	1	
1	0	1	(U4)
1	1	0	(U3)

"S'R' Latch Truth Table"

S'	R'	Q	Qn
0	0	1	1
0	1	1	0
1	0	0	1
1	1	Last Q	Last Qn

Resetting Q=0: (S=0, R=1, C=1)

NAND

A	B	Out	
0	0	1	
0	1	1	(U3)
1	0	1	
1	1	0	(U4)

"S'R' Latch Truth Table"

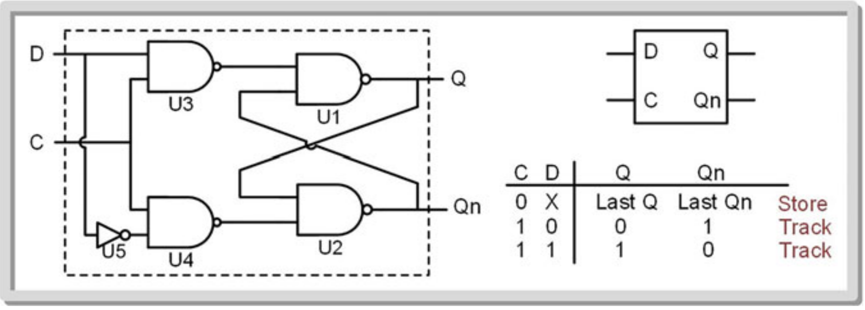
S'	R'	Q	Qn
0	0	1	1
0	1	1	0
1	0	0	1
1	1	Last Q	Last Qn

Fig. 2.28
SR Latch with enable behavior – store, set, and reset

Again, there is a potential metastable state when $S = 1$, $R = 1$ and $C = 1$ that should be avoided. There is also a second store state when $S = 0$, $R = 0$ and $C = 1$ that is not used because storage is to be dictated by the C input.

2.3.1.5 The D-Latch

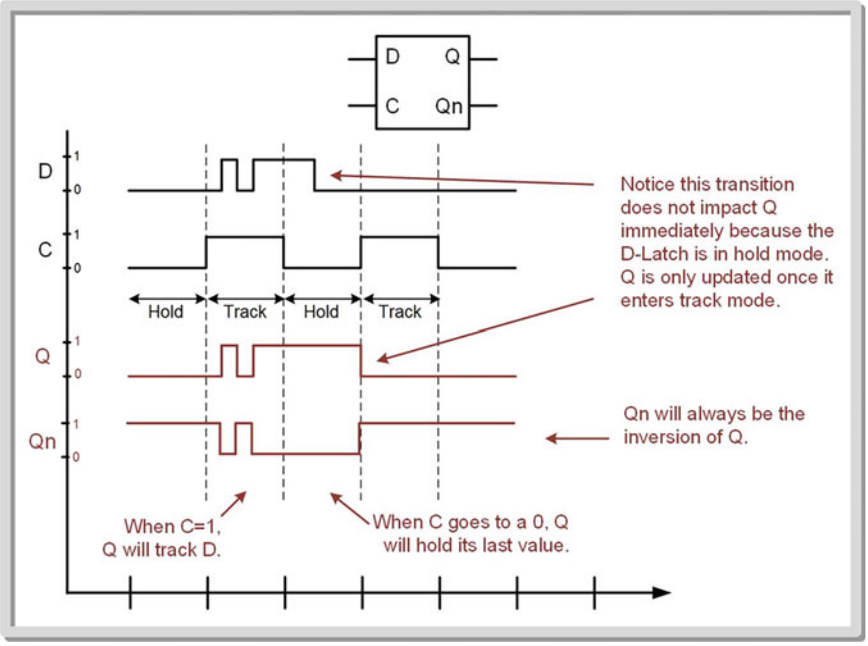
The SR Latch with enable can be modified to create a new storage device called a D-Latch. Instead of having two separate input lines to control the outputs of the latch, the *R* input of the latch is instead driven with an inverted version of the *S* input. This prevents the *S* and *R* inputs from ever being the same value and removes the two “Don’t Use” states in the truth table shown in Fig. 2.27. The new, single input is renamed D to stand for *data*. This new circuit still has the behavior that it will store the last value of Q and Qn when $C = 0$. When $C = 1$, the output will be $Q = 1$ when $D = 1$ and will be $Q = 0$ when $D = 0$. The behavior of the output when $C = 1$ is called *tracking* the input. The D-Latch schematic, symbol, and truth table are given in Fig. 2.29.



C	D	Q	Qn	
0	X	Last Q	Last Qn	Store
1	0	0	1	Track
1	1	1	0	Track

Fig. 2.29
D-Latch schematic, symbol, and truth table

The timing diagram for the D-Latch is shown in Fig. 2.30.



Notice this transition does not impact Q immediately because the D-Latch is in hold mode. Q is only updated once it enters track mode.

Qn will always be the inversion of Q.

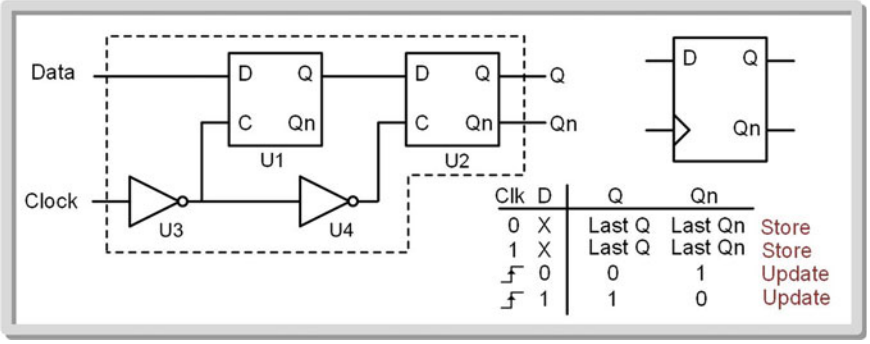
When C=1, Q will track D.

When C goes to a 0, Q will hold its last value.

Fig. 2.30
D-Latch timing diagram

2.3.1.6 The D-Flip-Flop

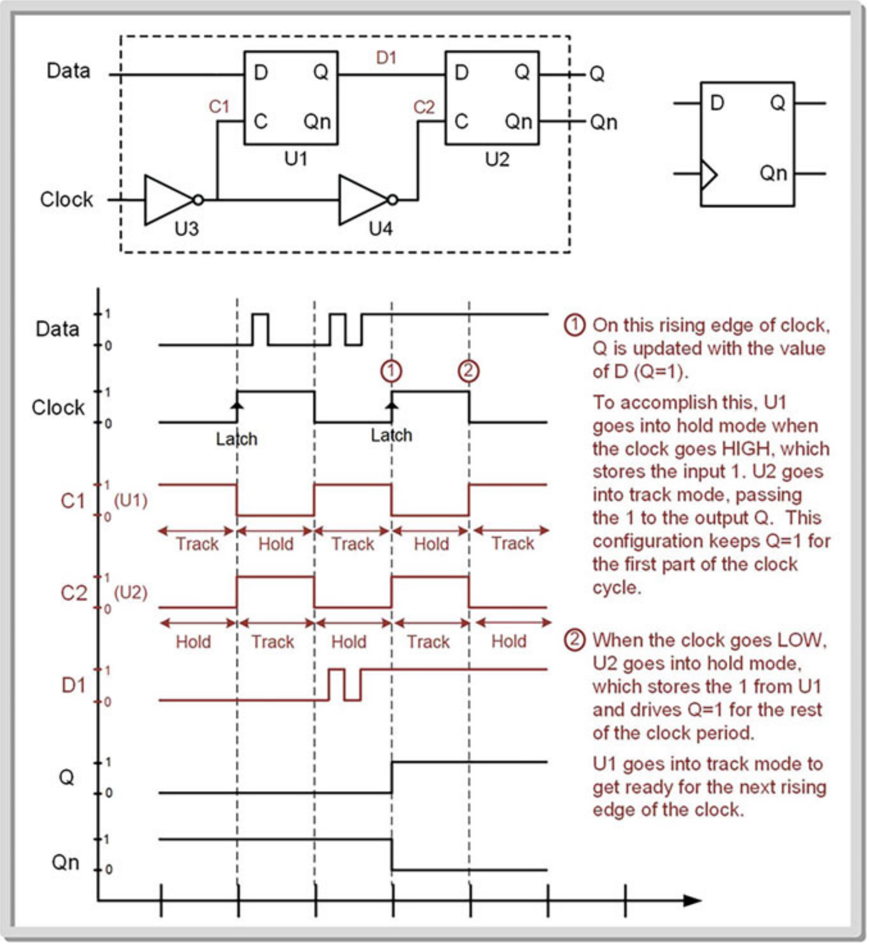
The final and most widely used storage device in sequential logic is the *D-Flip-Flop*. The D-Flip-Flop is similar in behavior to the D-Latch with the exception that the store mode is triggered by a transition, or edge, on the clock signal instead of a level. This allows the D-Flip-Flop to implement higher frequency systems since the outputs are updated in a shorter amount of time. The schematic, symbol, and truth table are given in Fig. 2.31 for a rising edge triggered D-Flip-Flop. To indicate that the device is edge sensitive, the input for the clock is designated with a ">". The U3 inverter in this schematic creates the rising edge behavior. If U3 is omitted, this circuit would be a negative edge triggered D-Flip-Flop.



Clk	D	Q	Qn	
0	X	Last Q	Last Qn	Store
1	X	Last Q	Last Qn	Store
↑	0	0	1	Update
↑	1	1	0	Update

Fig. 2.31
D-Flip-Flop (rising edge triggered) schematic, symbol, and truth table

The D-Flip-Flop schematic shown above is called a *master/slave* configuration because of how the data is passed through the two D-Latches (U1 and U2). Due to the U4 inverter, the two D-Latches will always be in complementary modes. When U1 is in hold mode, U2 will be in track mode and vice versa. When the clock signal transitions HIGH, U1 will store the last value of data. During the time when the clock is HIGH, U2 will enter track mode and pass this value to Q. In this way, the data is latched into the storage device on the rising edge of the clock and is present on Q. This is the *master* operation of the device because U1, or the first D-Latch, is holding the value, and the second D-Latch (the *slave*) is simply passing this value to the output Q. When the clock transitions LOW, U2 will store the output of U1. Since there is a finite delay through U1, the U2 D-Latch is able to store the value before U1 fully enters track mode. U2 will drive Q for the duration of the time that the clock is LOW. This is the *slave* operation of the device because U2, or the second D-Latch, is holding the value. During the time the clock is LOW, U1 is in track mode, which passes the input data to the middle of the D-Flip-Flop preparing for the next rising edge of the clock. The master/slave configuration creates a behavior where the Q output of the D-Flip-Flop is only updated with the value of D on a rising edge of the clock. At all other times, Q holds the last value of D. An example timing diagram for the operation of a rising edge D-Flip-Flop is given in Fig. 2.32.



① On this rising edge of clock, Q is updated with the value of D (Q=1).

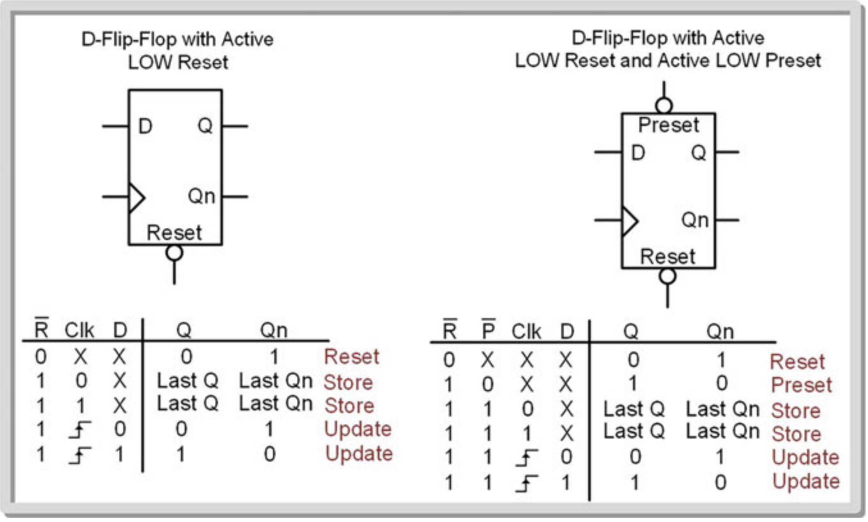
To accomplish this, U1 goes into hold mode when the clock goes HIGH, which stores the input 1. U2 goes into track mode, passing the 1 to the output Q. This configuration keeps Q=1 for the first part of the clock cycle.

② When the clock goes LOW, U2 goes into hold mode, which stores the 1 from U1 and drives Q=1 for the rest of the clock period.

U1 goes into track mode to get ready for the next rising edge of the clock.

Fig. 2.32
D-Flip-Flop (rising edge triggered) timing diagram

D-Flip-Flops often have additional signals that will set the initial conditions of the outputs that are separate from the clock. A *reset* input is used to force the outputs to Q = 0, $Q_n = 1$. A *preset* input is used to force the outputs to Q = 1, $Q_n = 0$. In most modern D-Flip-Flops, these inputs are active LOW, meaning that the line is asserted when the input is a 0. Active LOW inputs are indicated by placing an inversion bubble on the input pin of the symbol. These lines are typically *asynchronous*, meaning that when they are asserted, action is immediately taken to alter the outputs. This is different from a *synchronous* input in which action is only taken on the edge of the clock. Figure 2.33 shows the symbols and truth tables for two D-Flip-Flop variants, one with an active LOW reset and another with both an active LOW reset and active LOW preset.

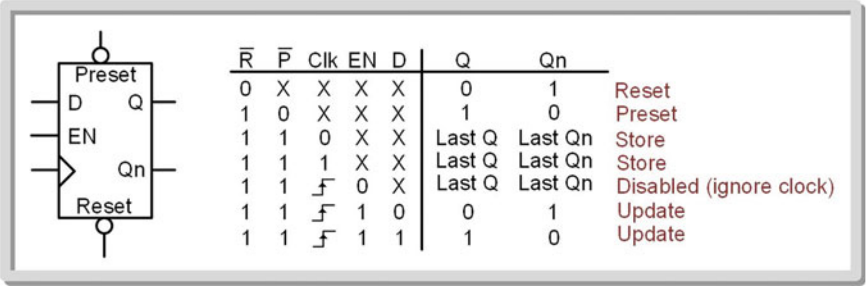

D-Flip-Flop with Active LOW Reset
D
Q
Qn
Reset
D-Flip-Flop with Active LOW Reset and Active LOW Preset
Preset
D
Q
Qn
Reset

$\overline{R}$	Clk	D	Q	Qn	
0	X	X	0	1	Reset
1	0	X	Last Q	Last Qn	Store
1	1	X	Last Q	Last Qn	Store
1	↑	0	0	1	Update
1	↑	1	1	0	Update

$\overline{R}$	$\overline{P}$	Clk	D	Q	Qn	
0	X	X	X	0	1	Reset
1	0	X	X	1	0	Preset
1	1	0	X	Last Q	Last Qn	Store
1	1	1	X	Last Q	Last Qn	Store
1	1	↑	0	0	1	Update
1	1	↑	1	1	0	Update

Fig. 2.33
D-Flip-Flop with asynchronous reset and preset

D-Flip-Flops can also be created with an *enable* line. An enable line controls whether or not the output is updated. Enable lines are synchronous, meaning that when they are asserted, the outputs will be updated on the rising edge of the clock. When de-asserted, the outputs are not updated. This behavior in effect ignores the clock input when de-asserted. Figure 2.34 shows the symbol and truth table for a D-Flip-Flop with a synchronous enable.



$\overline{R}$	$\overline{P}$	Clk	EN	D	Q	Qn	
0	X	X	X	X	0	1	Reset
1	0	X	X	X	1	0	Preset
1	1	0	X	X	Last Q	Last Qn	Store
1	1	1	X	X	Last Q	Last Qn	Store
1	1	↑	0	X	Last Q	Last Qn	Disabled (ignore clock)
1	1	↑	1	0	0	1	Update
1	1	↑	1	1	1	0	Update

Fig. 2.34
D-Flip-Flop with synchronous enable

There are a variety of timing specifications that need to be met in order to successfully design circuits using sequential storage devices. The first specification is called the *setup time* (t_{setup} or t_s). The setup time specifies how long the data input needs to be at a steady state *before* the clock event. The second specification is called the *hold time* (t_{hold} or t_h). The hold time specifies how long the data input needs to be at a steady state *after* the clock event. If these specifications are violated (i.e., the input transitions too close to the clock transition), the storage device will not be able to determine whether the input was a 1 or 0 and will go metastable. The time a storage device will remain metastable is a deterministic value and is specified by the part manufacturer (t_{meta}). In general, metastability should be avoided; however, knowing the maximum duration of metastability for a storage device allows us to design circuits to overcome potential metastable conditions. During the time the device is metastable, the output will have random behavior. It may go to a steady state 1, a steady state 0, or toggle between a

0 and 1 uncontrollably. Once the device comes out of metastability, it will come to rest in one of its two stable states ($Q = 0$ or $Q = 1$). The final resting state is random and unknown. Another specification for sequential storage devices is the delay from the time a clock transition occurs to the point that the data is present on the Q output. This specification is called the *Clock-to-Q* delay and is given the notation t_{CQ}. These specifications are shown in Fig. 2.35.

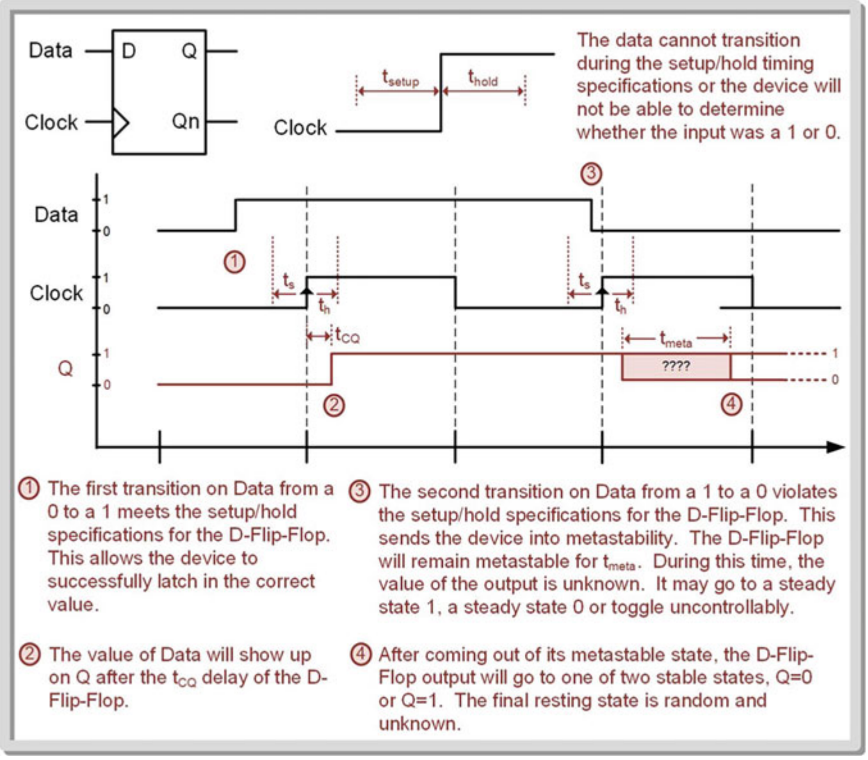

Data
D
Q
Clock
Qn
Clock
t_{setup}
t_{hold}
The data cannot transition during the setup/hold timing specifications or the device will not be able to determine whether the input was a 1 or 0.
Data
1
0
Clock
1
0
Q
1
0
t_s
t_h
t_{CQ}
t_{meta}
????
① ② ③ ④
① The first transition on Data from a 0 to a 1 meets the setup/hold specifications for the D-Flip-Flop. This allows the device to successfully latch in the correct value.
② The value of Data will show up on Q after the t_{CQ} delay of the D-Flip-Flop.
③ The second transition on Data from a 1 to a 0 violates the setup/hold specifications for the D-Flip-Flop. This sends the device into metastability. The D-Flip-Flop will remain metastable for t_{meta}. During this time, the value of the output is unknown. It may go to a steady state 1, a steady state 0 or toggle uncontrollably.
④ After coming out of its metastable state, the D-Flip-Flop output will go to one of two stable states, Q=0 or Q=1. The final resting state is random and unknown.

Fig. 2.35
Sequential storage device timing specifications

The behavior of the D-Flip-Flop allows us to design systems that are *synchronous* to a clock signal. A clock signal is a periodic square wave that dictates when events occur in a digital system. A synchronous system based on D-Flip-Flops will allow the outputs of its storage devices to be updated upon a rising edge of the clock. This is advantageous because when the Q outputs are storing values they can be used as inputs for combinational logic circuits. Since combinational logic circuits contain a certain amount of propagation delay before the final output is calculated, the D-Flip-Flop can hold the inputs at a steady value while the output is generated. Since the input on a D-Flip-Flop is ignored during all other times, the output of a combinational logic circuit can be fed back as an input to a D-Flip-Flop. This gives a system the ability to generate outputs based on the current values of inputs in addition to past values of the inputs that are being held on the outputs of D-Flip-Flops. This is the definition of sequential logic. An example synchronous, sequential system is shown in Fig. 2.36.

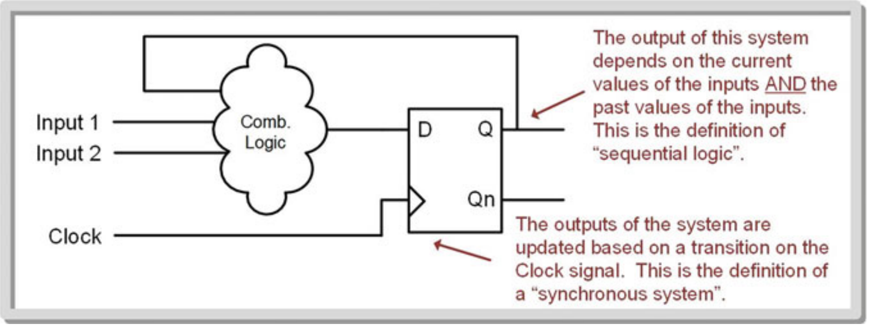

Input 1
Input 2
Clock
Comb. Logic
D
Q
Qn
The output of this system depends on the current values of the inputs AND the past values of the inputs. This is the definition of "sequential logic".
The outputs of the system are updated based on a transition on the Clock signal. This is the definition of a "synchronous system".

Fig. 2.36
An example of synchronous system based on a D-Flip-Flop

2.3.1.7 Registers

In modern digital systems, it is common to store data in groups. A group of signals is called a *bus*. D-Flip-Flops can be put together to form an *n*-bit storage device known as a *register*. A register consists of D-Flip-Flops each with their clock, EN, and reset lines tied together. The lines in the data bus are connected to the individual D inputs of the D-Flip-Flop. In this way, a bus can be stored synchronously when the register is enabled. Figure 2.37 shows an example of how to construct a 4-bit register.

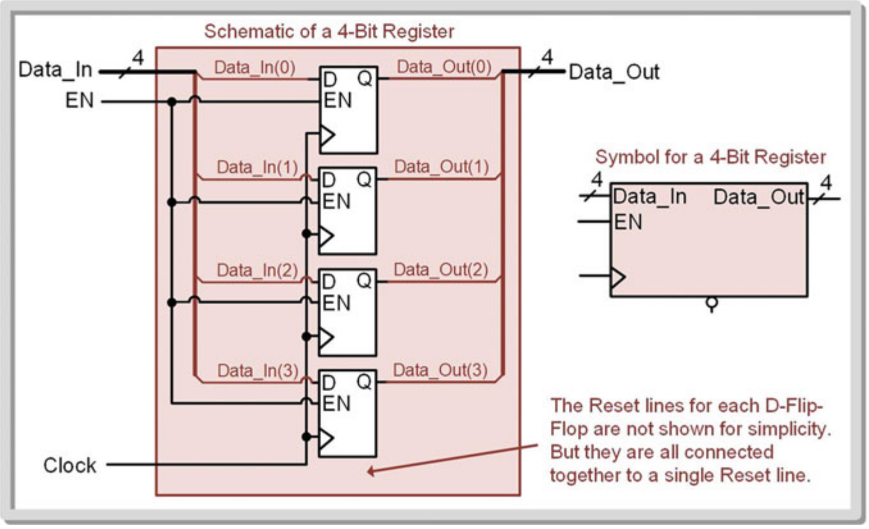

Schematic of a 4-Bit Register
Data_In
4
EN
Clock
Data_In(0)
Data_In(1)
Data_In(2)
Data_In(3)
D Q
EN
Data_Out(0)
Data_Out(1)
Data_Out(2)
Data_Out(3)
4
Data_Out
Symbol for a 4-Bit Register
4
Data_In
Data_Out
4
EN
The Reset lines for each D-Flip-Flop are not shown for simplicity. But they are all connected together to a single Reset line.

Fig. 2.37
4-bit register

Concept Check

CC2.3.1 If the clock for a register is always running, how does the register avoid continuously latching in data?

A) It does continuously latch in data. It just requires that the data is always the same so that the outputs don't change.

B) Each D-flip-flop in the register has an enable line that is used to control when data is latched in.

C) The register requires the system to halt the clock when it is not in use.

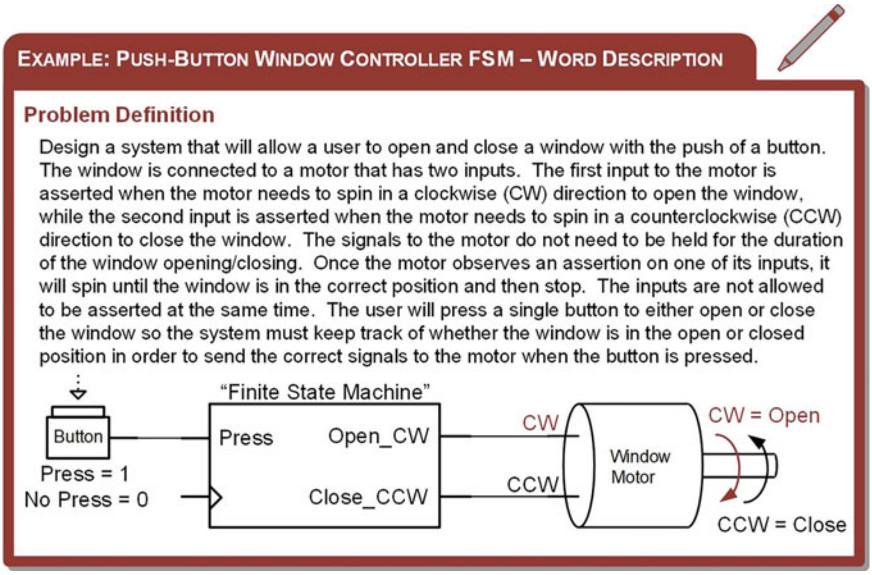
D) The D input is disconnected when not in use.

2.3.2 Finite State Machines

Now we turn our attention to one of the most powerful sequential logic circuits, the finite state machine (FSM). A FSM, or *state machine*, is a circuit that contains a predefined number of states (i.e., a finite number of states). The machine can exist in one and only one state at a time. The circuit *transitions* between states based on a triggering event, most commonly the edge of a clock, in addition to the values of any inputs of the machine. The number of states and all possible transitions are predefined. Through the use of states and a predefined sequence of transitions, the circuit is able to make decisions on the next state to transition to based on a history of past states. This allows the circuit to create outputs that are more intelligent compared to a simple combinational logic circuit that has outputs based only on the current values of the inputs.

2.3.2.1 Describing the Functionality of a FSM

The design of a state machine begins with an abstract word description of the desired circuit behavior. We will use a design example of a push-button motor controller to describe all of the steps involved in creating a finite state machine. Example 2.43 starts the FSM design process by stating the word description of the system.



EXAMPLE: PUSH-BUTTON WINDOW CONTROLLER FSM – WORD DESCRIPTION

Problem Definition

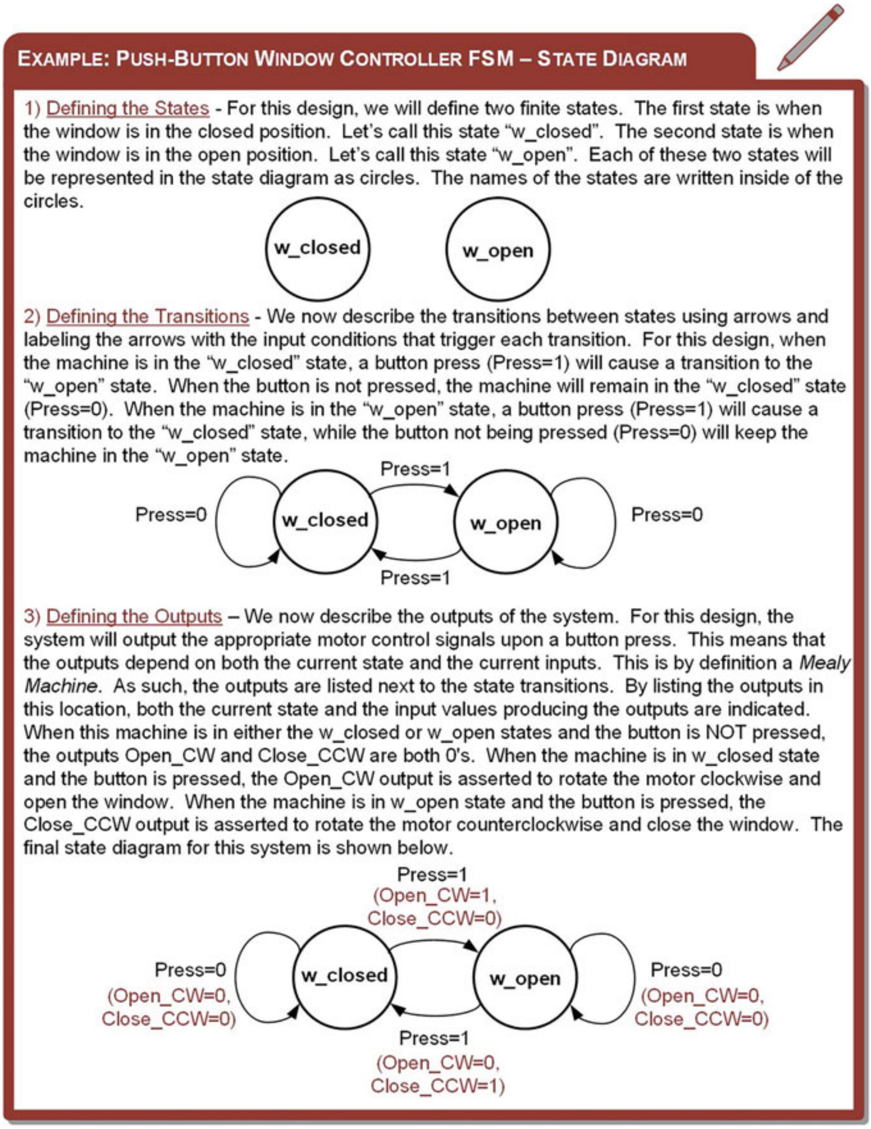
Design a system that will allow a user to open and close a window with the push of a button. The window is connected to a motor that has two inputs. The first input to the motor is asserted when the motor needs to spin in a clockwise (CW) direction to open the window, while the second input is asserted when the motor needs to spin in a counterclockwise (CCW) direction to close the window. The signals to the motor do not need to be held for the duration of the window opening/closing. Once the motor observes an assertion on one of its inputs, it will spin until the window is in the correct position and then stop. The inputs are not allowed to be asserted at the same time. The user will press a single button to either open or close the window so the system must keep track of whether the window is in the open or closed position in order to send the correct signals to the motor when the button is pressed.

Example 2.43
Push-button window controller – word description

2.3.2.2 State Diagrams

A state diagram is a graphical way to describe the functionality of a finite state machine. A state diagram is a form of a directed graph, in which each state (or vertex) within the system is denoted as a circle and given a descriptive name. The names are written inside of the circles. The transitions between states are denoted using arrows with the input conditions causing the transitions written next to them. Transitions (or edges) can move to different states upon particular input conditions or remain in the same state. For a state machine implemented using sequential logic storage, an evaluation of when to transition states is triggered every time the storage devices update their outputs. For example, if the system was implemented using rising edge triggered D-Flip-Flops, then an evaluation would occur on every rising edge of the clock.

There are two different types of output conditions for a state machine. The first is when the output only depends on the current state of the machine. This type of system is called a *Moore machine*. In this case, the outputs of the system are written inside of the state circles. This indicates the output value that will be generated for each specific state. The second output condition is when the outputs depend on both the current state and the system inputs. This type of system is called a *Mealy machine*. In this case, the outputs of the system are written next to the state transitions corresponding to the appropriate input values. Outputs in a state diagram are typically written inside of parentheses. Example 2.44 shows the construction of the state diagram for our push-button window controller design.



EXAMPLE: PUSH-BUTTON WINDOW CONTROLLER FSM – STATE DIAGRAM

1) Defining the States - For this design, we will define two finite states. The first state is when the window is in the closed position. Let's call this state "w_closed". The second state is when the window is in the open position. Let's call this state "w_open". Each of these two states will be represented in the state diagram as circles. The names of the states are written inside of the circles.

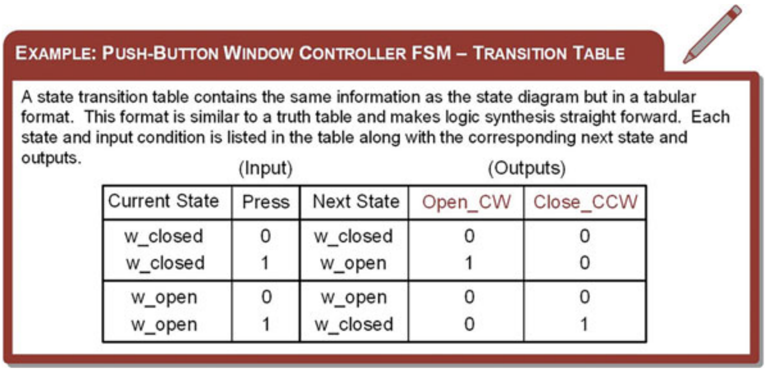
2) Defining the Transitions - We now describe the transitions between states using arrows and labeling the arrows with the input conditions that trigger each transition. For this design, when the machine is in the "w_closed" state, a button press (Press=1) will cause a transition to the "w_open" state. When the button is not pressed, the machine will remain in the "w_closed" state (Press=0). When the machine is in the "w_open" state, a button press (Press=1) will cause a transition to the "w_closed" state, while the button not being pressed (Press=0) will keep the machine in the "w_open" state.

3) Defining the Outputs – We now describe the outputs of the system. For this design, the system will output the appropriate motor control signals upon a button press. This means that the outputs depend on both the current state and the current inputs. This is by definition a *Mealy Machine*. As such, the outputs are listed next to the state transitions. By listing the outputs in this location, both the current state and the input values producing the outputs are indicated. When this machine is in either the w_closed or w_open states and the button is NOT pressed, the outputs Open_CW and Close_CCW are both 0's. When the machine is in w_closed state and the button is pressed, the Open_CW output is asserted to rotate the motor clockwise and open the window. When the machine is in w_open state and the button is pressed, the Close_CCW output is asserted to rotate the motor counterclockwise and close the window. The final state diagram for this system is shown below.

Example 2.44
Push-button window controller – state diagram

2.3.2.3 State Transition Tables

The state diagram can now be described in a table format that is similar to a truth table. This puts the state machine behavior in a form that makes logic synthesis straightforward. The table contains the same information as in the state diagram. The state that the machine exists in is called the *current state*. For each current state that the machine can reside in, every possible input condition is listed along with the destination state of each transition. The destination state for a transition is called the *next state*. Also listed in the table are the outputs corresponding to each current state and, in the case of a Mealy machine, the output corresponding to each input condition. Example 2.45 shows the construction of the state transition table for the push-button window controller design. This information is identical to the state diagram given in Example 2.44.



EXAMPLE: PUSH-BUTTON WINDOW CONTROLLER FSM – TRANSITION TABLE

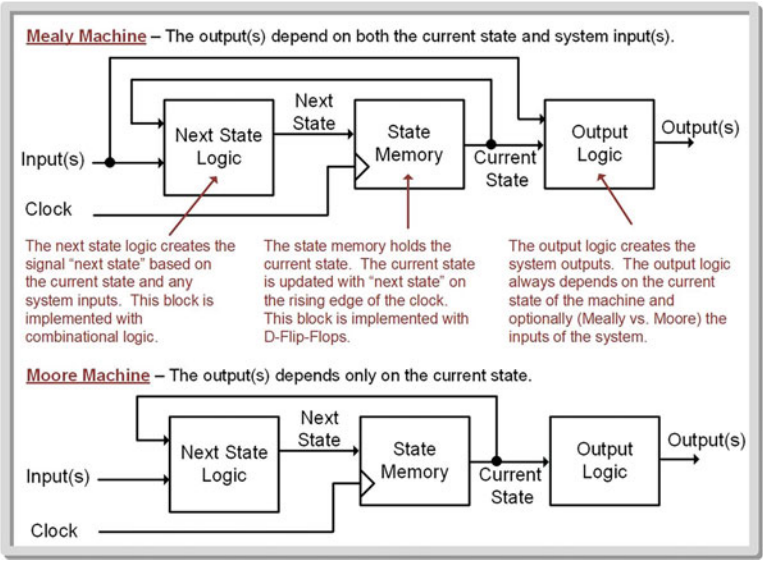
A state transition table contains the same information as the state diagram but in a tabular format. This format is similar to a truth table and makes logic synthesis straight forward. Each state and input condition is listed in the table along with the corresponding next state and outputs.

	(Input)		(Outputs)	
Current State	Press	Next State	Open_CW	Close_CCW
w_closed	0	w_closed	0	0
w_closed	1	w_open	1	0
w_open	0	w_open	0	0
w_open	1	w_closed	0	1

Example 2.45
Push-button window controller – state transition table

2.3.2.4 Logic Synthesis for a FSM

Once the behavior of the state machine has been described, it can be directly synthesized. There are three main components of a state machine: the state memory, the next state logic, and the output logic. Figure 2.38 shows a block diagram of a state machine highlighting these three components. The *next state logic* block is a group of combinational logic that produces the next state signals based on the current state and any system inputs. The *state memory* holds the current state of the system. The current state is updated with next state on every rising edge of the clock, which is indicated with the ">" symbol within the block. This behavior is created using D-Flip-Flops where the current state is held on the Q outputs of the D-Flip-Flops, while the next state is present on the D inputs of the D-Flip-Flops. In this way, every rising edge of the clock will trigger an evaluation of which state to move to next. This decision is based on the current state and the current inputs. The *output logic* block is a group of combinational logic that creates the outputs of the system. This block always uses the current state as an input and, depending on the type of machine (Mealy vs. Moore), uses the system inputs. It is useful to keep this block diagram in mind when synthesizing finite state machines as it will aid in keeping the individual design steps separate and clear.



Mealy Machine – The output(s) depend on both the current state and system input(s).

The next state logic creates the signal "next state" based on the current state and any system inputs. This block is implemented with combinational logic.

The state memory holds the current state. The current state is updated with "next state" on the rising edge of the clock. This block is implemented with D-Flip-Flops.

The output logic creates the system outputs. The output logic always depends on the current state of the machine and optionally (Mealy vs. Moore) the inputs of the system.

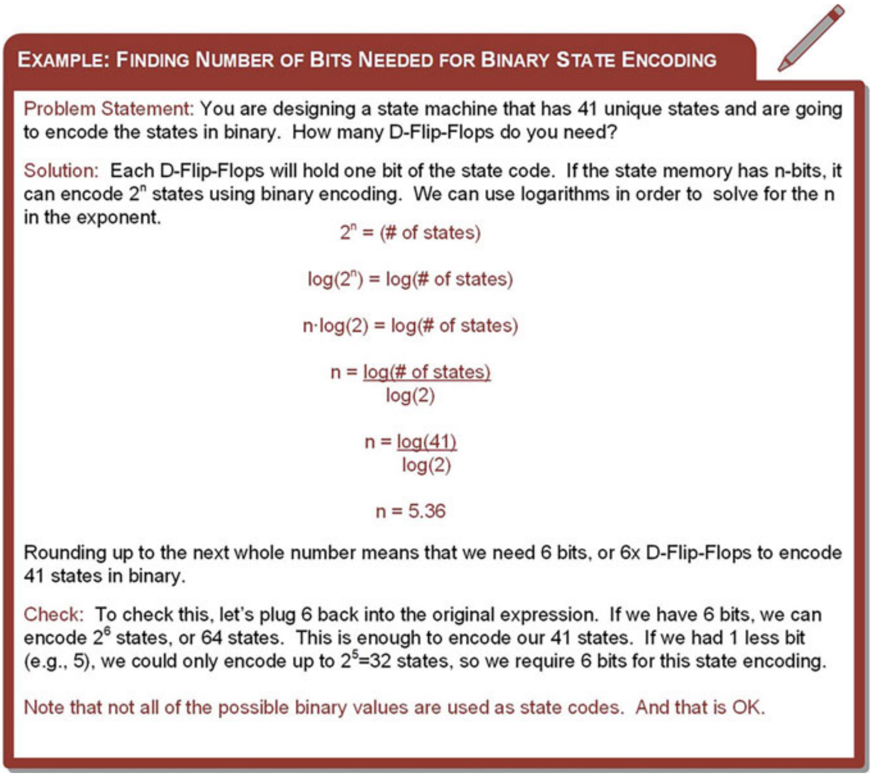
Moore Machine – The output(s) depends only on the current state.

Fig. 2.38
Main components of a finite state machine

2.3.2.5 State Memory

The state memory is the circuitry that will hold the current state of the machine. Upon a rising edge of a clock it will update the current state with the next state. At all other times, the next state input is ignored. This gives time for the next state logic circuitry to compute the results for the next state. This behavior is identical to that of a D-Flip-Flop; thus, the state memory is simply one or more D-Flip-Flops. The number of D-Flip-Flops required depends on how the states are *encoded*. *State encoding* is the process of assigning a binary value to the descriptive names of the states from the state diagram and state transition tables. Once the descriptive names have been converted into representative codes using 1s and 0s, the states can be implemented in real circuitry. The assignment of codes is arbitrary and can be selected in order to minimize the circuitry needed in the machine.

There are three main styles of state encoding. The first is straight *binary encoding*. In this approach the state codes are simply a set of binary counts (i.e., 00, 01, 10, 11. . .). The binary counts are assigned starting at the beginning of the state diagram and incrementally assigned toward the end. This type of encoding has the advantage that it is very efficient in minimizing the number of D-Flip-Flops needed for the state memory. With *n* D-Flip-Flops, 2^n states can be encoded. When a large number of states is required, the number of D-Flip-Flops can be calculated using the rules of logarithmic math. Example 2.46 shows how to solve for the number of bits needed in the binary state code based on the number of states in the machine.



EXAMPLE: FINDING NUMBER OF BITS NEEDED FOR BINARY STATE ENCODING

Problem Statement: You are designing a state machine that has 41 unique states and are going to encode the states in binary. How many D-Flip-Flops do you need?

Solution: Each D-Flip-Flops will hold one bit of the state code. If the state memory has n-bits, it can encode 2^n states using binary encoding. We can use logarithms in order to solve for the n in the exponent.

$$2^n = (\#\text{ of states})$$

$$\log(2^n) = \log(\#\text{ of states})$$

$$n\cdot\log(2) = \log(\#\text{ of states})$$

$$n = \frac{\log(\#\text{ of states})}{\log(2)}$$

$$n = \frac{\log(41)}{\log(2)}$$

$$n = 5.36$$

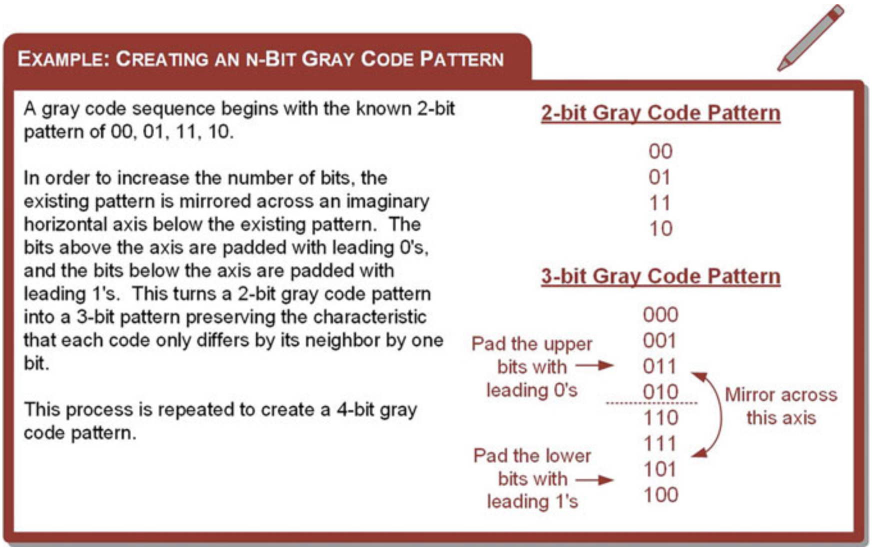
Rounding up to the next whole number means that we need 6 bits, or 6x D-Flip-Flops to encode 41 states in binary.

Check: To check this, let's plug 6 back into the original expression. If we have 6 bits, we can encode 2^6 states, or 64 states. This is enough to encode our 41 states. If we had 1 less bit (e.g., 5), we could only encode up to 2^5=32 states, so we require 6 bits for this state encoding.

Note that not all of the possible binary values are used as state codes. And that is OK.

Example 2.46
Finding the number of bits needed for binary state encoding

The second type of state encoding is called *gray code encoding*. A gray code is one in which the value of a code differs by only one bit from any of its neighbors, (i.e., 00, 01, 11, 10. . .). A gray code is useful for reducing the number of bit transitions on the state codes when the machine has a transition sequence that is linear. Reducing the number of bit transitions can reduce the amount of power consumption and noise generated by the circuit. When the state transitions of a machine are highly non-linear, a gray code encoding approach does not provide any benefit. Gray code is also an efficient coding approach. With *n* D-Flip-Flops, 2^n states can be encoded just as in binary encoding. Figure 2.39 shows the process of creating *n*-bit, gray code patterns.



EXAMPLE: CREATING AN N-BIT GRAY CODE PATTERN

A gray code sequence begins with the known 2-bit pattern of 00, 01, 11, 10.

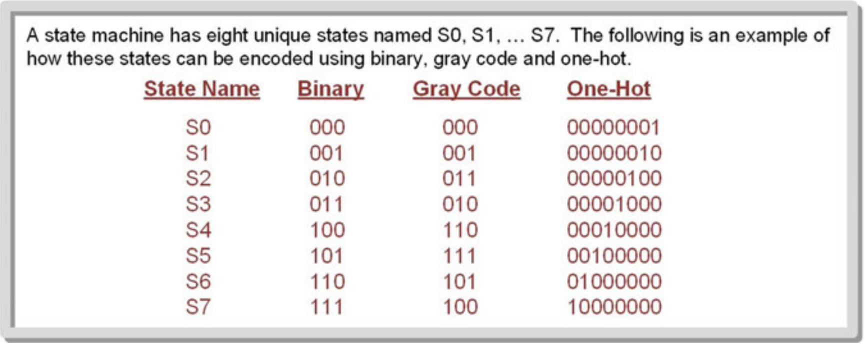
In order to increase the number of bits, the existing pattern is mirrored across an imaginary horizontal axis below the existing pattern. The bits above the axis are padded with leading 0's, and the bits below the axis are padded with leading 1's. This turns a 2-bit gray code pattern into a 3-bit pattern preserving the characteristic that each code only differs by its neighbor by one bit.

This process is repeated to create a 4-bit gray code pattern.

Fig. 2.39
Creating an *n*-bit gray code pattern

The third common technique to encode states is using *one-hot encoding*. In this approach, a separate D-Flip-Flop is asserted for each state in the machine. For an *n-state* machine, this encoding approach requires *n* D-Flip-Flops. For example, if a machine had three states, the one-hot state codes would be “001,” “010,” and “100.” This approach has the advantage that the next state logic circuitry is very simple; further, there is less chance that the different propagation delays through the next state logic will cause an inadvertent state to be entered. This approach is not as efficient as binary and gray code in terms of minimizing the number of D-Flip-Flops because it requires one D-Flip-Flop for each state; however, in modern digital integrated circuits that have abundant D-Flip-Flops, one-hot encoding is commonly used.

Figure 2.40 shows the differences between these three state encoding approaches.



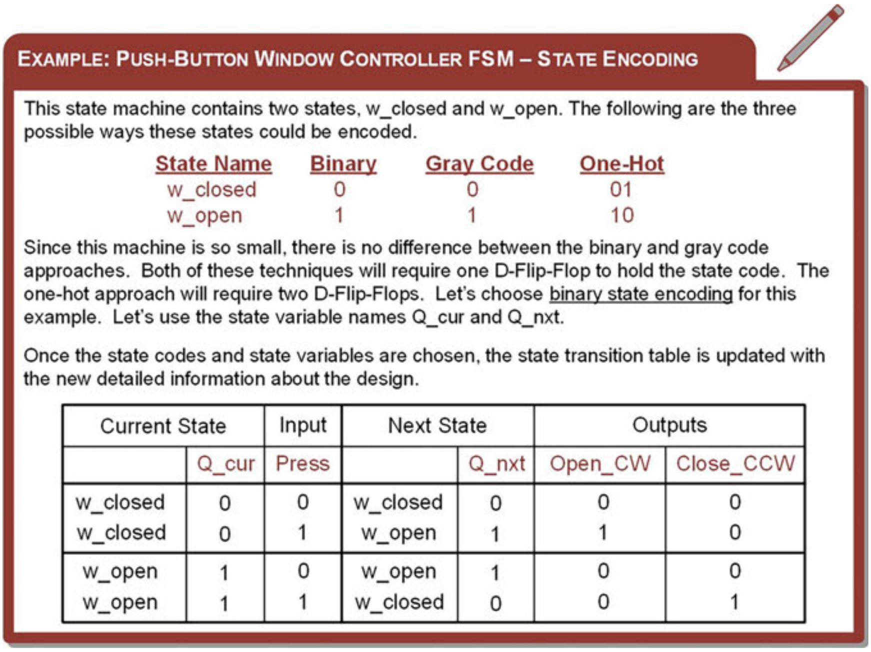
A state machine has eight unique states named S0, S1, … S7. The following is an example of how these states can be encoded using binary, gray code and one-hot.

State Name	Binary	Gray Code	One-Hot
S0	000	000	00000001
S1	001	001	00000010
S2	010	011	00000100
S3	011	010	00001000
S4	100	110	00010000
S5	101	111	00100000
S6	110	101	01000000
S7	111	100	10000000

Fig. 2.40
Comparison of different state encoding approaches

Once the codes have been assigned to the state names, each of the bits within the code must be given a unique signal name. The signal names are necessary because the individual bits within the state code are going to be implemented with real circuitry, so each signal name will correspond to an actual node in the logic diagram. These individual signal names are called *state variables*. Unique variable names are needed for both the current state and next state signals. The current state variables are driven by the Q outputs of the D-Flip-Flops holding the state codes. The next state variables are driven by the next state logic circuitry and are connected to the D inputs of the D-Flip-Flops. State variable names are commonly chosen that are descriptive both in terms of their purpose and connection location. For example, current state variables are often given the names Q, Q_cur, or Q_current to indicate that they come from the Q outputs of the D-Flip-Flops. Next state variables are given names such as Q*, Q_nxt, or Q_next to indicate that they are the *next* value of Q and are connected to the D input of the D-Flip-Flops. Once state codes and state variable names are assigned, the state transition table is updated with the detailed information.

Returning to our push-button window controller example, let's encode our states in straight binary and use the state variable names of Q_cur and Q_nxt. Example 2.47 shows the process of state encoding and the new state transition table.



EXAMPLE: PUSH-BUTTON WINDOW CONTROLLER FSM – STATE ENCODING

This state machine contains two states, w_closed and w_open. The following are the three possible ways these states could be encoded.

State Name	Binary	Gray Code	One-Hot
w_closed	0	0	01
w_open	1	1	10

Since this machine is so small, there is no difference between the binary and gray code approaches. Both of these techniques will require one D-Flip-Flop to hold the state code. The one-hot approach will require two D-Flip-Flops. Let's choose binary state encoding for this example. Let's use the state variable names Q_cur and Q_nxt.

Once the state codes and state variables are chosen, the state transition table is updated with the new detailed information about the design.

Current State		Input	Next State		Outputs	
	Q_cur	Press		Q_nxt	Open_CW	Close_CCW
w_closed	0	0	w_closed	0	0	0
w_closed	0	1	w_open	1	1	0
w_open	1	0	w_open	1	0	0
w_open	1	1	w_closed	0	0	1

Example 2.47
Push-button window ocntroller – state encoding

2.3.2.6 Next State Logic

The next step in the state machine design is to synthesize the next state logic. The next state logic will compute the values of the next state variables based on the current state and the system inputs. Recall that a combinational logic function drives one and only one output bit. This means that every bit within the next state code needs to have a dedicated combinational logic circuit. The state transition table contains all of the necessary information to synthesize the next state logic including the exact output values of each next state variable for each and every input combination of state code and system input(s).

In our push-button window controller example, we only need to create one combinational logic circuit because there is only one next state variable (Q_nxt). The inputs to the combinational logic circuit are Q_cur and Press. Notice that the state transition table was created such that the order of the input values is listed in a binary count just as in a formal truth table formation. This makes synthesizing the combinational logic circuit straightforward. Example 2.48 shows the steps to synthesize the next state logic for this the push-button window controller.

Example: Push-Button Window Controller FSM – Next State Logic

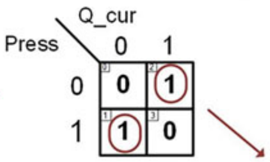
We need to synthesize the combinational logic circuit that will create the next state logic for Q_nxt. The behavior of this combinational logic circuit is described in the state transition table. In order to visualize where this information is within the table, let's pull it out and put it into a traditional truth table format.

Current State		Input	Next State		Outputs	
	Q_cur	Press		Q_nxt	Open_CW	Close_CCW
w_closed	0	0	w_closed	0	0	0
w_closed	0	1	w_open	1	1	0
w_open	1	0	w_open	1	0	0
w_open	1	1	w_closed	0	0	1

These columns are the inputs to the next state logic.

This column is the desired output for the next state logic variable Q_nxt.

Q_cur	Press	Q_nxt
0	0	0
0	1	1
1	0	1
1	1	0



Q_nxt = (Q_cur' · Press) + (Q_cur · Press')

or

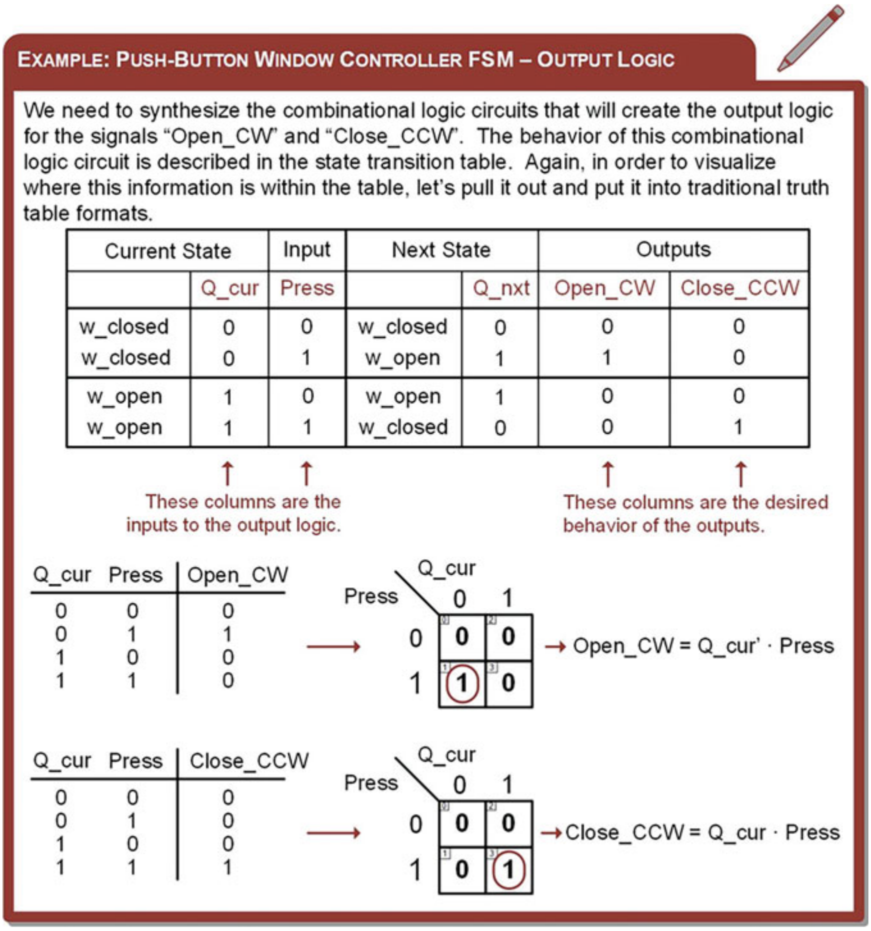
Q_nxt = Q_cur ⊕ Press

Example 2.48
Push-button window controller – next state logic

2.3.2.7 Output Logic

The next step in the state machine design is to synthesize the output logic. The output logic will compute the values of the system outputs based on the current state and, in the case of a Mealy machine, the system inputs. Each of the output signals will require a dedicated combinational logic circuit. Again, the state transition table contains all of the necessary information to synthesize the output logic.

In our push-button window controller example, we need to create one circuit to compute the output "Open_CW" and one circuit to compute the output "Close_CCW." In this example, the inputs to these circuits are the current state (Q_cur) and the system input (Press). Example 2.49 shows the steps to synthesize the output logic for the push-button window controller.



EXAMPLE: PUSH-BUTTON WINDOW CONTROLLER FSM – OUTPUT LOGIC

We need to synthesize the combinational logic circuits that will create the output logic for the signals "Open_CW" and "Close_CCW". The behavior of this combinational logic circuit is described in the state transition table. Again, in order to visualize where this information is within the table, let's pull it out and put it into traditional truth table formats.

Current State		Input	Next State		Outputs	
	Q_cur	Press		Q_nxt	Open_CW	Close_CCW
w_closed	0	0	w_closed	0	0	0
w_closed	0	1	w_open	1	1	0
w_open	1	0	w_open	1	0	0
w_open	1	1	w_closed	0	0	1

These columns are the inputs to the output logic.

These columns are the desired behavior of the outputs.

Q_cur	Press	Open_CW
0	0	0
0	1	1
1	0	0
1	1	0

→

Press \ Q_cur	0	1
0	0	0
1	1	0

→ Open_CW = Q_cur' · Press

Q_cur	Press	Close_CCW
0	0	0
0	1	0
1	0	0
1	1	1

→

Press \ Q_cur	0	1
0	0	0
1	0	1

→ Close_CCW = Q_cur · Press

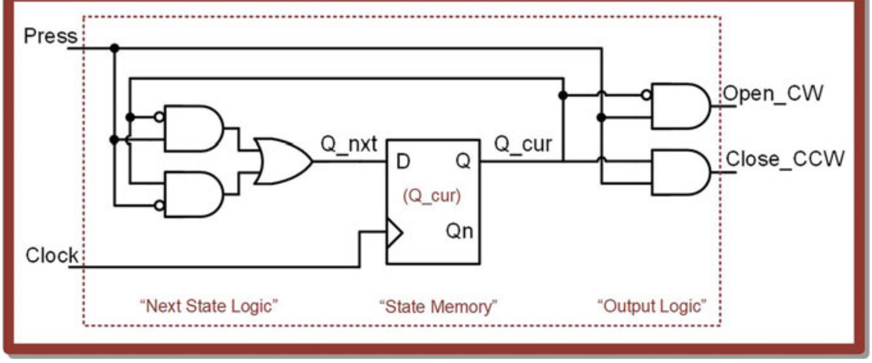
Example 2.49
Push-button window controller – output logic

2.3.2.8 The Final Logic Diagram

The final step in the design of the state machine is to create the logic diagram. It is useful to recall the block diagram for a state machine from Fig. 2.38. A logic diagram begins by entering the state memory. Recall that the state memory consists of D-Flip-Flops that hold the current state code. One D-Flip-Flop is needed for every current state variable. When entering the D-Flip-Flops, it is useful to label them with the current state variable they will be holding. The next part of the logic diagram is the next state logic. Each of the combinational logic circuits that compute the next state variables should be drawn to the left of D-Flip-Flop holding the corresponding current state variable. The output of each next state logic circuit is connected to the D input of the corresponding D-Flip-Flop. Finally, the output logic is entered with the inputs to the logic coming from the current state and potentially from the system inputs.

Example 2.50 shows the process for creating the final logic diagram for our push-button window controller. Notice that the state memory is implemented with one D-Flip-Flop since there is only 1-bit in the current state code (Q_cur). The next state logic is a combinational logic circuit that computes Q_nxt based on the values of Q_cur and Press. Finally, the output logic consists of two separate combinational logic circuits to compute the system outputs Open_CW and Close_CCW based on Q_cur and Press. In this diagram the Qn output of the D-Flip-Flop could have been used for the inverted versions of Q_cur; however, inversion bubbles were used instead in order to make the diagram more readable.

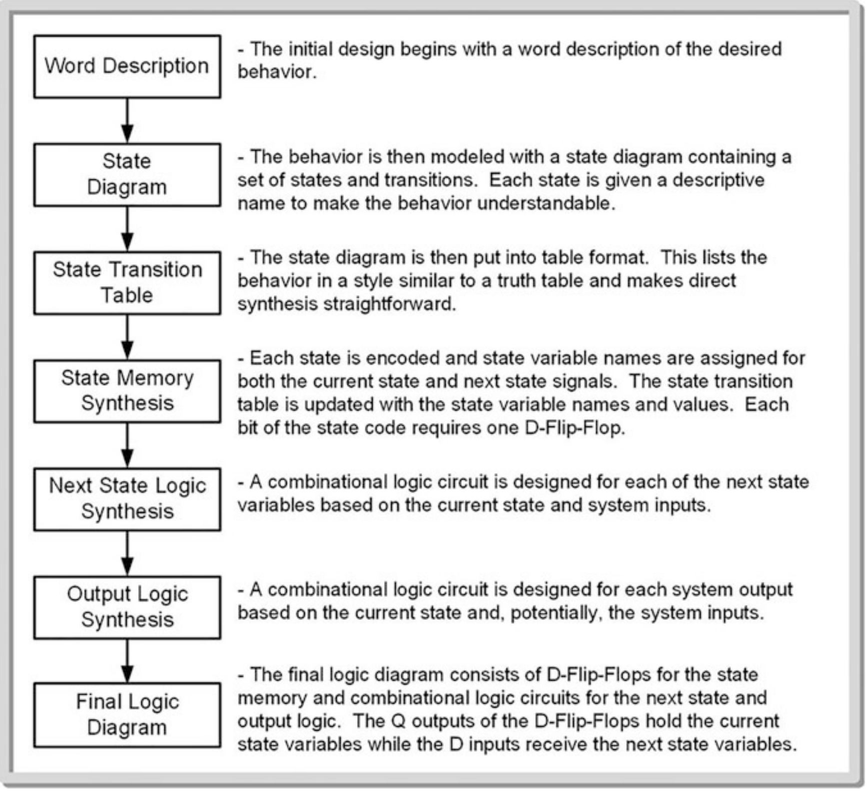
EXAMPLE: PUSH-BUTTON WINDOW CONTROLLER FSM – LOGIC DIAGRAM



Example 2.50
Push-button window controller – logic diagram

2.3.2.9 FSM Design Process Overview

The entire finite state machine design process is given in Fig. 2.41.



Word Description - The initial design begins with a word description of the desired behavior.

State Diagram - The behavior is then modeled with a state diagram containing a set of states and transitions. Each state is given a descriptive name to make the behavior understandable.

State Transition Table - The state diagram is then put into table format. This lists the behavior in a style similar to a truth table and makes direct synthesis straightforward.

State Memory Synthesis - Each state is encoded and state variable names are assigned for both the current state and next state signals. The state transition table is updated with the state variable names and values. Each bit of the state code requires one D-Flip-Flop.

Next State Logic Synthesis - A combinational logic circuit is designed for each of the next state variables based on the current state and system inputs.

Output Logic Synthesis - A combinational logic circuit is designed for each system output based on the current state and, potentially, the system inputs.

Final Logic Diagram - The final logic diagram consists of D-Flip-Flops for the state memory and combinational logic circuits for the next state and output logic. The Q outputs of the D-Flip-Flops hold the current state variables while the D inputs receive the next state variables.

Fig. 2.41
Finite state machine design flow

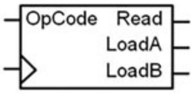
2.3.2.10 FSM Design Example: Simple Control Unit

Let's now look at another FSM design example and follow all of the steps from the prior section from start to finish. In this example, we are going to design a simple control unit representative of the functionality that exists within the central processing unit (CPU) of a computer. Example 2.51 provides the word description, state diagram, and state transition table for this finite state machine.

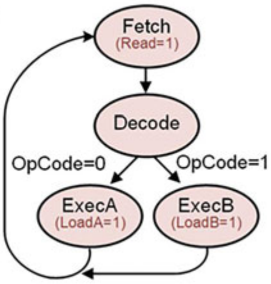
EXAMPLE: FSM DESIGN FOR A SIMPLE CONTROL UNIT – PROBLEM DEFINITION

Word Description - We are going to design a simple control unit for a central processing unit (CPU). A control unit is a FSM that performs the tasks of retrieving instruction codes (aka., the *OpCodes*) out of program memory, decoding them, and then executing the instruction by asserting various control signals to achieve the desired behavior of the instruction.

The simple control unit will retrieve the *OpCode* input from program memory by asserting a signal called *Read*. The FSM will be able to interpret two OpCodes (0 or 1). If the OpCode=0, it will assert a control signal called *LoadA*. If the OpCode=1, it will assert a control signal called *LoadB*. The FSM is synchronous to the rising edge of *Clock*.



State Diagram – To implement this behavior, our FSM will have a Fetch state that will assert the *Read* signal to retrieve the *OpCode* signal from program memory. It will then have a Decode state that will do nothing but provide some time for the OpCode to arrive at the control unit and then decide the next state. The FSM will then go to one of two states depending on the value of the OpCode. If the OpCode=0, it will go to the ExecA state ("Exec" stands for "Execute) and then assert the *LoadA* control signal. If OpCode=1, it will go to the ExecB state and then assert the *LoadB* control signal. After either execution state, it will return to Fetch. The following state diagram models this behavior.

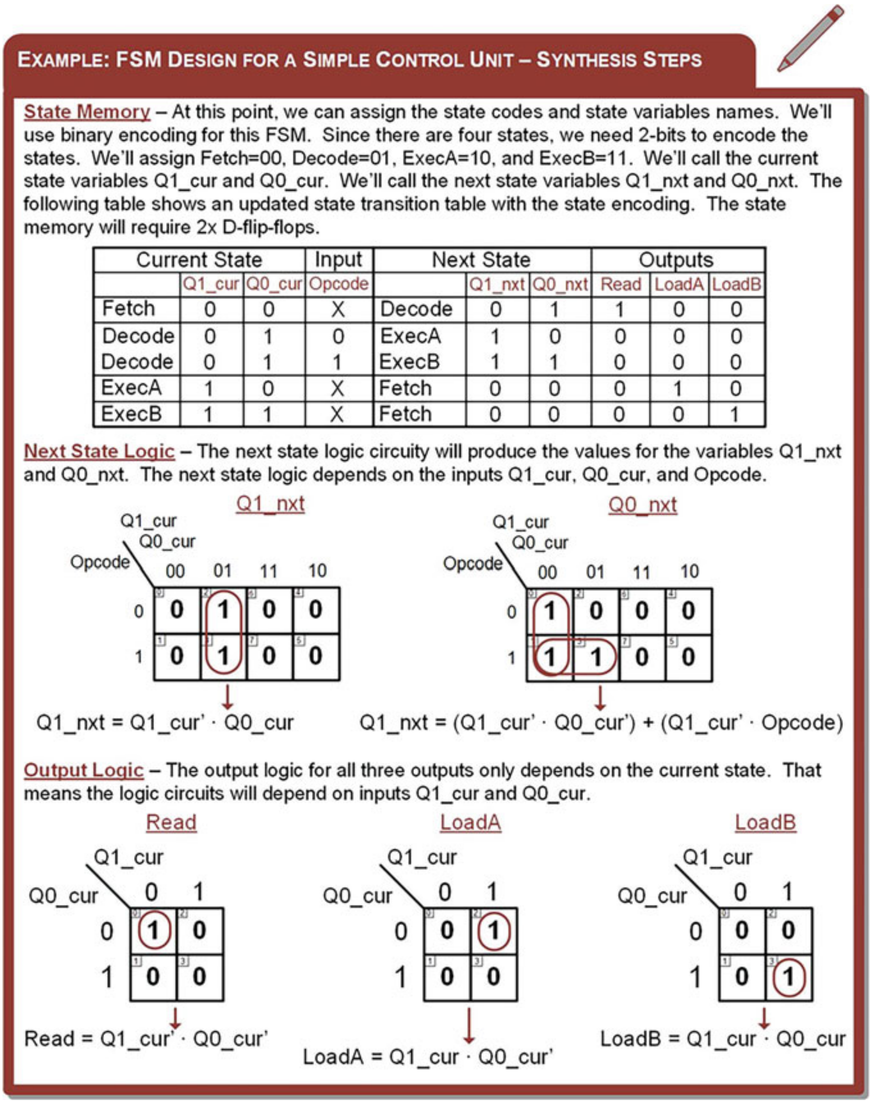


State Transition Table – The following table represents the information in the state diagram in tabular form.

Current State	Input	Next State	Outputs		
	Opcode		Read	LoadA	LoadB
Fetch	X	Decode	1	0	0
Decode	0	ExecA	0	0	0
Decode	1	ExecB	0	0	0
ExecA	X	Fetch	0	1	0
ExecB	X	Fetch	0	0	1

Example 2.51
FSM design for a simple control unit – problem description

Example 2.52 provides the state encoding, next state logic, and output logic synthesis for the simple control unit FSM.



EXAMPLE: FSM DESIGN FOR A SIMPLE CONTROL UNIT – SYNTHESIS STEPS

State Memory – At this point, we can assign the state codes and state variables names. We'll use binary encoding for this FSM. Since there are four states, we need 2-bits to encode the states. We'll assign Fetch=00, Decode=01, ExecA=10, and ExecB=11. We'll call the current state variables Q1_cur and Q0_cur. We'll call the next state variables Q1_nxt and Q0_nxt. The following table shows an updated state transition table with the state encoding. The state memory will require 2x D-flip-flops.

Current State			Input	Next State			Outputs		
	Q1_cur	Q0_cur	Opcode		Q1_nxt	Q0_nxt	Read	LoadA	LoadB
Fetch	0	0	X	Decode	0	1	1	0	0
Decode	0	1	0	ExecA	1	0	0	0	0
Decode	0	1	1	ExecB	1	1	0	0	0
ExecA	1	0	X	Fetch	0	0	0	1	0
ExecB	1	1	X	Fetch	0	0	0	0	1

Next State Logic – The next state logic circuity will produce the values for the variables Q1_nxt and Q0_nxt. The next state logic depends on the inputs Q1_cur, Q0_cur, and Opcode.

Q1_nxt

Opcode \ Q1_cur Q0_cur	00	01	11	10
0	0	1	0	0
1	0	1	0	0

Q1_nxt = Q1_cur' · Q0_cur

Q0_nxt

Opcode \ Q1_cur Q0_cur	00	01	11	10
0	1	0	0	0
1	1	1	0	0

Q1_nxt = (Q1_cur' · Q0_cur') + (Q1_cur' · Opcode)

Output Logic – The output logic for all three outputs only depends on the current state. That means the logic circuits will depend on inputs Q1_cur and Q0_cur.

Read

Q0_cur \ Q1_cur	0	1
0	1	0
1	0	0

Read = Q1_cur' · Q0_cur'

LoadA

Q0_cur \ Q1_cur	0	1
0	0	1
1	0	0

LoadA = Q1_cur · Q0_cur'

LoadB

Q0_cur \ Q1_cur	0	1
0	0	0
1	0	1

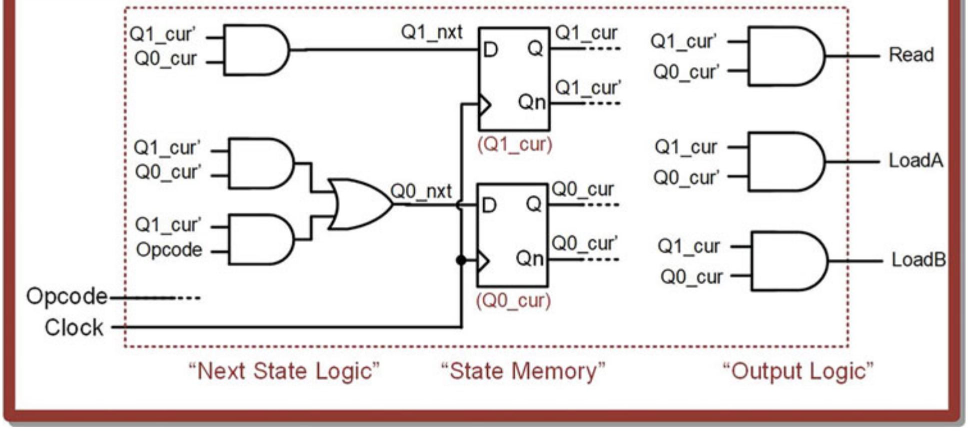
LoadB = Q1_cur · Q0_cur

Example 2.52
FSM design for a simple control unit – synthesis steps

Example 2.53 shows the final logic diagram for the simple control unit FSM.

Example: FSM Design for a Simple Control Unit – Final Logic Diagram

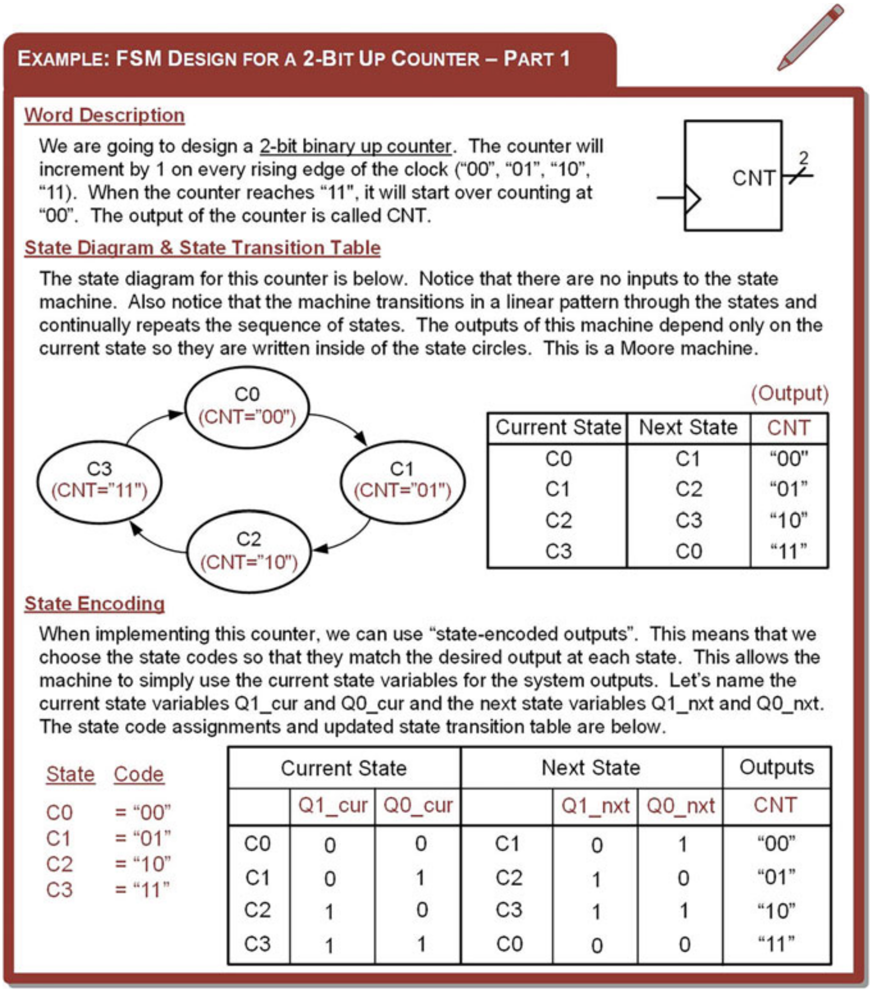
Logic Diagram – The following shows the final logic diagram for the simple control unit.



Example 2.53
FSM design for a simple control unit – final logic diagram

2.3.2.11 FSM Design Example: 2-Bit Up Counter

A *counter* is a special type of finite state machine. A counter will traverse the states within a state diagram in a linear fashion continually circling around all states. This behavior allows a special type of output topology called *state-encoded outputs*. Since each state in the counter represents a unique counter output, the states can be encoded with the associated counter output value. In this way, the current state code of the machine can be used as the output of the entire system. Let's consider the design of a 2-bit binary up counter. The term *up* in this counter means that the counter will simply count in an ascending order and repeat (i.e., 00 → 01 → 10 → 11 → 00. . .). Example 2.54 provides the word description, state diagram, state transition table, and state encoding for this counter.



Example: FSM Design for a 2-Bit Up Counter – Part 1

Word Description

We are going to design a 2-bit binary up counter. The counter will increment by 1 on every rising edge of the clock ("00", "01", "10", "11). When the counter reaches "11", it will start over counting at "00". The output of the counter is called CNT.

State Diagram & State Transition Table

The state diagram for this counter is below. Notice that there are no inputs to the state machine. Also notice that the machine transitions in a linear pattern through the states and continually repeats the sequence of states. The outputs of this machine depend only on the current state so they are written inside of the state circles. This is a Moore machine.

(Output)

Current State	Next State	CNT
C0	C1	"00"
C1	C2	"01"
C2	C3	"10"
C3	C0	"11"

State Encoding

When implementing this counter, we can use "state-encoded outputs". This means that we choose the state codes so that they match the desired output at each state. This allows the machine to simply use the current state variables for the system outputs. Let's name the current state variables Q1_cur and Q0_cur and the next state variables Q1_nxt and Q0_nxt. The state code assignments and updated state transition table are below.

State	Code
C0	= "00"
C1	= "01"
C2	= "10"
C3	= "11"

Current State			Next State			Outputs
	Q1_cur	Q0_cur		Q1_nxt	Q0_nxt	CNT
C0	0	0	C1	0	1	"00"
C1	0	1	C2	1	0	"01"
C2	1	0	C3	1	1	"10"
C3	1	1	C0	0	0	"11"

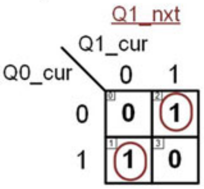
Example 2.54
FSM design for a 2-bit binary up counter (part 1)

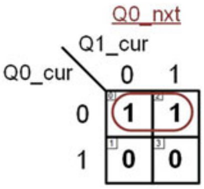
Example 2.55 shows the next state and output logic synthesis, the final logic diagram, and resultant representative timing diagram for the 2-bit binary up counter.

EXAMPLE: FSM DESIGN FOR A 2-BIT UP COUNTER – PART 2

Next State Logic

The next state logic for this counter only depends on the current state variables since there are no inputs to the system.





Q1_nxt = (Q1_cur' · Q0_cur) + (Q1_cur · Q0_cur')

or

Q1_nxt = Q1_cur ⊕ Q0_cur

Q0_nxt = Q0_cur'

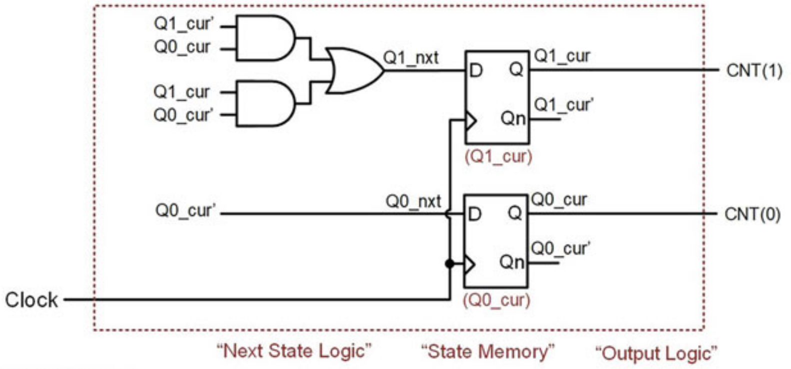
Output Logic

Since we are using state-encoded outputs, the outputs of the system will simply be the current state variables.

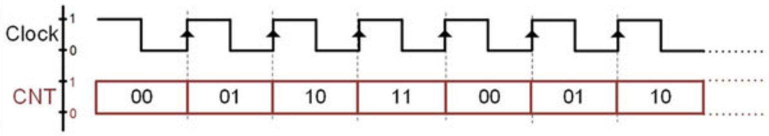
CNT(1) = Q1_cur

CNT(0) = Q0_cur

Logic Diagram



Timing Diagram



Example 2.55
FSM design for a 2-bit binary up counter (part 2)

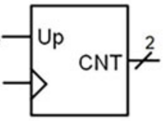
2.3.2.12 FSM Design Example: 2-Bit Binary Up/Down Counter

Let's now consider a 2-bit binary up/down counter. In this type of counter, there is an input that dictates whether the counter increments or decrements. This counter can still be implemented as a Moore machine and use state-encoded outputs. Example 2.56 provides the word description, state diagram, state transition table, and state encoding for this counter.

Example: FSM Design for a 2-Bit Up/Down Counter – Part 1

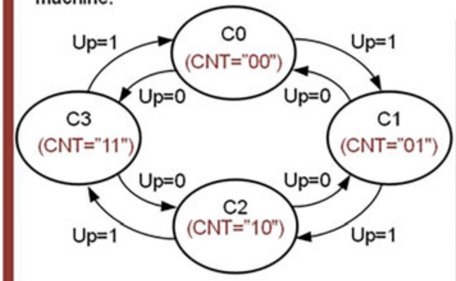
Word Description

We are going to design a 2-bit binary up/down counter. When the system input “Up” is asserted, the counter will increment by 1 on every rising edge of the clock. When Up=0, the counter will decrement by 1 on every rising edge of the clock. The output of the counter is called CNT.



State Diagram & State Transition Table

The state diagram for this counter is below. In this diagram, if the input Up=1, the machine will traverse the states in order to create an incrementing count. If the input Up=0, the machine will traverse the states in the opposite order. The outputs of this machine again only depend on the current state so they are written inside of the state circles. This is a Moore machine.



	(Input)		(Output)
Current State	Up	Next State	CNT
C0	0	C3	“00”
	1	C1	
C1	0	C0	“01”
	1	C2	
C2	0	C1	“10”
	1	C3	
C3	0	C2	“11”
	1	C0	

State Encoding

Again, this counter will use “state-encoded outputs”. Let’s name the current state variables Q1_cur and Q0_cur and the next state variables Q1_nxt and Q0_nxt. The state code assignments and updated state transition table are below.

State	Code
C0	= “00”
C1	= “01”
C2	= “10”
C3	= “11”

Current State			Input	Next State			Outputs
	Q1_cur	Q0_cur	Up		Q1_nxt	Q0_nxt	CNT
C0	0	0	0	C3	1	1	“00”
C0	0	0	1	C1	0	1	“00”
C1	0	1	0	C0	0	0	“01”
C1	0	1	1	C2	1	0	“01”
C2	1	0	0	C1	0	1	“10”
C2	1	0	1	C3	1	1	“10”
C3	1	1	0	C2	1	0	“11”
C3	1	1	1	C0	0	0	“11”

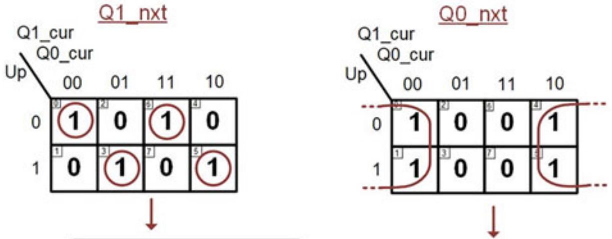
Example 2.56
FSM design for a 2-bit binary up/down counter (part 1)

Example 2.57 shows the next state and output logic synthesis, the final logic diagram, and resultant representative timing diagram for the 2-bit binary up/down counter.

Example: FSM Design for a 2-Bit Up/Down Counter – Part 2

Next State Logic

The next state logic for this counter depends on both the current state variables and the input Up.



Q1_nxt

Up \ Q1_cur Q0_cur	00	01	11	10
0	1	0	1	0
1	0	1	0	1

Q0_nxt

Up \ Q1_cur Q0_cur	00	01	11	10
0	1	0	0	1
1	1	0	0	1

$Q1\_nxt = \overline{Q1\_cur \oplus Q0\_cur \oplus Up}$

$Q0\_nxt = Q0\_cur'$

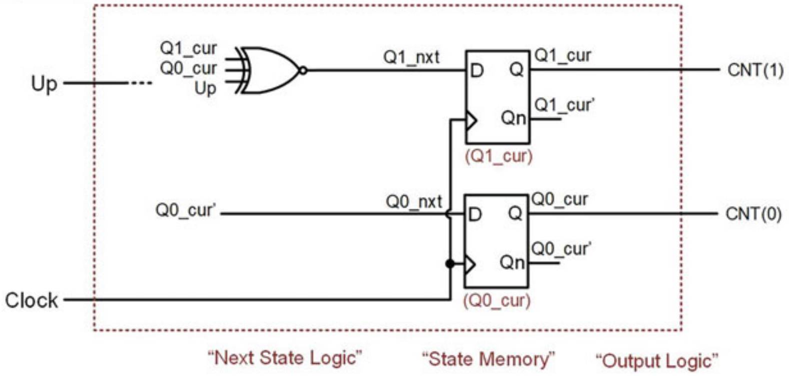
Output Logic

Since we are using state-encoded outputs, the outputs of the system will simply be the current state variables.

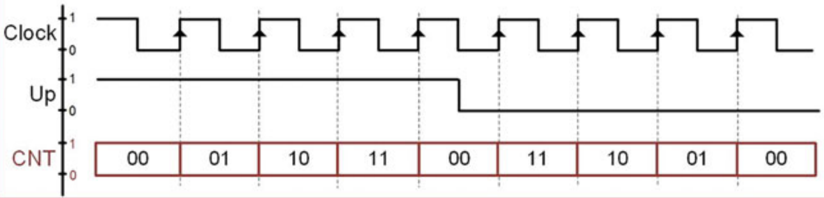
$CNT(1) = Q1\_cur$

$CNT(0) = Q0\_cur$

Logic Diagram



Timing Diagram



Example 2.57
FSM design for a 2-bit binary up/down counter (part 2)

Concept Check

CC2.3.2 What allows a finite state machine to make more intelligent decisions about the system outputs compared to combinational logic alone?

A) A finite state machine has knowledge about the past inputs.

B) The D-flip-flops allow the outputs to be generated more rapidly.

C) The next state and output logic allow the finite state machine to be more complex and implement larger truth tables.

D) A synchronous system is always more intelligent.

2.4 Memory

This chapter introduces the basic concepts and terminology of semiconductor-based memory used in computer systems. The term *memory* is used to describe a system with the ability to store digital information; however, the usage of the word memory in computer systems refers to large, dense arrays of storage. To achieve this increased density, different technologies and architectures are used that differentiate memory storage from storage using registers. The different design approaches used in memory typically result in slower performance and reduced functionality compared to storage using registers but yield the large storage capacity needed to run software in modern computers.

The term *semiconductor memory* refers to systems that are implemented using integrated circuit technology. These types of systems store the digital information using transistors, fuses, magnet materials, and/or capacitors on a single semiconductor substrate. Memory can also be implemented using technology other than semiconductors. Semiconductor memory does not have any moving parts, so it is also referred to as *solid state memory.* Regardless of the technology used to store the binary data, all memory has common attributes and terminology that are discussed in this section.

2.4.1 Memory Terminology

The information stored in memory is called *data*. When data is placed into memory, it is called a *write*. When information is retrieved from memory, it is called a *read*. In order to access data in memory, an *address* is used. While data can be accessed as individual bits, in order to reduce the number of address locations needed, data is typically grouped into *N-bit words*. If a memory system has $N = 8$, this means that 8 bits of data are stored at each address. The number of address locations is described using the variable M. The overall size of the memory is called its *capacity* and can be stated by saying $M \times N$. For example, if we had a 16×8 memory system, that means that there are 16 address locations, each capable of storing 8 bits of data. This memory would have a capacity of $16 \times 8 = 128$ bits. We could also say that this memory has a capacity of 16 bytes. Since the address is implemented as a binary code, the number of lines in the address bus (n) will dictate the number of address locations that the memory system will have ($M = 2^n$). Figure 2.42 shows a graphical depiction of how data resides in memory. This type of graphic is called a *memory map model*.

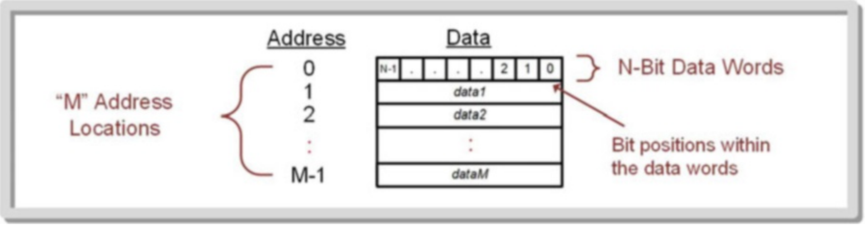


Fig. 2.42
Memory map model

Memory is classified into two categories depending on whether it can store information when power is removed or not. The term *non-volatile* is used to describe memory that *holds* information when the power is removed, while the term *volatile* is used to describe memory that loses its information when power is removed. Historically, volatile memory is able to run at faster speeds compared to non-volatile memory, so it is used as the primary storage mechanism while a digital system is running. Non-volatile memory is necessary in order to hold critical operation information for a digital system such as startup instructions, operation systems, and applications.

Memory can also be classified into two categories with respect to how data is accessed. *Read Only Memory (ROM)* is a device that cannot be written to during normal operation. This type of memory is useful for holding critical system information or programs that should not be altered while the system is running. *Read/write* (R/W) memory refers to memory that can be read and written to during normal operation and is used to hold temporary data and variables.

Random Access Memory (RAM) describes memory in which any location in the system can be accessed at any time. The opposite of this is *sequential access* memory, in which not all address locations are immediately available. An example of a sequential access memory system is a tape drive. In order to access the desired address in this system, the tape spool must be spun until the address is in a position that can be observed. Most semiconductor memory in modern systems is random access. The terms RAM and ROM have been adopted, somewhat inaccurately, to also describe groups of memory with particular behavior. While the term ROM technically describes a system that cannot be written to, it has taken on the additional association of being the term to describe non-volatile memory. While the term RAM technically describes how data is accessed, it has taken on the additional association of being the term to describe volatile memory. When describing modern memory systems, the terms RAM and ROM are used most commonly to describe the characteristics of the memory being used; however, modern memory systems can be both read/write and non-volatile, and the majority of memory is random access.

Concept Check

CC2.4.1 An 8-bit wide memory has 8 address lines. What is its capacity in bits?

A) 64

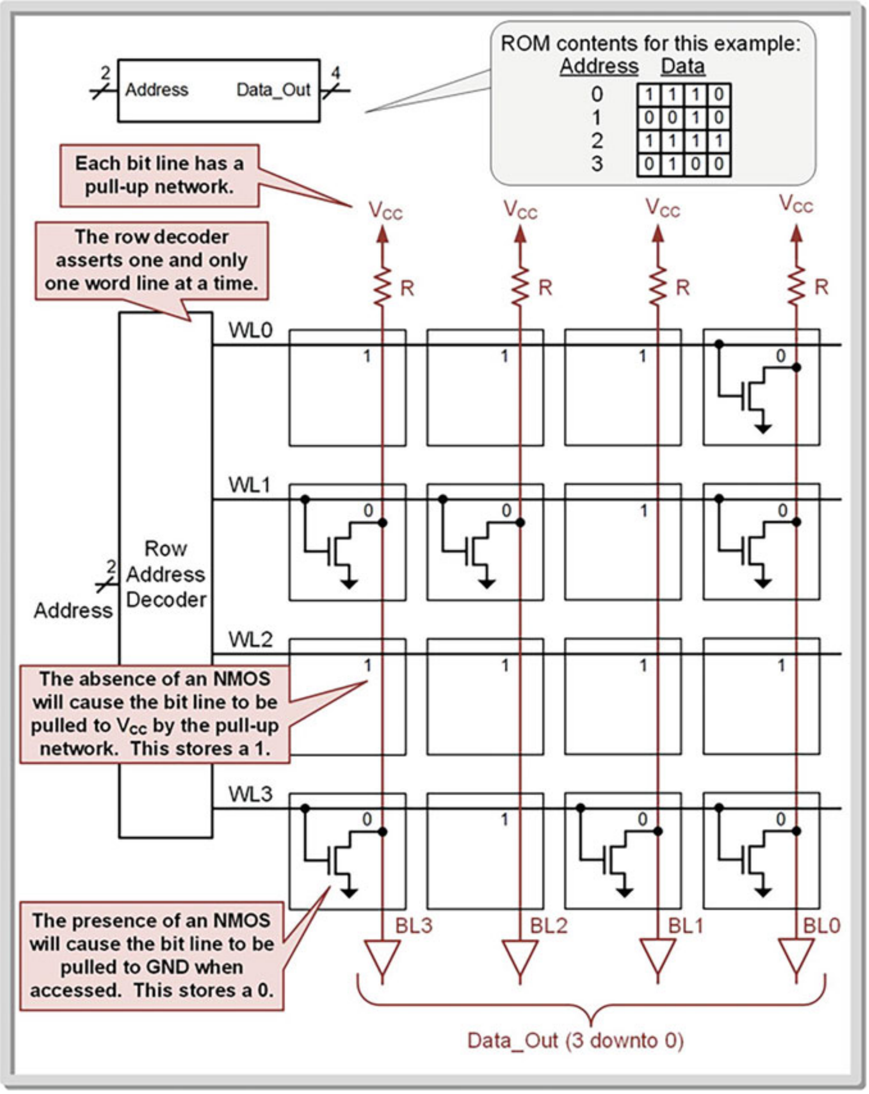
B) 256

C) 1024

D) 204

2.4.2 Memory Architecture

This section describes the basic architecture of a memory array. To begin, we will look at a simple non-volatile ROM array. Figure 2.43 shows the basic architecture of a 4 × 4 ROM. An address decoder is used to access individual data words within the memory system. The address decoder asserts one and only one *word line* (WL) for each unique binary address that is present on its input. This operation is identical to a binary-to-one-hot decoder. For an *n*-bit address, the decoder can access 2^n, or M words in memory. The word lines historically run horizontally across the memory array; thus, they are often called *row lines* and the word line decoder is often called the *row decoder*. *Bit lines* (BL) run perpendicular to the word lines in order to provide individual bit storage access at the intersection of the bit and word lines. These lines typically run vertically through the memory array; thus, they are often called *column lines*. The output of the memory system (i.e., Data_Out) is obtained by providing an address and then reading the word from buffered versions of the bit lines. When a system provides individual bit access to a row, or access to multiple data words sharing a row line, a column decoder is used to route the appropriate bit line(s) to the data out port.



ROM contents for this example:

Address	Data
0	1 1 1 0
1	0 0 1 0
2	1 1 1 1
3	0 1 0 0

Each bit line has a pull-up network.

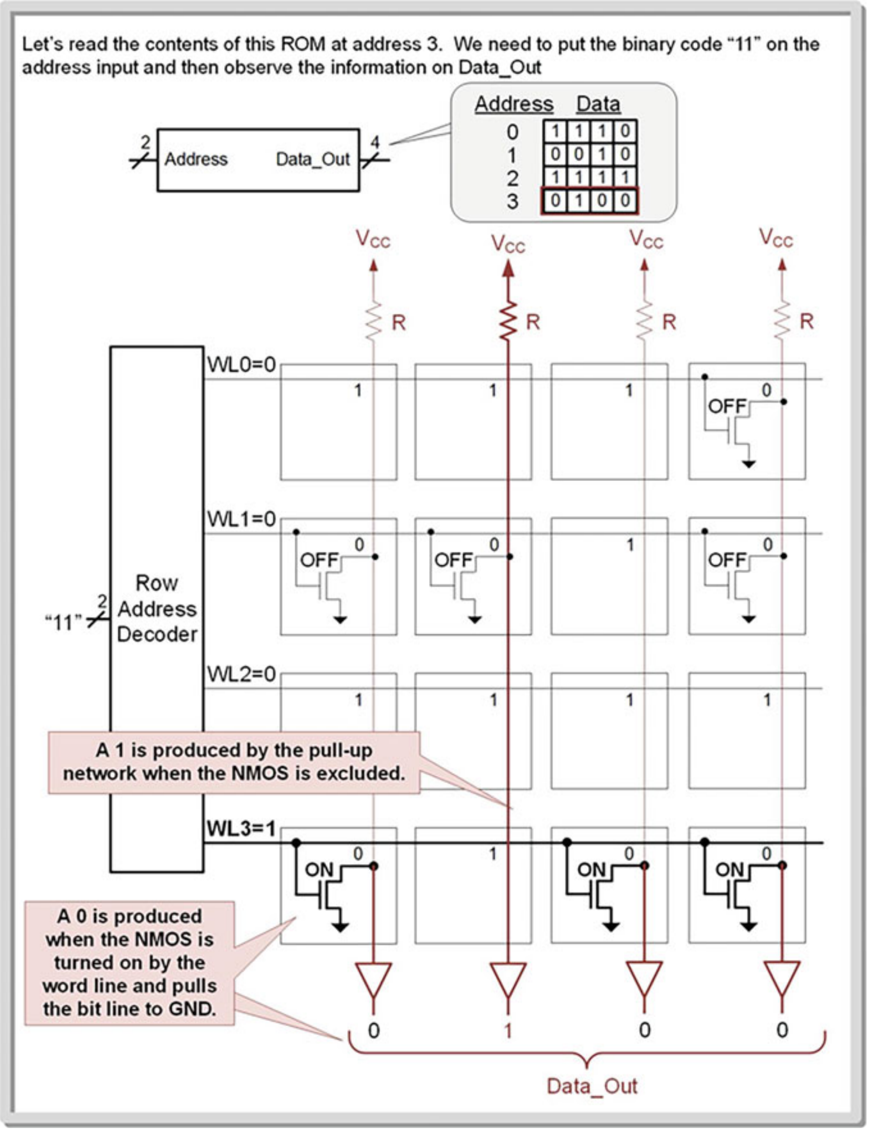
The row decoder asserts one and only one word line at a time.

The absence of an NMOS will cause the bit line to be pulled to V_{CC} by the pull-up network. This stores a 1.

The presence of an NMOS will cause the bit line to be pulled to GND when accessed. This stores a 0.

Fig. 2.43
Basic architecture of Read Only Memory (ROM)

In a traditional ROM array, each bit line contains a pull-up network to V_{CC}. This provides the ability to store a logic 1 at all locations within the array. If a logic 0 is desired at a particular location, an NMOS pull-down transistor is inserted. The gate of the NMOS is connected to the appropriate word line and the drain of the NMOS is connected to the bit line. When reading, the word line is asserted and turns on the NMOS transistor. This pulls the bit line to GND and produces a logic 0 on the output. When the NMOS transistor is excluded, the bit line remains at a logic 1 due to the pull-up network. Figure 2.44 shows the operation of a ROM when information is being read from address "11."



Let's read the contents of this ROM at address 3. We need to put the binary code "11" on the address input and then observe the information on Data_Out

Address	Data			
0	1	1	1	0
1	0	0	1	0
2	1	1	1	1
3	0	1	0	0

A 1 is produced by the pull-up network when the NMOS is excluded.

A 0 is produced when the NMOS is turned on by the word line and pulls the bit line to GND.

Fig. 2.44
ROM operation during a read

Memory can be designed to be either asynchronous or synchronous. *Asynchronous memory* updates its data outputs immediately upon receiving an address. *Synchronous memory* only updates its data outputs on the rising edge of a clock. The term *latency* is used to describe the delay between when a signal is sent to the memory (either the address in an asynchronous system or the clock in a synchronous system) and when the data is available. Figure 2.45 shows a comparison of the timing diagrams between asynchronous and synchronous ROM systems during a read cycle.

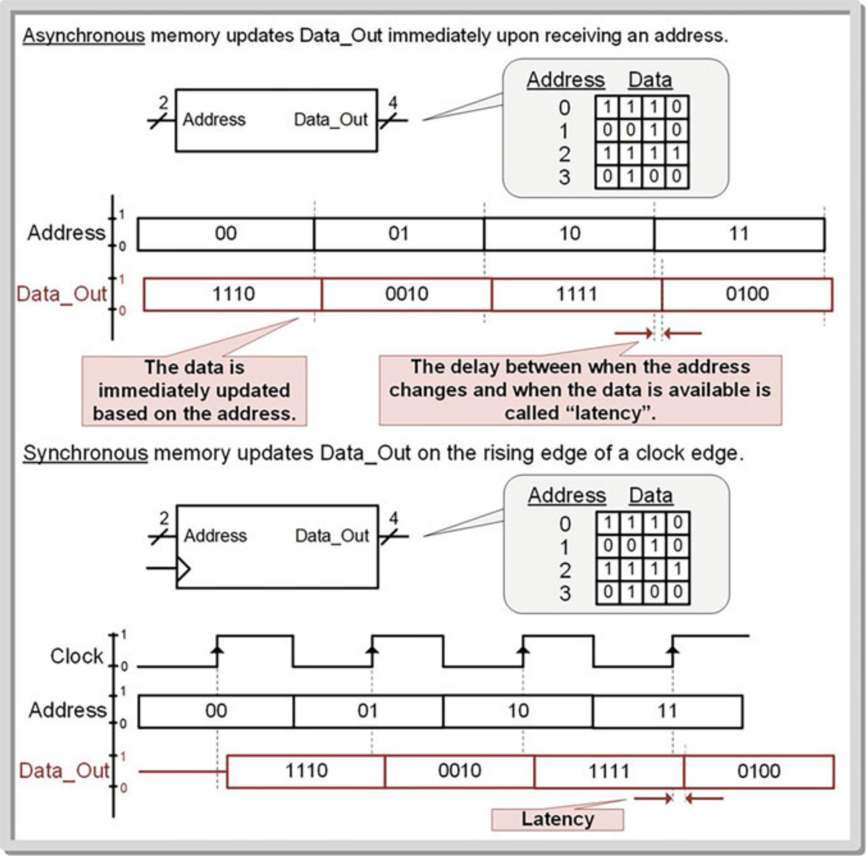

Asynchronous memory updates Data_Out immediately upon receiving an address.
2
Address
Data_Out
4
Address
Data
0
1
2
3
1 1 1 0
0 0 1 0
1 1 1 1
0 1 0 0
Address
Data_Out
1
0
00
01
10
11
1110
0010
1111
0100
The data is immediately updated based on the address.
The delay between when the address changes and when the data is available is called "latency".
Synchronous memory updates Data_Out on the rising edge of a clock edge.
Clock
Latency

Fig. 2.45
Asynchronous versus synchronous ROM operation during a read cycle

A R/W memory array is implemented in a similar manner as a ROM array; however, the storage cells are implemented with technology that can be written to and read from during normal operation. This means that circuitry must exist to be able to both retrieve information from the cells and drive data into the cells. The most common storage cells in R/W memory require the data to be read/written using *differential signals*. A differential signal is one that uses two wires to transmit the data, one with the value of the data and the other with the complement. A *differential line driver* with enable is used to create the differential signal driven into the storage cell. A *differential amplifier* determines the value stored in the cell during a read by taking the difference between the two lines. This allows the intended value to be determined using lower voltage levels and also in the presence of increased noise. Figure 2.46 shows the basic architecture of the R/W array.

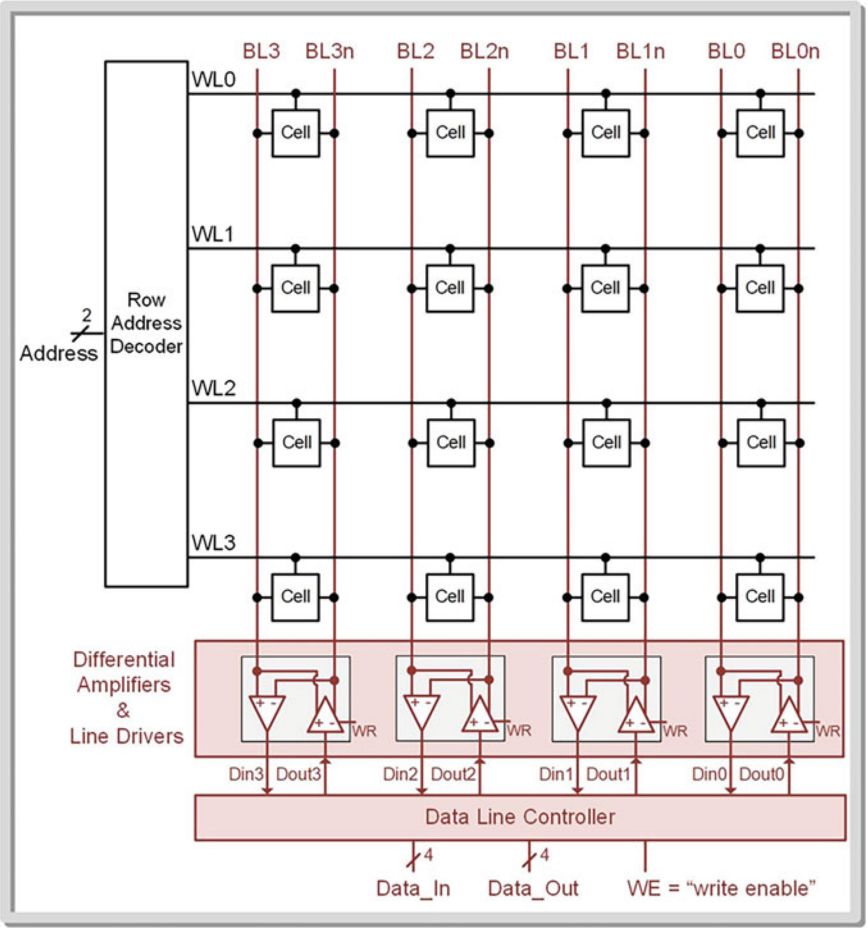


Fig. 2.46
Basic architecture of read/write memory (R/W)

Concept Check

CC2.4.2(a) Which of the following is suitable for implementation in a read only memory?

A) Variables that a computer program needs to continuously update.

B) Information captured by a digital camera.

C) A computer program on a spacecraft.

D) Incoming digitized sound from a microphone.

CC2.4.2(b) Which of the following is suitable for implementation in a read/write memory?

A) A look up table containing the values of sine.

B) Information captured by a digital camera.

C) The boot up code for a computer.

D) A computer program on a spacecraft.

2.4.3 Memory Technologies

The primary difference between the semiconductor memory types used in computer systems is the approach or technology used to create the storage cells. The storage cell technology dictates whether the cells are volatile vs. non-volatile, ROM vs. R/W, the speed data can be accessed, and the power consumed by the array.

2.4.3.1 Masked Read Only Memory (MROM)

A Masked Read Only Memory (MROM) is a non-volatile device that is programmed during fabrication. The term *mask* refers to a transparent plate that contains patterns to create the features of the devices on an integrated circuit using a process called photolithography. An MROM is fabricated with all of the features necessary for the memory device with the exception of the final connections between the NMOS transistors and the word and bit lines. This allows the majority of the device to be created prior to knowing what the final information to be stored is. Once the desired information to be stored is provided by the customer, the fabrication process is completed by adding connections between certain NMOS transistors and the word/bit lines in order to create logic 0s.

2.4.3.2 Programmable Read Only Memory (PROM)

A Programmable Read Only Memory (PROM) is created in a similar manner as an MROM except that the programming is accomplished post-fabrication through the use of fuses or anti-fuses. A *fuse* is an electrical connection that is normally conductive. When a certain amount of current is passed through the fuse, it will melt and create an open circuit. The amount of current necessary to open the fuse is much larger than the current the fuse would conduct during normal operation. An *anti-fuse* operates in the opposite manner as a fuse. An anti-fuse is normally an open circuit. When a certain amount of current is forced into the anti-fuse, the insulating material breaks down and creates a conduction path. This turns the anti-fuse from an open circuit into a wire. Again, the amount of current necessary to close the anti-fuse is much larger than the current the anti-fuse would experience during normal operation. A PROM uses fuses or anti-fuses in order to connect/disconnect the NMOS transistors in the ROM array to the word/bit lines. A PROM programmer is used to *burn* the fuses or anti-fuses. A PROM can only be programmed once in this manner; thus, it is a read only memory and non-volatile. A PROM has the advantage that programming can take place quickly as opposed to an MROM that must be programmed through device fabrication.

2.4.3.3 Erasable Programmable Read Only Memory (EPROM)

As an improvement to the one-time programming characteristic of PROMs, an electrically programmable ROM with the ability to be erased with ultraviolet (UV) light was created. The Erasable Programmable Read Only Memory (EPROM) is based on a *floating-gate transistor*. In a floating-gate transistor, an additional metal-oxide structure is added to the gate of an NMOS. This has the effect of increasing the threshold voltage. The geometry of the second metal oxide is designed such that the threshold voltage is high enough that normal CMOS logic levels are not able to turn the transistor on (i.e., $V_{T1} \geq V_{CC}$). This threshold can be changed by applying a large electric field across the two metal structures in the gate. This causes charge to tunnel into the secondary oxide, ultimately changing it into a conductor. This phenomenon is called *Fowler–Nordheim tunneling*. The new threshold voltage is low enough that normal CMOS logic levels are not able to turn the transistors off (i.e., $V_{T2} \leq$ GND). This process is how the device is *programmed*. This process is accomplished using a dedicated programmer; thus, the EPROM must be removed from its system to program. In order to change the floating-gate transistor back into its normal state, the device is exposed to a strong ultraviolet light source. When the UV light strikes the trapped charge in the secondary oxide, it transfers enough energy to the charge particles that they can move back into the metal plates in the gate. This, in effect, erases the device and restores it back to a state with a high threshold voltage. EPROMs contain a transparent window on the top of their package that allows the UV light to strike the devices. The EPROM must be removed from its system to perform the erase procedure. When the UV light erase procedure is performed, every device in the memory array is erased. EPROMs are a significant improvement over PROMs because they can be programmed multiple times; however, the programming and erase procedures are manually intensive and require an external programmer and external eraser.

2.4.3.4 Electrically Erasable Programmable Read Only Memory (EEPROM)

In order to address the inconvenient programming and erasing procedures associated with EPROMs, the Electrically Erasable Programmable ROM (EEPROM) was created. In this type of circuit, the floating-gate transistor is erased by applying a large electric field across the secondary oxide. This electric field provides the energy to move the trapped charge from the secondary oxide back into the metal plates of the gate. The advantage of this approach is that the circuitry to provide the large electric field can be generated using circuitry on the same substrate as the memory array, thus eliminating the need for an external UV light eraser. In addition, since the circuitry exists to generate large on-chip voltages, the device can also be programmed without the need for an external programmer. This allows an EEPROM to be programmed and erased while it resides in its target environment. Early EEPROMs were very slow and had a limited number of program/erase cycles; thus, they were classified into the category of non-volatile, Read Only Memory. Modern floating-gate transistors are now capable of access times on scale with other volatile memory systems; thus, they have evolved into one of the few non-volatile, read/write memory technologies used in computer systems today. Almost all modern microcontrollers, including the MSP430, include on-chip EEPROM to hold their programs.

2.4.3.5 FLASH Memory

One of the early drawbacks of EEPROM was that the circuitry that provided the capability to program and erase individual bits also added to the size of each individual storage element. FLASH EEPROM was a technology that attempted to improve the density of floating-gate memory by programming and erasing in large groups of data, known as *blocks*. This allowed the individual storage cells to shrink and provided higher-density memory parts. This new architecture was called *NAND FLASH* and provided faster write and erase times coupled with higher-density storage elements. The limitation of NAND FLASH was that reading and writing could only be accomplished in a block-by-block basis. This characteristic precluded the use of NAND FLASH for run-time variables and data storage but was well suited for streaming applications such as audio/video and program loading. As NAND FLASH technology advanced, the block size began to shrink, and software adapted to accommodate the block-by-block data access. This expanded the applications that NAND FLASH could be deployed in. Today, NAND FLASH memory is used in nearly all portable devices (e.g., smart phones and tablets), and its use in solid-state hard drives is on pace to replace hard disk drives and optical disks as the primary non-volatile storage medium in modern computers.

In order to provide individual word access, NOR FLASH was introduced. In NOR FLASH, circuitry is added to provide individual access to data words. This architecture provided faster read times than NAND FLASH, but the additional circuitry causes the write and erase times to be slower and the individual storage cell size to be larger. Due to NAND FLASH having faster write times and higher density, it is seeing broader scale adoption compared to NOR FLASH despite only being able to access information in blocks. NOR FLASH is considered random access memory while NAND FLASH is typically not; however, as the block size of NAND FLASH is continually reduced, its use for variable storage is becoming more attractive. All FLASH memory is non-volatile and read/write.

2.4.3.6 Static Random Access Memory (SRAM)

Static Random Access Memory (SRAM) is a semiconductor technology that stores information using a cross-coupled inverter feedback loop. Figure 2.47 shows the schematic for the basic SRAM storage cell. In this configuration, two access transistors (M1 and M2) are used to read and write from the storage cell. The cell has two complementary ports called bit line (BL) and bit line' (BLn). Due to the inverting functionality of the feedback loop, these two ports will always be the complement of each other. This behavior is advantageous because the two lines can be compared to each other to determine the data value. This allows the voltage levels used in the cell to be lowered while still being able to detect the stored data value. Word lines are used to control the access transistors. This storage element takes six CMOS transistors to implement and is often called a 6T configuration. The advantage of this memory cell is that it has very fast performance compared to other sub-systems because of its underlying technology being simple CMOS transistors. SRAM is volatile memory because when the power is removed, the cross-coupled inverters are not able to drive the feedback loop and the data is lost. SRAM is also read/write memory because the storage cells can be easily read from or written to during normal operation. SRAM cells are commonly implemented on the same IC substrate as the rest of the system, thus allowing a fully integrated system to be realized. SRAM cells are used for cache memory in computer systems.

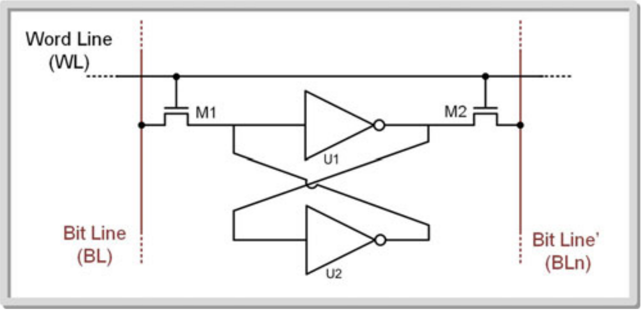


Fig. 2.47
SRAM storage cell

2.4.3.7 Dynamic Random Access Memory (DRAM)

Dynamic Random Access Memory (DRAM) is a semiconductor technology that stores information using a capacitor. A capacitor is a fundamental electrical device that stores charge. Figure 2.48 shows the schematic for the basic DRAM storage cell. The capacitor is accessed through a transistor (M1). Since this storage element takes one transistor and one capacitor, it is often referred to as a 1T1C configuration. Just as in SRAM memory, word lines are used to access the storage elements. The term *digit line* is used to describe the vertical connection to the storage cells.

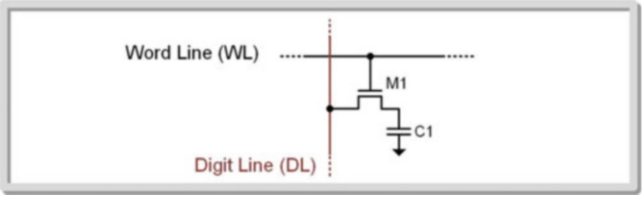


Fig. 2.48
DRAM storage cell

DRAM has an advantage over SRAM in that the storage element requires less area to implement. This allows DRAM memory to have much higher density compared to SRAM; however, there are a variety of architecture issues that cause DRAM to be slower than SRAM. First, the storage capacitors are non-ideal so they continually leak charge and ultimately lose the digital value they are storing. As a result, DRAM cells need to be continually *refreshed* (i.e., writing the same values to the array), in order to maintain their integrity. Next, when the storage capacitor is storing a one, or V_{CC}, the threshold voltage needed to turn on the access transistor needs to be higher than V_{CC} (i.e., $V_{CC} + V_T$). This means that a voltage is required for the word line that is higher than the power supply. A *charge pump* system is used in DRAMs in order to build up a voltage higher than $\mathrm{V_{CC}}$ to access the storage cells. This pumping takes additional time, which slows down the performance of the array. Finally, since the storage cell is a capacitor and not an active device, when reading from the cell it does not have the ability to drive the digit line to a full V_{CC} or GND. Instead, the charge in the capacitor is distributed across capacitance of the digit line. This *charge sharing* reduces the actual voltage that develops on the digit line. This necessitates amplifiers to be used in order to boost voltage on the digit line so that it can be read by a digital receiver. All of these considerations makes DRAM slightly slower and more complex than SRAM. Modern DRAM contain all of the necessary circuitry within the same package as the storage array so much of this complexity is abstracted from the user.

2.4.3.8 Ferroelectric Random Access Memory (FRAM)

One of the newest memory technologies within the semiconductor industry is Ferroelectric Random Access Memory (FRAM). FRAM is very similar to DRAM in its architecture; however, instead of using a traditional capacitor as its storage element, it uses a *ferroelectric capacitor*. In a traditional capacitor, two metal plates are separated by a *dielectric* material. When an external electric field is applied to the capacitor, the charge in the dielectric material is *polarized* or aligns to the direction of the electric field. Once the external electric field is removed, the material depolarizes back to its normal state. In a ferroelectric capacitor, the dielectric is replaced with a ferroelectric material. The ferroelectric material has magnetic properties that give it *hysteresis*, or the characteristic that once the charge in the material is polarized by an external electric field, the material remains polarized after the electric field is removed. This gives the capacitor the ability to be both read/write and non-volatile. FRAM has the advantage that it achieves the higher density of DRAM, but is non-volatile; however, current FRAM technology tends to be slower than DRAM. Figure 2.49 shows the schematic for the basic FRAM storage cell.

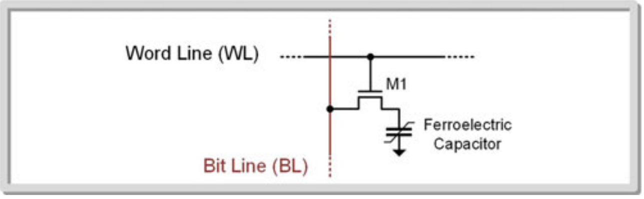


Fig. 2.49
FRAM storage cell

Concept Check

CC2.4.3 Memory arrays tend to follow a very similar architecture. What is the primary difference between the memory technologies?

A) The approached used to implement the storage cell.

B) Where they are manufactured.

C) The number of engineers needed to design them.

D) The number of companies making them.

Summary

- The base, or radix, of a number system refers to the number of unique symbols within its set. The definition of a number system includes both the symbols used and the relative values of each symbol within the set.
- A positional number system allows larger (or smaller) numbers to be represented beyond the values within the original symbol set. This is accomplished by having each position within a number have a different *weight*.
- There are specific algorithms that are used to convert any base to or from decimal. There are also algorithms to convert between number systems that contain a power-of-two symbols (e.g., binary to hexadecimal and hexadecimal to binary).
- Binary arithmetic is performed on a fixed width of bits (*n*). When an *n*-bit addition results in a sum that cannot fit within *n*-bits, it generates a *carryout* bit. In an *n*-bit subtraction, if the minuend is smaller than the subtrahend, a *borrow in* can be used to complete the operation.
- Binary codes can represent both unsigned and signed numbers. For an arbitrary *n*-bit binary code, it is important to know the encoding technique and the range of values that can be represented.
- Signed numbers use the most significant position to represent whether the number is negative (0 = positive, 1 = negative). The width of a signed number is always fixed.
- Two's complement is the most common encoding technique for signed numbers. It has an advantage that there are no duplicate codes for 0 and that the encoding approach provides a monotonic progression of codes from the most negative number that can be represented to the most positive. This allows addition and subtraction to work the same on two's complement numbers as it does on unsigned numbers.
- Logic *gates* represent the most basic operations that can be performed on binary numbers. They are BUF, INV, AND, NAND, OR, NOR, XOR, and XNOR.
- Boolean algebra defines the axioms and theorems that guide the operations that can be performed on a two-valued number system.
- The *canonical form* of a logic expression is one that has not been minimized.
- A *canonical sum of products* form is a logic synthesis technique based on *minterms*. A minterm is a product term that will output a *one* for only one unique input code. A minterm is used for each row of a truth table corresponding to an output of a *one*. Each of the minterms is then summed together to create the final system output.
- A *minterm list* is a shorthand way of describing the information in a truth table. The symbol "Σ" is used to denote a minterm list. Each of the input variables is added to this symbol as comma-delimited subscripts. The row number is then listed for each row corresponding to an output of a *one*.
- A *canonical product of sums* form is a logic synthesis technique based on *maxterms*. A maxterm is a sum term that will output a *0* for only one unique input code. A maxterm is

used for each row of a truth table corresponding to an output of a *0.* Each of the maxterms is then multiplied together to create the final system output.

- A *maxterm list* is a shorthand way of describing the information in a truth table. The symbol "Π" is used to denote a maxterm list. Each of the input variables is added to this symbol as comma-delimited subscripts. The row number is then listed for each row corresponding to an output of a *0*.
- A *Karnaugh map* (K-map) is a graphical approach to minimizing logic expressions. A K-map arranges a truth table into a grid in which the neighboring cells have input codes that differ by only one bit. This allows the impact of an input variable on a group of outputs to be quickly identified.
- A *don't care* (X) can be used when the output of a truth table row can be either a 0 or a 1 without affecting the system behavior. This typically occurs when some of the input codes of a truth table will never occur. The value for the row of a truth table containing a don't care output can be chosen to give the most minimal logic expression. In a K-map, don't cares can be included to form the largest groupings in order to give the least amount of logic.
- While exclusive-OR gates are not used in Boolean algebra, they can be visually identified in K-maps by looking for checkerboard patterns.
- The term medium-scale integrated circuit (MSI) logic refers to a set of basic combinational logic circuits that implement simple, commonly used functions such as decoders, encoders, multiplexers, and demultiplexers. MSI logic can also include operations such as comparators and simple arithmetic circuits.
- Sequential logic refers to a circuit that bases its outputs on both the present and past values of the inputs. Past values are held in sequential logic storage device.
- All sequential logic storage devices are based on a cross-coupled feedback loop. The positive feedback loop formed in this configuration will hold either a 1 or a 0. This is known as a bistable device.
- A D-Flip-Flop will update its Q output with the value on its D input on every triggering edge of a clock. The amount of time that it takes for the Q output to update after a triggering clock edge is called the "t-clock-to-Q" (t_{CQ}) specification.
- A register is a group of D-Flip-Flops that all share the same clock and store groups of data lines called a *bus*.
- A synchronous system is one in which all logic transitions occur based on a single timing event. The timing event is typically the triggering edge of a clock.
- A finite state machine (FSM) is a system that produces outputs based on the current value of the inputs and a history of past inputs. The history of inputs is recorded as *states* that the machine has been in. As the machine responds to new inputs, it transitions between states. This allows a finite state machine to make more sophisticated decisions about what outputs to produce by knowing its history.
- A counter is a special type of finite state machine in which the states are traversed linearly. The linear progression of states allows the next state logic to be simplified. The complexity of the output logic in a counter can also be reduced by encoding the states with the desired counter output for that state. This technique, known as *state-encoded outputs,* allows the system outputs to simply be the current state of the FSM.
- The term memory refers to large arrays of digital storage. The technology used in memory is typically optimized for storage density at the expense of control capability. This is different from a D-Flip-Flop, which is optimized for complete control at the bit level.
- A memory device always contains an address bus input. The number of bits in the address bus dictates how many storage locations can be accessed. An *n*-bit address bus can access 2^n (or M) storage locations.
- The width of each storage location (N) allows the density of the memory array to be increased by reading and writing vectors of data instead of individual bits.
- A memory map is a graphical depiction of a memory array. A memory map is useful to give an overview of the capacity of the array and how different address ranges of the array are used.
- Volatile memory will lose its data when the power is removed. Non-volatile memory will retain its data when the power is removed.
- Read Only Memory (ROM) is a memory type that cannot be written to during normal operation. Read/Write (R/W) memory is a memory type that can be written to during normal operation. Both ROM and R/W memory can be read from during normal operation.

- Random Access Memory (RAM) is a memory type in which any location in memory can be accessed at any time. In Sequential Access Memory the data can only be retrieved in a linear sequence. This means that in sequential memory the data cannot be accessed arbitrarily.
- Memory technologies differ in the way that they store the digital values. The technology used dictates whether the memory is R/W or ROM, volatile or non-volatile, and its performance.

Exercise Problems

Section 2.1: Number Systems

2.1.1 For the number 261.367, what position (p) is the number 2 in?

2.1.2 For the number 261.367, what position (p) is the number 3 in?

2.1.3 What is the name of the number system containing 10_2?

2.1.4 What is the name of the number system containing 10_{16}?

2.1.5 Which of the three number systems covered in this chapter (i.e., binary, decimal, and hexadecimal) could the number 22 be part of? Give all that are possible.

2.1.6 If the number 101.111 has a radix of 2, what is the weight of the position containing the leftmost 1?

2.1.7 If the number 101.111 has a radix of 2, what is the weight of the position containing the rightmost 1?

2.1.8 Convert $11\ 1111_2$ to decimal. Treat all numbers as *unsigned*.

2.1.9 Convert 11.01_2 to decimal. Provide the full answer without limiting its accuracy or rounding. Treat all numbers as *unsigned*.

2.1.10 Convert $F3_{16}$ to decimal. Treat all numbers as *unsigned*.

2.1.11 Convert $EE.0F_{16}$ to decimal. Provide the full answer without limiting its accuracy or rounding. Treat all numbers as *unsigned*.

2.1.12 Convert 67_{10} to binary. Treat all numbers as *unsigned*.

2.1.13 Convert 1.4375_{10} to binary. Provide the full answer without limiting its accuracy or rounding. Treat all numbers as *unsigned*.

2.1.14 Convert 67_{10} to hexadecimal. Treat all numbers as *unsigned*.

2.1.15 Convert 10.6640625_{10} to hexadecimal. Provide the full answer without limiting its accuracy or rounding. Treat all numbers as *unsigned*.

2.1.16 Compute $1010_2 + 1011_2$ by hand. Treat all numbers as *unsigned*. Provide the 4-bit sum and indicate whether a *carryout* occurred.

2.1.17 Compute $1111\ 1111_2 + 0000\ 0001_2$ by hand. Treat all numbers as *unsigned*. Provide the 8-bit sum and indicate whether a *carryout* occurred.

2.1.18 Compute $1010_2 - 1011_2$ by hand. Treat all numbers as *unsigned*. Provide the 4-bit difference and indicate whether a *borrow in* occurred.

2.1.19 Compute $1111\ 1111_2 - 0000\ 0001_2$ by hand. Treat all numbers as *unsigned*. Provide the 8-bit difference and indicate whether a *borrow in* occurred.

2.1.20 What range of decimal numbers can be represented by *8-bit, two's complement* numbers?

2.1.21 What range of decimal numbers can be represented by *16-bit, two's complement* numbers?

2.1.22 What is the 8-bit, two's complement code for $+88_{10}$?

2.1.23 What is the 8-bit, two's complement code for -88_{10}?

2.1.24 What is the decimal value of the 4-bit, two's complement code 0010_2?

2.1.25 What is the decimal value of the 8-bit, two's complement code $1111\ 1110_2$?

2.1.26 Compute $1110_2 + 1011_2$ by hand. Treat all numbers as *4-bit, two's complement codes*. Provide the 4-bit sum and indicate whether *two's complement overflow* occurred.

2.1.27 Compute $1101\ 1111_2 + 0000\ 0001_2$ by hand. Treat all numbers as *8-bit, two's complement codes*. Provide the 8-bit sum and indicate whether *two's complement overflow* occurred.

Section 2.2: Combinational Logic

2.2.1 For the 3-input truth table in Fig. 2.50, give the *canonical sum of products (SOP) logic expression*.

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Fig. 2.50
Combinational logic synthesis (1)

2.2.2 For the 3-input truth table in Fig. 2.50, give the *canonical sum of products (SOP) logic diagram*.

2.2.3 For the 3-input truth table in Fig. 2.50, give the *minterm list*.

2.2.4 For the 3-input truth table in Fig. 2.50, give the *canonical product of sums (POS) logic expression*.

2.2.5 For the 3-input truth table in Fig. 2.50, give the *canonical product of sums (POS) logic diagram*.

2.2.6 For the 3-input truth table in Fig. 2.50, give the *maxterm list*.

2.2.7 For the 3-input truth table in Fig. 2.51, give the *canonical sum of products (SOP) logic expression*.

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Fig. 2.51
Combinational logic synthesis (2)

2.2.8 For the 3-input truth table in Fig. 2.51, give the *canonical sum of products (SOP) logic diagram*.

2.2.9 For the 3-input truth table in Fig. 2.51, give the *minterm list*.

2.2.10 For the 3-input truth table in Fig. 2.51, give the *canonical product of sums (POS) logic expression*.

2.2.11 For the 3-input truth table in Fig. 2.51, give the *canonical product of sums (POS) logic diagram*.

2.2.12 For the 3-input truth table in Fig. 2.51, give the *maxterm list*.

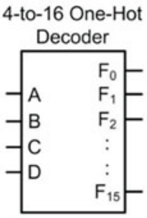
2.2.13 For the 3-input truth table in Fig. 2.50, give the *minimized sum of products (SOP) logic expression*.

2.2.14 For the 3-input truth table in Fig. 2.50, give the *minimized product of sums (POS) logic expression*.

2.2.15 For the 3-input truth table in Fig. 2.51, give the *minimized sum of products (SOP) logic expression*.

2.2.16 For the 3-input truth table in Fig. 2.51, give the *minimized product of sums (POS) logic expression*.

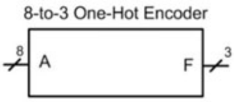
2.2.17 Design a 4-to-16 one-hot decoder by hand. The block diagram and truth table for the decoder are given in Fig. 2.52. Give the minimized logic expressions for each output (i.e., F_0, F_1, . . ., F_{15}) and the full logic diagram for the system.



A	B	C	D	F_{15}	F_{14}	F_{13}	F_{12}	F_{11}	F_{10}	F_9	F_8	F_7	F_6	F_5	F_4	F_3	F_2	F_1	F_0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fig. 2.52
4-to-16 one-hot decoder functionality

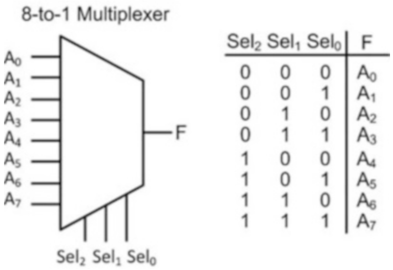
2.2.18 Design an 8-to-3 binary encoder by hand. The block diagram and truth table for the encoder are given in Fig. 2.53. Give the logic expressions for each output and the full logic diagram for the system.



A(7)	A(6)	A(5)	A(4)	A(3)	A(2)	A(1)	A(0)	F(2)	F(1)	F(0)
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Fig. 2.53
8-to-3 one-hot encoder functionality

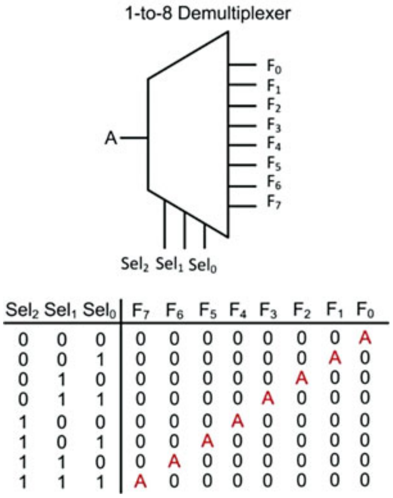
2.2.19 Design an 8-to-1 multiplexer by hand. The block diagram and truth table for the multiplexer are given in Fig. 2.54. Give the minimized logic expressions for the output and the full logic diagram for the system.



Sel_2	Sel_1	Sel_0	F
0	0	0	A_0
0	0	1	A_1
0	1	0	A_2
0	1	1	A_3
1	0	0	A_4
1	0	1	A_5
1	1	0	A_6
1	1	1	A_7

Fig. 2.54
8-to1 multiplexer functionality

2.2.20 Design a 1-to-8 demultiplexer by hand. The block diagram and truth table for the demultiplexer are given in Fig. 2.55. Give the minimized logic expressions for each output and the full logic diagram for the system.



Sel_2	Sel_1	Sel_0	F_7	F_6	F_5	F_4	F_3	F_2	F_1	F_0
0	0	0	0	0	0	0	0	0	0	A
0	0	1	0	0	0	0	0	0	A	0
0	1	0	0	0	0	0	0	A	0	0
0	1	1	0	0	0	0	A	0	0	0
1	0	0	0	0	0	A	0	0	0	0
1	0	1	0	0	A	0	0	0	0	0
1	1	0	0	A	0	0	0	0	0	0
1	1	1	A	0	0	0	0	0	0	0

Fig. 2.55
1-to-8 demultiplexer functionality

Section 2.3: Sequential Logic

2.3.1 How does the width of a register relate to the number of D-Flip-Flops used in the circuit?

2.3.2 For the state diagram in Fig. 2.56, how many D-Flip-Flops will this machine take if the states are encoded in *binary*?

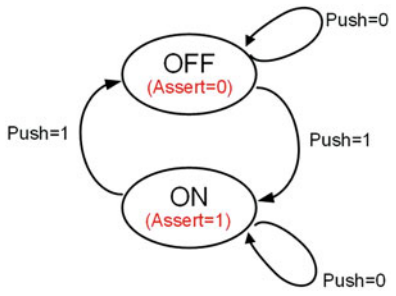


Fig. 2.56
FSM 0 state diagram

2.3.3 For the state diagram in Fig. 2.56, how many D-Flip-Flops will this machine take if the states are encoded in *gray code*?

2.3.4 For the state diagram in Fig. 2.56, how many D-Flip-Flops will this machine take if the states are encoded in *one-hot*?

2.3.5 For the state diagram in Fig. 2.56, is this a Mealy or Moore machine?

The next set of questions are about the design of a finite state machine by hand to implement the behavior described by the state diagram in Fig. 2.56. For this design, you will name the current state variable Q0_cur and name the next state variable Q0_nxt. You will also use the following state codes:

OFF = '0'

ON = '1'

2.3.6 For the state diagram in Fig. 2.56, what is the next state logic expression for Q0_nxt?

2.3.7 For the state diagram in Fig. 2.56, what is the output logic expression for Assert?

2.3.8 For the state diagram in Fig. 2.56, provide the final logic diagram for this machine.

2.3.9 For the state diagram in Fig. 2.57, how many D-Flip-Flops will this machine take if the states are encoded in *binary*?

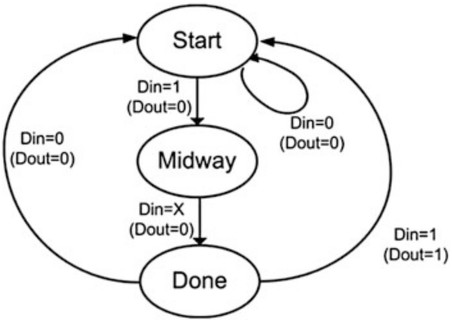


Fig. 2.57
FSM 1sState diagram

2.3.10 For the state diagram in Fig. 2.57, how many D-Flip-Flops will this machine take if the states are encoded in *gray code*?

2.3.11 For the state diagram in Fig. 2.57, how many D-Flip-Flops will this machine take if the states are encoded in *one-hot*?

2.3.12 For the state diagram in Fig. 2.57, is this a Mealy or Moore machine?

The next set of questions are about the design of a finite state machine by hand to implement the behavior described by the state diagram in Fig. 2.57. For this design, you will name the current state variables Q1_cur and Q0_cur and name the next state variables Q1_nxt and Q0_nxt. You will also use the following state codes:

Start = "00"

Midway = "01"

Done = "10"

2.3.13 For the state diagram in Fig. 2.57, what is the next state logic expression for Q1_nxt?

2.3.14 For the state diagram in Fig. 2.57, what is the next state logic expression for Q0_nxt?

2.3.15 For the state diagram in Fig. 2.57, what is the output logic expression for Dout?

2.3.16 For the state diagram in Fig. 2.57, provide the final logic diagram for this machine.

Section 2.4: Memory

2.4.1 For a 512k × 32 memory system, how many unique address locations are there? Give the exact number. Remember that 512k is shorthand for 524,288.

2.4.2 For a 512k × 32 memory system, what is the data width at each address location? Remember that 512k is shorthand for 524,288.

2.4.3 For a 512k × 32 memory system, what is the *capacity* in *bits*? Remember that 512k is shorthand for 524,288.

2.4.4 For a 512k × 32-bit memory system, what is the *capacity* in *bytes*? Remember that 512k is shorthand for 524,288.

2.4.5 For a 512k × 32 memory system, how wide does the incoming address bus need to be in order to access every unique address location? Remember that 512k is shorthand for 524,288.

2.4.6 Name the *type of memory* with the following characteristic: *when power is removed, the data is lost.*

2.4.7 Name the *type of memory* with the following characteristic: *when power is removed, the memory still holds its information.*

2.4.8 Name the *type of memory* with the following characteristic: *it can only be read from during normal operation.*

2.4.9 Name the *type of memory* with the following characteristic: *during normal operation, it can be read and written to.*

2.4.10 Name the *type of memory* with the following characteristic: *data can be accessed from any address location at any time.*

2.4.11 Name the *type of memory* with the following characteristic: *data can only be accessed in consecutive order; thus, not every location of memory is available instantaneously.*

2.4.12 Name the *type of memory* with the following characteristic: *this memory is non-volatile, read/write, and only provides data access in blocks.*

2.4.13 Name the *type of memory* with the following characteristic: *this memory uses a floating-gate transistor, can be erased with electricity, and provides individual bit access.*

2.4.14 Name the *type of memory* with the following characteristic: *this memory is non-volatile, read/write, and provides word-level data access.*

2.4.15 Name the *type of memory* with the following characteristic: *this memory uses a floating-gate transistor that is erased with UV light.*

2.4.16 Name the *type of memory* with the following characteristic: *this memory is programmed by blowing fuses or anti-fuses.*

2.4.17 Name the *type of memory* with the following characteristic: *this memory is partially fabricated prior to knowing the information to be stored.*

2.4.18 How many transistors does it take to implement an SRAM cell?

2.4.19 Why is a DRAM cell referred to as a 1T 1C configuration?

2.4.20 What is the key difference between a DRAM storage cell and an FRAM storage cell?

Chapter 3: Computer Systems

This chapter introduces the general organization of a computer system [4, 5, 6, 10]. The chapter describes the key hardware components in a computer including the central processing unit, registers, the control unit, the arithmetic/logic unit, program memory, data memory, and input/output. Software is then introduced as a set of computer-specific instructions that are inserted in program memory in a particular order to accomplish a desired task. Then a basic overview of how an instruction is executed is covered. The primary goal of this chapter is to introduce the key components of a computer and all of the terminology that will be used throughout the rest of the book, plus provide a basic understanding of how computers execute instructions.

Learning Outcomes—After completing this chapter you will be able to:

3.1 Describe the basic concept of a computer system.
3.2 Describe the basic components and operation of computer hardware.
3.3 Describe the basic components and operation of computer software.

3.1 Computer Overview

A computer is a collection of hardware and software working together to accomplish a task. Every computer has a set of specific *instructions* that it is designed to execute. This group of instructions is called its *instruction set*. Computers have instructions that can move information in and out of memory, manipulate the information using arithmetic/logical operations, and control the flow of the instruction execution. Each instruction in the set has a unique operation code, or *op-code*, which is a binary code that identifies the instruction. Individual instructions are relatively simple; however, when a large sequence of instructions is executed in a particular order, a computer is able to accomplish very complex tasks. While it may take executing millions of instructions to accomplish anything meaningful, modern computers are able to execute the instructions so rapidly that to a human they seem nearly instantaneous. Computer *software* refers to the sequence of instructions that when executed one by one, will accomplish a desired task. The sequence of instructions that perform the task is called the *program*. A *software developer*, or *programmer*, is a person who designs the program by deciding which instructions and in what order to use.

The computer *hardware* provides all of the necessary functionality to store the program, retrieve the individual instructions from memory, and execute them. The hardware also includes additional useful circuits such as fast storage and data manipulation logic that can be used by the program. The computer also contains circuitry that allows the program to interact with the outside world. The term “hardware” refers to all of the physical components within the system. Specific circuitry that comprises computer hardware includes memory devices, registers, finite state machines, combinational logic, and the bus system to move data between sub-systems.

B. J. LaMeres, *Embedded Systems Design using the MSP430FR2355 LaunchPad™*,
https://doi.org/10.1007/978-3-031-20888-1_3

Concept Check

CC3.1 Does a software developer decide which instructions that the computer should be able to execute when designing a program?

A) No. The instructions that a computer can execute are part of its built-in architecture. The software developer instead puts together a collection of these built-in instructions in a certain order so that when executed will accomplish a task.

B) Yes. The developer can create any instruction needed to accomplish the desired task.

C) Yes. A developer has complete control over what the computer can do.

D) Yes. Of course they usually try to use the computer's existing instructions. But if needed, they can change the computer's architecture.

3.2 Computer Hardware

Figure 3.1 shows a block diagram of the basic hardware components in a computer.

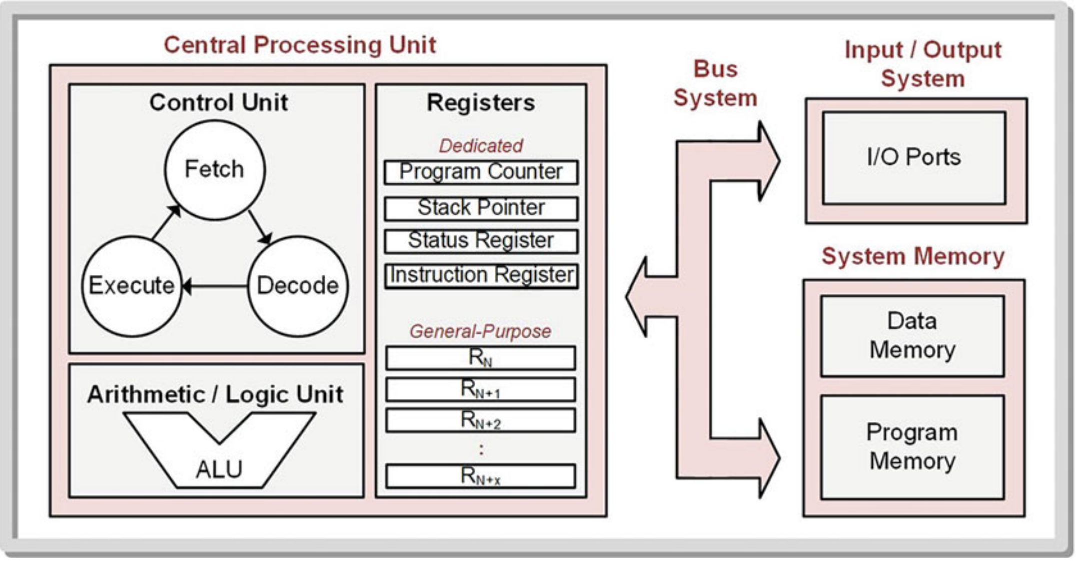


Fig. 3.1
Hardware components of a computer system

3.2.1 Program Memory

A computer accomplishes a desired task by executing instructions in a particular order or by selectively executing some instructions while skipping others. In order for the computer hardware to execute the instructions, they must reside within the computer hardware. The storage system that holds the instructions is called *program memory*. Program memory is treated as read only memory during execution in order to prevent the instructions from being overwritten by the computer. Some computer systems will implement the program memory on a true ROM device (MROM or PROM), while others will use a EEPROM that can be read from during normal operation but can only be written to using a dedicated write procedure. Programs for microcontrollers are typically held in non-volatile memory so that the computer system does not lose its program when power is removed. Modern general-purpose computers will often copy programs from non-volatile memory (e.g., a hard disk drive) to volatile memory (i.e., “RAM”) after startup in order to speed up instruction execution. In this case, care must be taken that the program does not overwrite itself.

3.2.2 Data Memory

Computers also contain *data memory*, which can be written to and read from during normal operation. This memory is used to hold temporary variables that are created by the software program. This memory expands the capability of the computer system by allowing large amounts of information to be created and stored by the program. Additionally, computations can be performed that are larger than the width of the computer’s circuitry by holding interim portions of the calculation (e.g., performing a 128-bit addition using 16-bit adders). Data memory is implemented with R/W memory, most often SRAM or DRAM. When someone talks about the “RAM” in a computer, they are talking about the data memory.

3.2.3 Central Processing Unit

The *central processing unit* (CPU) is considered the *brains* of the computer because it facilitates all steps involved in executing instructions. The CPU handles reading instructions from memory, decoding the op-code to determine which instruction is being performed, and executing the necessary steps to complete the instruction. The CPU also contains a set of registers that are used for general-purpose data storage, operational information, and system status. Finally, the CPU contains circuitry to perform arithmetic and logic operations on data.

3.2.3.1 Control Unit

The *control unit* is a finite state machine that controls the operation of the computer. This FSM has states that perform fetching the instruction (i.e., reading it from program memory), decoding the instruction op-code, and executing the appropriate steps to accomplish the functionality of the instruction. This process is known as the *fetch* → *decode* → *execute* cycle and is repeated each time an instruction is executed. As the control unit FSM traverses through its states, it asserts control signals that move and manipulate data to achieve the desired functionality. Figure 3.2 shows a conceptual model for the control unit FSM. The state diagram contains states that perform fetching the op-code from program memory and decoding it to determine which instruction is being executed. Then a sequence of instruction-specific states are traversed to accomplish the desired functionality of the instruction. Each instruction that the CPU can execute can be visualized as a separate sequence of execution states. After execution, the FSM returns to the fetch state to retrieve the next op-code in program memory corresponding to the next instruction in program memory.

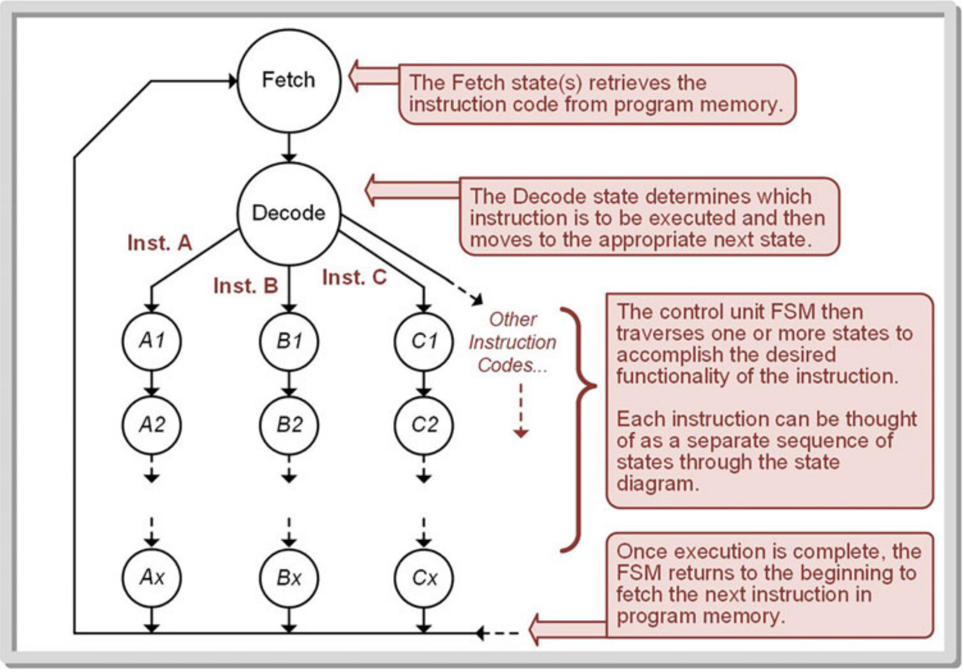

Fetch
Decode
Inst. A
Inst. B
Inst. C
A1
A2
Ax
B1
B2
Bx
C1
C2
Cx
Other Instruction Codes...
The Fetch state(s) retrieves the instruction code from program memory.
The Decode state determines which instruction is to be executed and then moves to the appropriate next state.
The control unit FSM then traverses one or more states to accomplish the desired functionality of the instruction.
Each instruction can be thought of as a separate sequence of states through the state diagram.
Once execution is complete, the FSM returns to the beginning to fetch the next instruction in program memory.

Fig. 3.2
Conceptual model for the control unit FSM

3.2.3.2 Registers

The CPU contains a variety of registers that are necessary to execute instructions and hold status information about the system. Different computer architectures have different number of registers and also different register names; however, the purpose of the registers is essentially the same across basic computer architectures. There are two groups of registers: dedicated and general-purpose. Dedicated registers are used as part of the normal operation of the computer and are generally not under the control of the program. General-purpose registers can be used by the program as needed. The following is a description of the common registers within a computer CPU.

- **Program Counter (PC)** – The program counter holds the address of next instruction in program memory to execute. In the fetch state of the control unit FSM, the op-code is read from this address in program memory. As soon as this address is used to retrieve the op-code of the current instruction, the PC is incremented to the next address location in program memory. By incrementing to the next location in memory, the control unit knows where to read the next op-code after the current instruction completes. When a computer first powers up, the PC must be initialized with the address of the first instruction in the program.
- **Stack Pointer (SP)** – The stack pointer provides a way to dynamically allocate variable space in the data memory without having to keep track of specific addresses. The concept of a *stack* and its use in the MSP430 will be covered later.
- **Status Register (SR)** – The status register contains status bits, or *flags*, that are asserted when various conditions occur during the execution of the program. The most commonly used status bits within the SR are two’s complement overflow (V), negative (N), zero (Z), and carry (C). These flags are updated based on the results of operations in the ALU.
- **Instruction Register (IR)** – This register is used to hold the op-code/operand that is fetched from program memory during the first part each instruction.
- **General-Purpose Registers** – These registers can be used for anything the program wants.

3.2.3.3 Arithmetic Logic Unit (ALU)

The *arithmetic logic unit* (ALU) is the system that performs all mathematical (i.e., add, subtract, increment, decrement) and logic operations (i.e., AND, OR, XOR, setting bits, clearing bits). This system operates on data being held in CPU registers. The ALU has a unique symbol associated with it to distinguish it from other functional units in the CPU as shown in Fig. 3.1. The ALU also contains logic to produce status bits that provide the ability for subsequent instructions to react to particular results. The status bits coming from the ALU are carry (C), negative (N), zero (Z), and two's complement overflow (V). These bits are latched into the status register upon completion of the ALU operation.

3.2.4 Input/Output Ports

A computer is only useful if it can interact with the outside world. The hardware component used to access the outside world is called a *port*. Ports can be input, output, or bidirectional. Input/output (I/O) ports can be designed to pass information in a parallel or serial format. Parallel ports pass data as a bus and allow more information to be transferred per instruction. Serial ports use a single line and send data bit by bit. This has the advantage that it requires fewer pins on a device; however, serial buses typically take more time to transmit the same information compared to parallel ports.

3.2.5 Bus System

The bus system of a computer handles routing all signals between the CPU and memory. The bus system contains a *memory address bus* (MAB) that provides a single address to data memory, program memory, and the I/O ports. The system also contains *memory data bus* (MDB) that carries information back and forth between the CPU and the memory and I/O ports. Various control signals are also included in the bus system to facilitate reading and writing. One key concept of the bus system is that the I/O ports, data memory, and program memory share the address and data buses. Every specific location, regardless of being an I/O port, a location in data memory, or a location in program memory, is assigned a unique address. This is called a *memory mapped system*. Each microcontroller has a specific *memory map*, which gives the addresses for all locations in memory. The detailed memory map for the MSP430 will be covered later. Figure 3.3 shows an updated block diagram of the hardware components in a computer with the I/O, data memory, and program memory logically grouped within a memory system block.

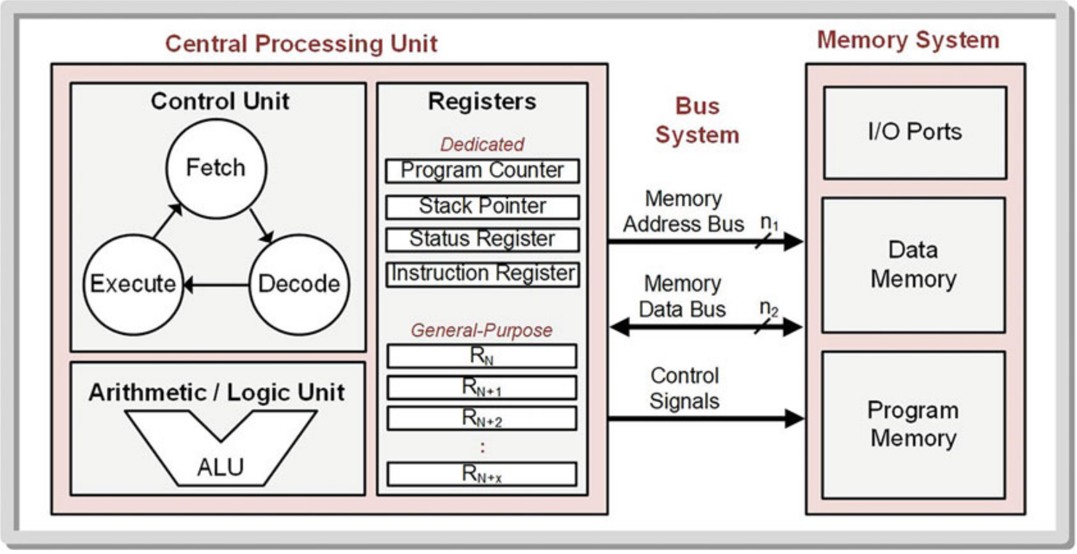


Fig. 3.3
Computer hardware overview with memory mapped organization

Concept Check

CC3.2 Why is it important to have the first program memory that is accessed on power-up be non-volatile?

A) Because when the computer powers up, it will start the fetch → decode → execute cycle. The computer needs a sequence of instructions to be immediately available so that it can start operating.

B) It isn't. Computers run without programs all the time.

C) So that the computer can run programs when the power is removed.

D) So that we have an equal amount of volatile and non-volatile memory in order ensure the computer is balanced.

3.3 Computer Software

As mentioned in Sect. 3.1 computer software is a collection of instructions that have been placed in a particular order that when executed, accomplishes a task. A software programmer creates the program by deciding which instructions to use and in what order. A computer is designed to execute a unique set of instructions, which is called its instruction set. Some computer systems have a very small number of instructions in order to reduce the physical size of the circuitry needed in the CPU. This allows the CPU to execute the instructions very quickly but requires a large number of operations to accomplish a given task. This architectural approach is called a *reduced instruction set computer* (RISC). The alternative to this approach is to make an instruction set with a large number of dedicated instructions that can accomplish a given task in fewer CPU operations. The drawback of this approach is that the physical size of the CPU must be larger in order to accommodate the various instructions. This architectural approach is called a *complex instruction set computer* (CISC).

3.3.1 Classes of Instructions

There are three general classes of instructions in a computer: (1) data movement, (2) data manipulation, and (3) program flow.

3.3.1.1 Data Movement Instructions

Data movement instructions move information between the CPU and memory or between two memory locations. Data movement instructions do not use the ALU, but rather assert control signals within the bus system to facilitate the routing and storing of information between a source and destination.

3.3.1.2 Data Manipulation Instructions

Data manipulation instructions use the ALU to perform arithmetic or logical operations on information. Examples of data manipulation instructions are addition, subtraction, ANDs, XORs, increments, decrements, bit-sets, and bit-clears.

3.3.1.3 Program Flow Instructions

As mentioned in Sect. 3.2.3.2, the program counter tracks the current location in program memory of the instruction to be executed. During the normal fetch→decode→execute cycle, the instruction is read from the current address held in the program counter during the fetch. It is then assumed that the next instruction will reside in program memory immediately following the current instruction. Based on this assumption, the program counter is incremented to point to the next location in program memory so that when the current instruction completes, the CPU is ready to fetch the next instruction. If there was no way to alter the program counter beyond what was just described, the only type of program that could be written would be a serial sequence of instructions without any ability to repeat (i.e., loop) or selectively execute certain instructions under certain conditions (i.e., conditional statements such as if/else, case, etc.). *Program flow* instructions provide the ability to alter the program counter to support looping and conditional statement functionality.

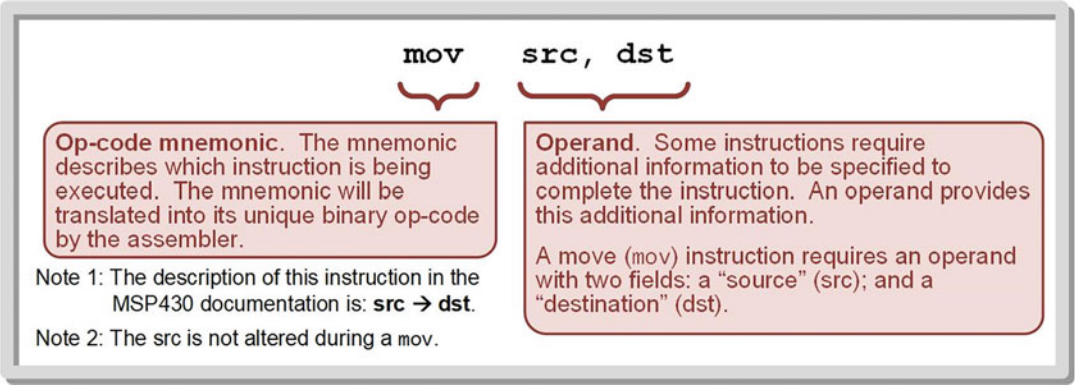
Computers contain *unconditional program flow* instructions and *conditional program flow* instructions. Unconditional program flow instructions always change the program counter to a fixed value. Unconditional instruction supports loop-forever functionality. Conditional program flow instructions only alter the program counter when certain conditions exist within the status register, specifically assertions on the VNZC bits. If a certain SR condition exists, then the conditional instruction will change the program counter to a new value. If the SR condition does not exist, the program counter is incremented as in normal operation. This behavior allows programs to conditionally execute sections of instructions in program memory to support behavior such as while loops, if/else, and case statements.

3.3.2 Op-codes and Operands

A computer instruction contains two main components, an *op-code* and an *operand*. The op-code is a unique binary code given to each instruction in the set. The CPU decodes the op-code in order to know which instruction is being executed and then takes the appropriate steps to complete the instruction. Each op-code is assigned a *mnemonic*, which is a descriptive name for the op-code that can be used when discussing the instruction functionally. For example, a data movement instruction may have a mnemonic of **`mov`**; an addition instruction may have a mnemonic of **`add`**; and an increment instruction may have a mnemonic of **`inc`**. The mnemonics allow us to communicate which instruction we are talking about and are also used in low-level programming languages. A low-level language that will be used throughout this book is called *assembly*. In assembly, a program is created by listing out each and every instruction to be placed into program memory using its mnemonic and any other information needed by the instruction. A piece of software called an *assembler* then translates the mnemonics and additional information for each instruction into its specific op-code so that the program can be placed into program memory.

The second part of an instruction is an *operand*. The operand provides additional information needed to complete the instruction. For example, if we wanted to perform a **`mov`** instruction, we would need to tell the CPU where the information to be moved currently resides and where it is going. By using the concept of an operand, a single instruction op-code can perform a large number of unique move operations. These include moving/copying information: from memory into any of the general-purpose CPU registers, from any CPU register into memory, between any two CPU registers, and between any two memory locations. In the MSP430, instructions use the concept of a *source (src)* and a *destination (dst)* to construct the operand. When both a source and destination are needed, the src is listed first and the dst is listed second separated by a comma. The operand is provided after the mnemonic, space, or tab delimited.

Let's examine the anatomy of an instruction by breaking down the **mov** instruction as shown in Fig. 3.4. This instruction will copy the contents from the src location into the dst location. This instruction example is beginning to take the form of how a line of an assembly program will look. Each line represents an instruction that will be placed into program memory. The first item listed is the instruction mnemonic. In this case, **mov** is used to indicate the instruction. The second item listed is the operand. In this case, the operand is a src and dst. The assembler will translate this line into a unique binary code consisting of the op-code and operand that can be placed into program memory and executed by the CPU.



```
mov    src, dst
```

Op-code mnemonic. The mnemonic describes which instruction is being executed. The mnemonic will be translated into its unique binary op-code by the assembler.

Note 1: The description of this instruction in the MSP430 documentation is: **src → dst**.

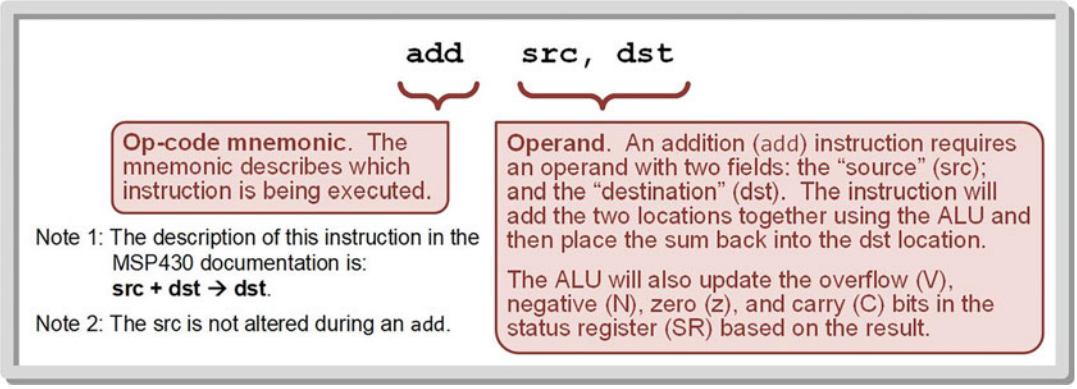
Note 2: The src is not altered during a mov.

Operand. Some instructions require additional information to be specified to complete the instruction. An operand provides this additional information.

A move (mov) instruction requires an operand with two fields: a "source" (src); and a "destination" (dst).

Fig. 3.4
The anatomy of a `mov` instruction

Let's take a look at another instruction that uses src and dst within the operand, but is slightly more complicated. The addition (**add**) instruction will add two inputs using the ALU. Figure 3.5 shows the anatomy of an **add** instruction. This instruction will add the src to the dst, but the result of the addition must be placed somewhere in the CPU. This instruction is designed to put the sum back into the dst register. The src register is not altered. Additionally, the ALU will update the overflow (V), negative (N), zero (z), and carry © bits in the status register.



```
add    src, dst
```

Op-code mnemonic. The mnemonic describes which instruction is being executed.

Note 1: The description of this instruction in the MSP430 documentation is: **src + dst → dst**.

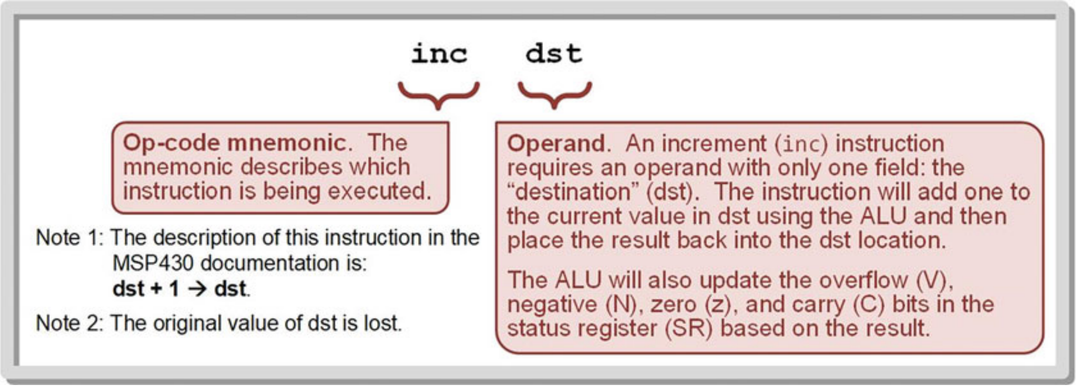
Note 2: The src is not altered during an add.

Operand. An addition (add) instruction requires an operand with two fields: the "source" (src); and the "destination" (dst). The instruction will add the two locations together using the ALU and then place the sum back into the dst location.

The ALU will also update the overflow (V), negative (N), zero (z), and carry (C) bits in the status register (SR) based on the result.

Fig. 3.5
The anatomy of an `add` instruction

Let's look at one more instruction that only requires a single item in the operand. The increment (**inc**) instruction will add one to the location specified in the operand and then place the new value back into the dst location. Figure 3.6 shows the anatomy of an **inc** instruction. This instruction only requires the dst to be specified in the operand. Additionally, the ALU will update the overflow (V), negative (N), zero (Z), and carry (C) bits in the status register. The original contents of dst prior to the increment are lost.



Op-code mnemonic. The mnemonic describes which instruction is being executed.

Note 1: The description of this instruction in the MSP430 documentation is:
dst + 1 → dst.

Note 2: The original value of dst is lost.

Operand. An increment (`inc`) instruction requires an operand with only one field: the "destination" (dst). The instruction will add one to the current value in dst using the ALU and then place the result back into the dst location.

The ALU will also update the overflow (V), negative (N), zero (z), and carry (C) bits in the status register (SR) based on the result.

Fig. 3.6
Anatomy of an `inc` instruction

Concept Check

CC3.3 Software development consists of choosing which instructions, and in what order, will be executed to accomplish a certain task. The group of instructions is called the *program* and is inserted into program memory. Which of the following might a software developer care about?

A) Minimizing the number of instructions that need to be executed to accomplish the task in order to increase the computation rate.

B) Minimizing the number of registers used in the CPU to save power.

C) Minimizing the overall size of the program to reduce the amount of program memory needed.

D) Both A and C.

3.3.3 Program Development Flow

Program development for the MSP430 can occur at multiple levels of abstraction. At the highest level, the language C can be used to develop programs without needing to understand the architecture of the CPU or how the internal registers are used to move and manipulate data. For programs developed in C, a *compiler* is used to convert the high-level programming constructs into individual assembly language instructions.

Programs can also be developed at the instruction level using an assembly language. In assembly language programming, each instruction is listed in a file using its mnemonic with the associated operands. Additionally, information can be provided in an assembly file that will dictate memory allocation for variables and also a variety of other settings for the MSP430. For programs developed in assembly, an *assembler* is used to translate the instruction mnemonics and operands into their corresponding binary codes. The binary file(s) that are created by the assembler are called *object files*. At this level of the development cycle, the binary files (or *binaries* for short) have not been assigned to specific memory locations within the MSP430.

A *linker* is a tool within the development flow that joins multiple source files and assigns the final addresses for the program code. The reason that the binaries are not assigned to specific addresses prior to the linker step is so that programs can be developed for any device within the MSP430 family and leave the decision about which specific part number will be used until the program is complete. Many times, it is unknown how much memory or peripherals will be needed until the program is finished. Leaving the memory assignment until the linker stage allows the programmer to continue development while postponing the final device selection until the end when the final requirements of the software are known. The output of the linker stage is an *executable object file* that is ready to be downloaded to the MSP430 using an *EEPROM programmer*.

Once the program has been downloaded into the non-volatile memory of the MSP430, the program can be run or debugged. A *debugger* is a tool that allows the program to be executed instruction by instruction, or *stepped*, and the values within the CPU registers and memory to be observed after each instruction is executed. This allows the developer to debug the program if it is not operating properly. Figure 3.7 shows a flow chart of the MSP430 development cycle. Note that in this textbook we will first learn about the operation of the MSP430 by programming it in assembly. This will allow us to learn the detailed architecture of the computer system and understand the intricacies of instruction execution and memory usage. Then we will move into a higher level of abstraction and begin programming the MSP430 in C. Programming in C will allow more complex programs to be created in a reasonable amount of time and is how embedded computer programs are typically developed.

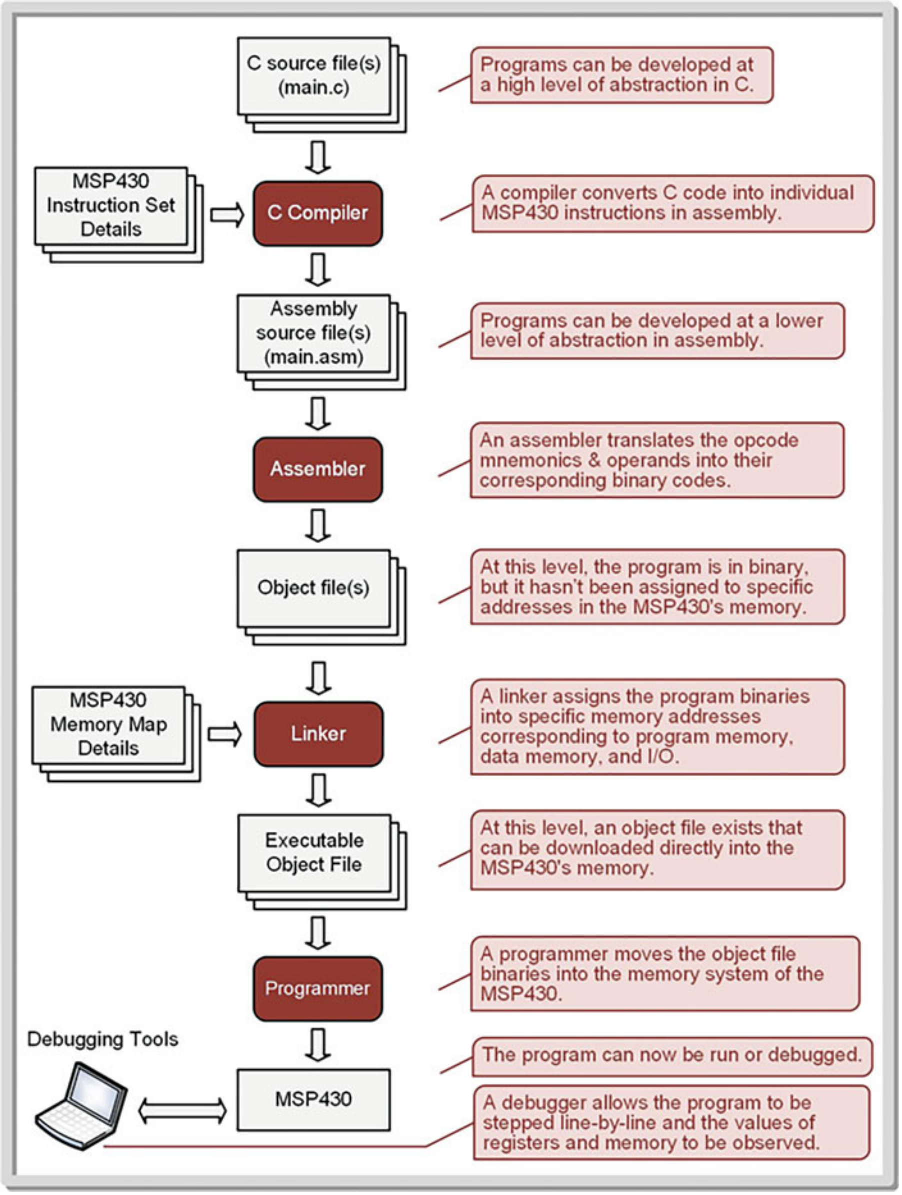

C source file(s) (main.c)
Programs can be developed at a high level of abstraction in C.
MSP430 Instruction Set Details
C Compiler
A compiler converts C code into individual MSP430 instructions in assembly.
Assembly source file(s) (main.asm)
Programs can be developed at a lower level of abstraction in assembly.
Assembler
An assembler translates the opcode mnemonics & operands into their corresponding binary codes.
Object file(s)
At this level, the program is in binary, but it hasn't been assigned to specific addresses in the MSP430's memory.
MSP430 Memory Map Details
Linker
A linker assigns the program binaries into specific memory addresses corresponding to program memory, data memory, and I/O.
Executable Object File
At this level, an object file exists that can be downloaded directly into the MSP430's memory.
Programmer
A programmer moves the object file binaries into the memory system of the MSP430.
Debugging Tools
MSP430
The program can now be run or debugged.
A debugger allows the program to be stepped line-by-line and the values of registers and memory to be observed.

Fig. 3.7
MSP430 program development flow

Summary

- A computer is a collection of hardware and software working together to accomplish a task.
- Computer hardware components are constructed to execute a specific set of instructions.
- The instructions that a computer is designed to execute are called its *instruction set*.
- The main hardware components of a computer are the central processing unit (CPU), program memory, data memory, and input/output ports.
- The CPU consists of registers for fast storage, an arithmetic logic unit (ALU) for data manipulation, and a control state machine

that directs all activity to execute an instruction.

- The control unit continuously performs a *fetch-decode-execute* cycle in order to complete instructions.
- A *memory mapped* system is one in which the program memory, data memory, and I/O ports are all assigned a unique address. This allows the CPU to simply process information as data and addresses and allows the program to handle where the information is being sent to. A *memory map* is a graphical representation of what address ranges various components are mapped to.
- Instructions are inserted into *program memory* in a sequence that when executed will accomplish a particular task. This sequence of instructions is called computer software, or a program.
- There are three primary classes of instructions: data movement, data manipulation, and program flow.
- An instruction consists of an *op-code* and a potential *operand*. The op-code is the unique binary code that tells the control state machine which instruction is being executed. An operand is additional information that may be needed for the instruction.
- Data movement instructions copy information between memory and CPU registers, between CPU registers, or between memory locations.
- Data manipulation instructions perform ALU operations on information being held in CPU registers.
- Program flow instructions alter the flow of instruction execution by altering the program counter.
- *Unconditional* program flow instructions always change the PC to a specific address.
- *Conditional* program flow instructions only change the PC under certain conditions dictated by status flags in the status register.
- The status flags are held in the status register. The most commonly used flags are the negative flag (N), zero flag (Z), two's complement overflow flag (V), and carry flag (C).
- Programs can be developed in higher-level languages such as C or in lower-level languages such as assembly.
- When developing in C, a compiler is used to convert the program into an assembly level file.
- An assembly program is a program written at the instruction level but using the instruction mnemonics instead of the instruction op-codes to make more readable.
- An assembler is used to convert an assembly source file into unique binary codes that the CPU understands.
- A linker contains the information about the computer system's memory map and assigns the output of the assembler into specific addresses within the memory system.
- Once downloaded onto the computer, a debugger can be used to step the program instruction by instruction and provide the values of the CPU registers and memory at each step. This allows the program to be evaluated for proper operation.

Exercise Problems

Section 3.1: Computer Overview

3.1.1 What is the basic description of a computer?

3.1.2 What is the basic description of computer hardware?

3.1.3 What is the basic description of a computer software?

3.1.4 If instructions are simple operations, how can a computer perform meaningful tasks?

3.1.5 What is the list of instructions that a computer is designed to execute called?

3.1.6 Can computer hardware do anything without software?

3.1.7 Can computer software do anything without hardware?

3.1.8 Once the sequence of instructions that will perform the task is designed, what is it called?

3.1.9 What is the person called that decides the sequence of instructions to be executed in order to accomplish a task?

Section 3.2: Computer Hardware

3.2.1 What computer hardware sub-system holds the temporary variables used by the program?

3.2.2 What computer hardware sub-system contains fast storage for holding and/or manipulating data and addresses?

3.2.3 What computer hardware sub-system allows the computer to interface to the outside world?

3.2.4 What computer hardware sub-system orchestrates the fetch-decode-execute process?

3.2.5 What computer hardware sub-system contains the circuitry that performs mathematical and logic operations?

3.2.6 What computer hardware sub-system holds the instructions being executed?

3.2.7 What computer hardware sub-system facilitates routing the signals back and forth between the various components in a computer?

3.2.8 What computer hardware sub-system groups the control unit, registers, and ALU together?

3.2.9 How does a memory mapped system make computer operation simpler?

Section 3.3: Computer Software

3.3.1 Which element of computer software is the binary code that tells the CPU which instruction is being executed?

3.3.2 Which element of computer software is the supplementary information required by an instruction such as constants or which registers to use?

3.3.3 Which class of instructions handles moving information between memory and CPU registers, between CPU registers, or between memory locations?

3.3.4 Which class of instructions alters the flow of program execution?

3.3.5 Which class of instructions alters data using either arithmetic or logical operations?

3.3.6 If you are developing a program in C, would you use a compiler or assembler to convert your code into a lower-level version of the instructions?

3.3.7 If an assembly program is essentially a list of instructions, why not just program using the op-codes and skip the assembler step?

3.3.8 What is the purpose of a linker?

3.3.9 What is the purpose of a debugger?

Chapter 4: The MSP430

This chapter introduces the MSP430 microcontroller family and then moves into the specifics of the MSP430FR2355 device that will be used throughout the rest of this book [1–3]. At this point, the reader should have an understanding of terminology and the basic operation of computer systems. The goal of this chapter is to provide the key details necessary to begin programming the MSP430FR2355 using the LaunchPad™ Development Kit. While the code examples presented through the rest of this book are applicable to other MSP430 MCUs, the purpose of this book is to provide specific code examples that can be directly run on the LaunchPad™ using the TI Code Composer Studio.

Learning Outcomes—After completing this chapter you will be able to:

4.1 Describe the basic components and operation of the MSP430 hardware.
4.2 Describe the basic components and operation of the MSP430 software.
4.3 Describe the basic components and operation of the MSP430FR2355 LaunchPad™ Development Kit.

4.1 MSP430 Hardware Overview

The MSP430 is a family of MCUs produced by Texas Instruments. The MSP430 family is based on a 16-bit CPU and is optimized for low-cost, low-power signal processing applications. The MSP430 family contains a general architecture that contains a 16-bit CPU, ROM and R/W memory, and an abundant suite of peripherals on a single chip. The MSP430's peripherals contain many signal processing capabilities, thus providing the rationale for its name *Mixed Signal Processor*. Figure 4.1 shows a simplified functional block diagram for the MSP430 MCU.

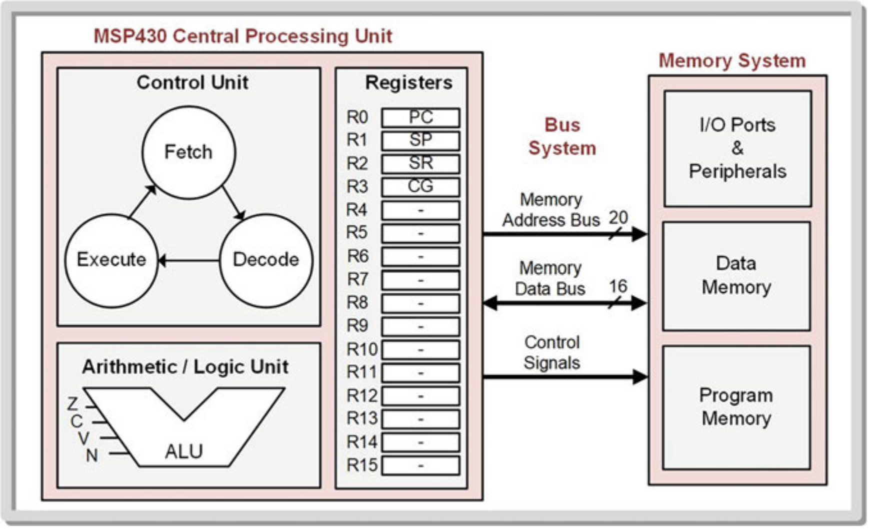


Fig. 4.1
MSP430 simplified functional block diagram

B. J. LaMeres, *Embedded Systems Design using the MSP430FR2355 LaunchPad™*,
https://doi.org/10.1007/978-3-031-20888-1_4

4.1.1 Word Versus Byte Memory Access

The MSP430 supports both byte and word memory access. Bytes are located at even or odd addresses. Words are located in the ascending memory locations aligned to even addresses with the low byte (LB) at the even address, followed by the high byte (HB) at the next odd address. Figure 4.2 shows the two memory models representing the same functionality within the MSP430 memory.

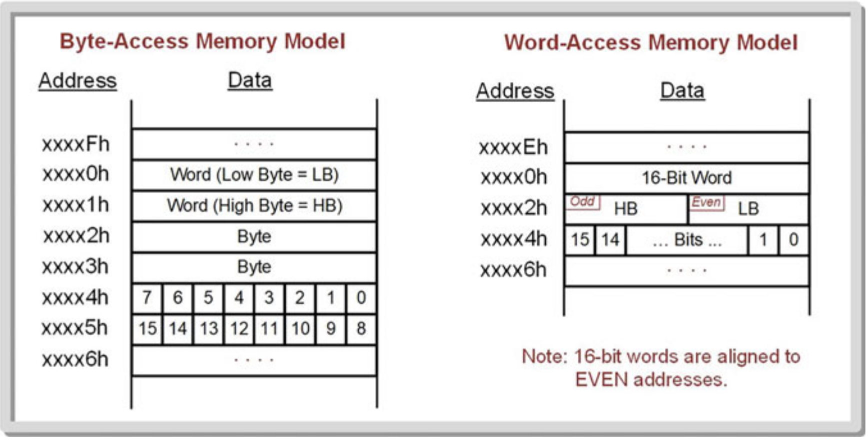


Fig. 4.2
Byte versus word access memory model

4.1.2 Program Memory

The MSP430 family supports varying sizes and technologies for non-volatile memory. Non-volatile memory sizes range from 0 to 512 kB. Technologies for the non-volatile memory include MROM, Flash, and FRAM. For FRAM-based devices, there are regions of the non-volatile memory that can be written to by the program for storage of data when power is removed.

4.1.3 Data Memory

The MSP430 documentation refers to its R/W, volatile memory as “RAM” throughout its documentation. To be consistent, this book will attempt to use the same terminology as the TI documentation. RAM sizes in the MSP430 range from 125 bytes to 66 kB. MSP430 RAM memory is implemented primarily with SRAM technology; however, some MCUs can also contain a small amount of FRAM for data memory.

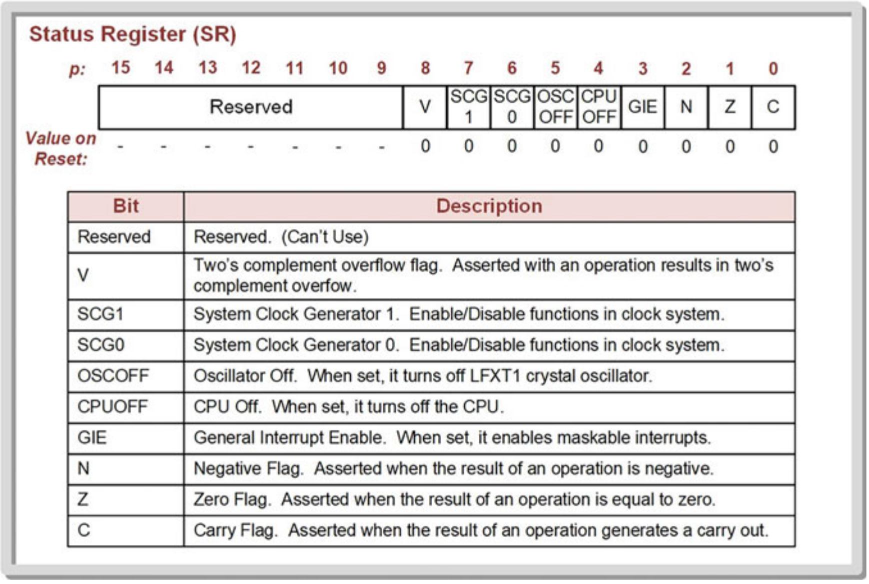
4.1.4 Central Processing Unit

The MSP430 family is based on a 16-bit RISC CPU.

4.1.4.1 Registers

The MSP430 CPU has 16 registers that are 20-bits wide. By default, the registers are operated on as 16-bit words; however, specific instructions exist to operate on the registers as either 20-bit or 8-bit words. The reason that the registers are 20-bits wide is so that they can be used to hold 20-bit addresses to support MCUs with 1 M of system memory. The registers are named R0, R1, R2, R3, . . ., R15. R0-R3 are special-purpose registers and cannot be manipulated directly by the software. R4 through R15 are general-purpose *registers* because they can be used in any manner by the program. The descriptions of the MSP430 registers are as follows:

- **R0: Program Counter (PC)** – The program counter holds the address of next instruction in program memory to execute. The program counter in the MSP430 is always used as a full 20-bit register, so it can provide access to $2^{20} = 1{,}048{,}576$ address locations (a.k.a. 1M) in the memory system.
- **R1: Stack Pointer (SP)** – The stack pointer also is used as a 20-bit address to access memory; however, it specifically addresses data memory. The SP provides a way to dynamically allocate variable space in the data memory without having to keep track of specific addresses. The concept of a *stack* and its use in the MSP430 will be covered later.
- **R2: Status Register (SR)** – The status register contains status bits, or *flags*, that are asserted when various conditions occur during the execution of the program. This register also contains bits that can be configured to turn on and off certain functions within the computer. Figure 4.3 provides the details of the flags within SR. The detailed use of these flags will be covered later in this book.



Status Register (SR)

p:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Reserved							V	SCG 1	SCG 0	OSC OFF	CPU OFF	GIE	N	Z	C
Value on Reset:	-	-	-	-	-	-	-	0	0	0	0	0	0	0	0	0

Bit	Description
Reserved	Reserved. (Can't Use)
V	Two's complement overflow flag. Asserted with an operation results in two's complement overfow.
SCG1	System Clock Generator 1. Enable/Disable functions in clock system.
SCG0	System Clock Generator 0. Enable/Disable functions in clock system.
OSCOFF	Oscillator Off. When set, it turns off LFXT1 crystal oscillator.
CPUOFF	CPU Off. When set, it turns off the CPU.
GIE	General Interrupt Enable. When set, it enables maskable interrupts.
N	Negative Flag. Asserted when the result of an operation is negative.
Z	Zero Flag. Asserted when the result of an operation is equal to zero.
C	Carry Flag. Asserted when the result of an operation generates a carry out.

Fig. 4.3
Status register (SR) details

- **R3: Constant Generator (GC)** – This register is used by the CPU to speed up the instruction execution. The details of its functionality are not key to understanding the basic operation of the CPU at this point.
- **R4 ➔ R15: General-Purpose Registers** – These registers can be used for anything the programmer wants.

Note that the instruction register is not shown in Fig. 4.1 because it cannot be accessed by the programmer.

4.1.4.2 ALU

The *arithmetic logic unit* (ALU) in the MSP430 performs all mathematical (add, subtract, increment, decrement) and logical (AND, XOR, invert, rotate, bit clear/set) operations. This system operates on data being held in CPU registers. The ALU contains logic to produce status bits (VNZC) that are latched into the status register for specific instructions. The ALU is a 16-bit circuit, so it only operates on the lower 16-bits of the CPU registers. Figure 4.4 gives a block diagram of the MSP430 ALU showing how the data is routed into and out of it.

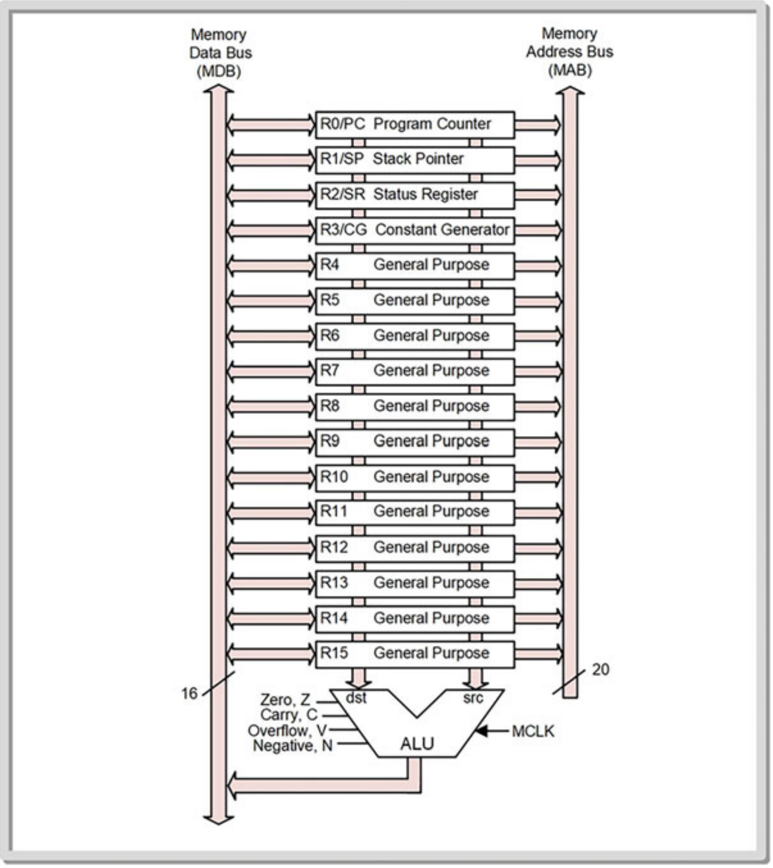


Fig. 4.4
MSP430 ALU overview

4.1.5 Input/Output Ports and Peripherals

What makes the MSP430 an MCU and not a standard computer is its built-in peripherals. The following subsections provide a brief introduction to some of the common peripherals used on the MSP430. This introduction is only meant to introduce these peripherals. In-depth coverage of using the peripherals will be covered later in the book. Figure 4.5 shows an expanded MSP430 functional block diagram highlighting the peripherals.

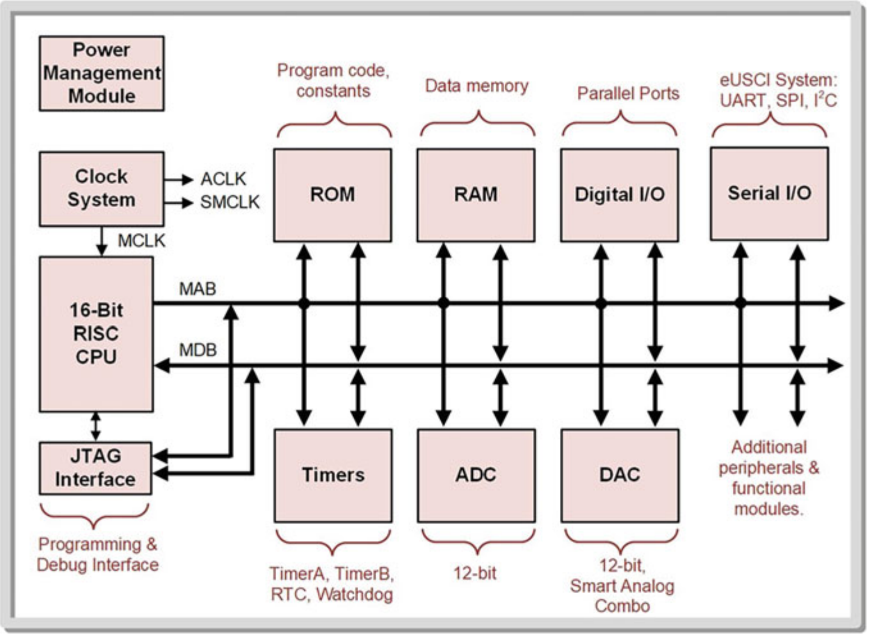


Fig. 4.5
MSP430 functional block diagram highlighting peripherals

4.1.5.1 Digital I/O

The MSP430 contains *digital I/O* that acts as parallel buses that interface with the outside world. Each of these I/O pins can be programmed to either be inputs or outputs and also contains an optional pull-up or pull-down resistor. The full MSP430 architecture has up to 12-, 8-bit I/O ports; however, very few MCU devices have these many I/O ports implemented.

4.1.5.2 Serial I/O

The MSP430 contains numerous serial communication peripherals that support multiple serial communication protocols. The MSP430 calls these modules the *enhanced Universal Serial Communication Interface (eUSCI)* modules. One flavor of these modules is labeled eUSCI_A and can be configured to implement either a *universal asynchronous receiver/transmitter* (UART) protocol or a *serial peripheral interface* (SPI) protocol. The other flavor of these modules is labeled eUSCI_B and can be configured to implement either a SPI protocol or an *inter-integrated circuit* (I^2C). An MSP430 MCU can contain multiple eUSCI_As and eUSCI_Bs. The naming convention in the TI documentation for the proliferation of these modules is eUSCI_Ax (i.e., eUSCI_A0, eUSCI_A1) and eUSCI_Bx (i.e., eUSCI_B0, eUSCI_B1).

4.1.5.3 Timers

The MSP430 also contains numerous *timer* peripherals. A timer is a binary counter that runs independent from the CPU and can be used to track or generate events based on its value. Timers allow the selection of various clock sources to control how fast the counter runs. Events can be generated when the counter reaches certain values (compare mode) and when the counter overflows. The counters can also store their current value when external signals are asserted (capture mode). Events that can be triggered range from altering the program execution to performing a full system reset.

The MSP430 contains four specific types of timer modules called Timer_A, Timer_B, Real-Time Clock (RTC) counter, and a Watchdog counter. An MCU can have multiple Timer_A and Timer_B modules. The naming convention in the TI documentation for the proliferation of these modules is Timer_Ax (i.e., Timer_A0, Timer_A1) and Timer_Bx (i.e., Timer_B0, Timer_B1).

4.1.5.4 Analog-to-Digital Converter

The MSP430 also contains an *analog-to-digital converter* (ADCs) module. An ADC takes in an analog signal on a pin of the device and produces a binary equivalent value of the voltage level. Each time the digital value is generated, it is called a *sample*. These samples can be used to alter the execution of code. By accumulating a series of samples, the MCU can recreate a digital representation of the analog signal over a period of time and then perform signal conditioning to the sample set. The MSP430 allows multiple input pins to be fed into a single ADC using a selectable switch. This allows the ADC to perform conversions on multiple inputs. The MSP430 ADC can support up to 12 bits of resolution for the conversion and can route up to 16 input channels to the ADC using a switch circuit.

4.1.5.5 Digital-to-Analog Converters

Some versions of the MSP430 also contain *digital-to-analog converters* (DACs). As the name suggests, these circuits take a binary code and generate an output voltage on one of the pins of the device. The DACs are usually fast enough to produce output signals for audio and video applications. The MSP430 puts all of its DAC circuits within a module named the *smart analog combo (SAC)*. The SACs contain additional circuitry to support creating analog voltages such as operational amplifiers and switch arrays. The MSP430's DACs support 12-bit resolution.

4.1.5.6 Clock System

The MSP430 clock system (CS) module generates and distributes the various clock sources used on MCU. The CS module supports takes in both internal and external clock sources and then creates on-chip clocks for use by the MCU and peripheral. The three primary clocks that the CS module produces are MCLK (master clock), SMCLK (subsystem master clock), and ACLK (auxiliary clock). MCLK is the clock source for the CPU. SMCLK is the clock for the peripherals that can work independently from CPU operation. ACLK can be used for peripherals that require low-frequency operation. ACLK is typically set to 32.768 kHz.

4.1.5.7 Power Management Module

The MSP430 power management module (PMM). The PMM's primary function is to generate a power supply voltage for the core logic on-chip. Its secondary function is to provide power supply monitoring capabilities. The supervising capabilities help guide actions that need to be taken when the power supply begins to drop.

4.1.6 Bus System

This MSP430 CPU has a 16-bit memory data bus and a 20-bit memory address bus. The 16-bit MDB allows information to be moved between memory and the CPU in 16-bit words. The 20-bit MAB allows the CPU to access up to $2^{20} = 1{,}048{,}576$ unique address locations. All I/O, peripherals, data memory, and program memory are assigned to a unique address within the MSP430's unified memory map.

4.1.7 MSP430 Part Numbering

The specific part numbers that make up the MSP430 family vary in the amount of memory they contain, the type of memory used, the clock frequency, operating temperature range, packaging type, and the testing that the device undergoes. All of the possible combinations of options within the MSP430 family results in thousands of different part numbers. This is one of the most challenging concepts to grasp as a student when first starting to work with modern MCUs. Making the concept even more difficult is that the documentation for the MSP430 exists at a variety of levels. There is a user's guide for the full functionality of the MSP430 family, yet very few, if any, part numbers will have all of the functionality described in the guide. There is also documentation on how to program the MSP430, but the programmer must be aware that the code examples might be for some functionality that isn't on their own device. Additionally, when learning an MCU, the device needs to be loaded onto a development board. The development boards will have a variety of built-in features such as LEDs and switches. This means the programmer needs to be aware of what actual I/O is available to use when designing programs. Figure 4.6 provides a key for how the specific device part numbers are created within the MSP430 family.

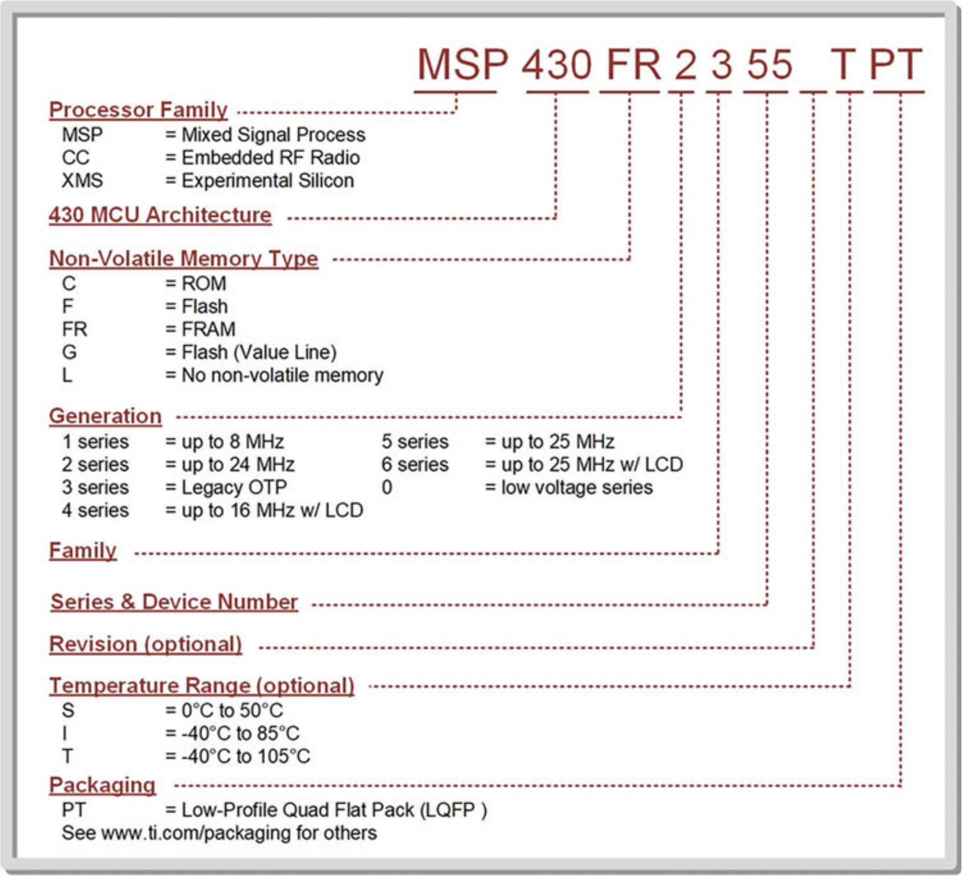


Fig. 4.6
MSP430 part numbering scheme

Concept Check

CC4.1 Why don't they just make a single MSP430 MCU that contains all of the possible functionality instead of having thousands of different part numbers?

A) Not all applications require all of the functionality in the MSP430 architecture. By creating smaller versions of the MSP430, the MCUs can be customized for the application and be lower in power and cost.

B) It is too difficult to get everything in the MSP430 working at the same time.

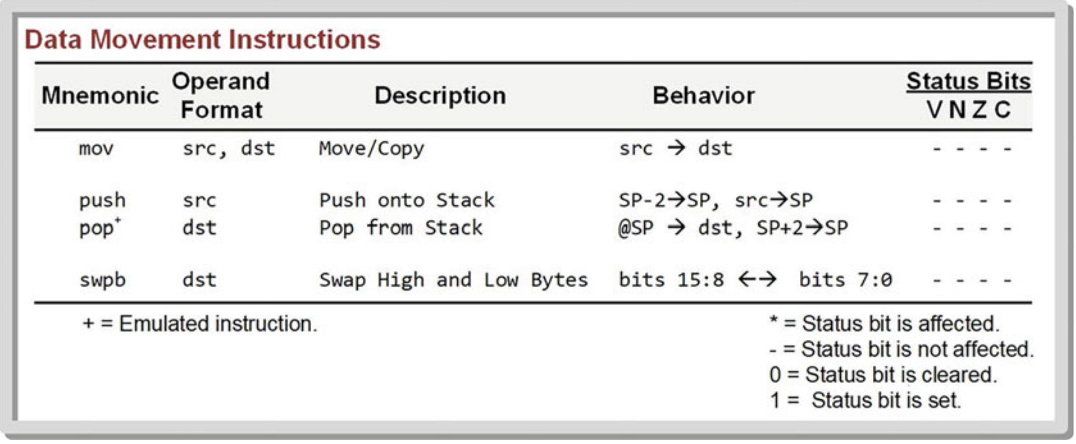
C) The full MSP430 MCU is too expensive to be practical.

D) They can't find a package large enough to fit everything in the MSP430.

4.2 MSP430 Software Overview

4.2.1 The MSP430 Instruction Set

The MSP430's instruction set contains 27 core instructions plus 24 *emulated* instructions. Emulated instructions are instructions that make code easier to write and read, but do not have op-codes themselves. Emulated instructions have unique mnemonics and are used when programming in assembly. Emulated instructions are replaced automatically by the assembler with a core instruction. There is no performance penalty when using emulated instructions because the assembler replaces them with an equivalent core instruction of the same size and execution requirements. Tables 4.1, 4.2, and 4.3 give the 51 instructions for the MSP430 family grouped by their class of instruction (i.e., data movement, data manipulation, and program flow).



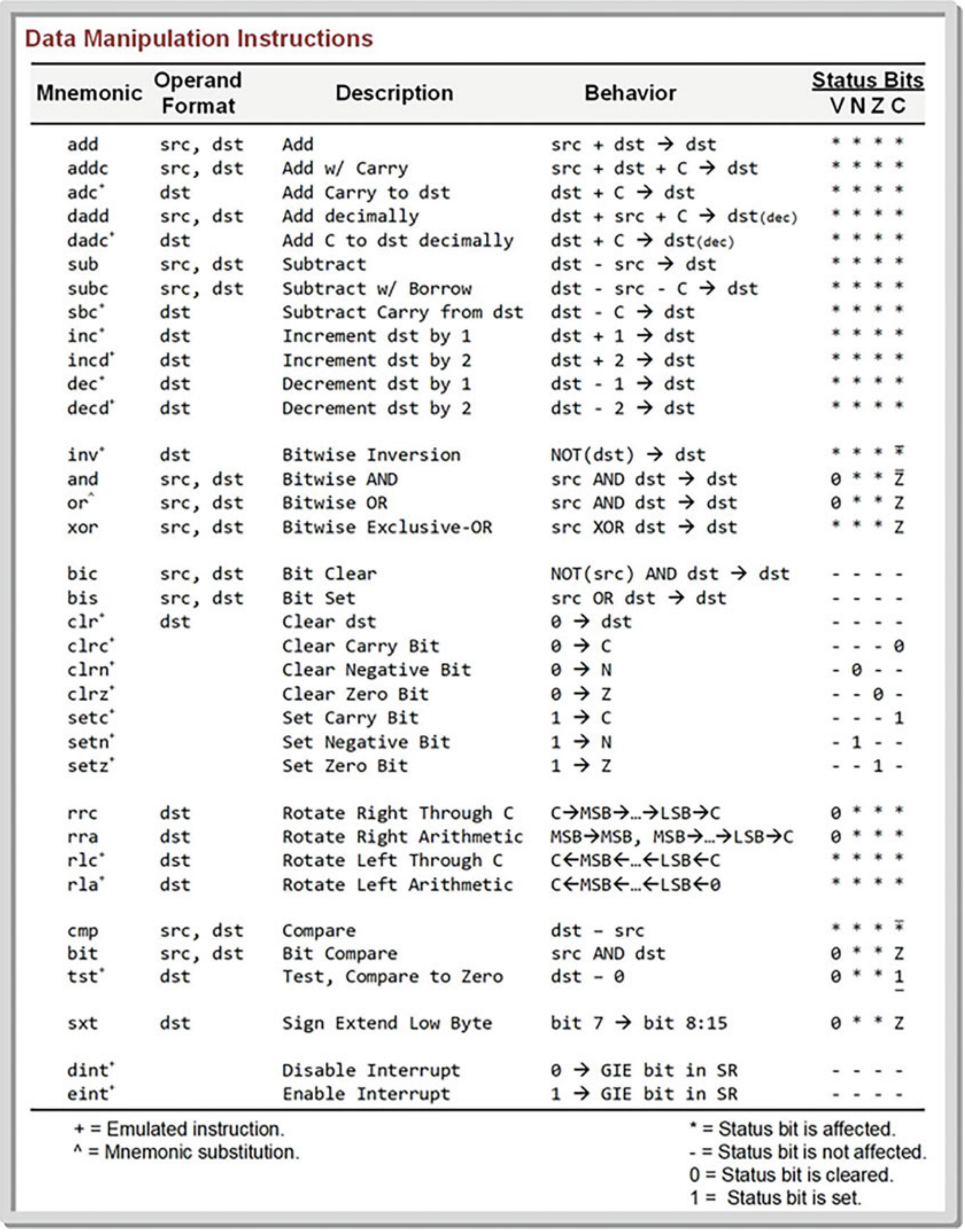
Data Movement Instructions

Mnemonic	Operand Format	Description	Behavior	Status Bits V N Z C
mov	src, dst	Move/Copy	src → dst	- - - -
push	src	Push onto Stack	SP-2→SP, src→SP	- - - -
pop+	dst	Pop from Stack	@SP → dst, SP+2→SP	- - - -
swpb	dst	Swap High and Low Bytes	bits 15:8 ←→ bits 7:0	- - - -

+ = Emulated instruction.

* = Status bit is affected.
- = Status bit is not affected.
0 = Status bit is cleared.
1 = Status bit is set.

Table 4.1
MSP430 data movement instructions



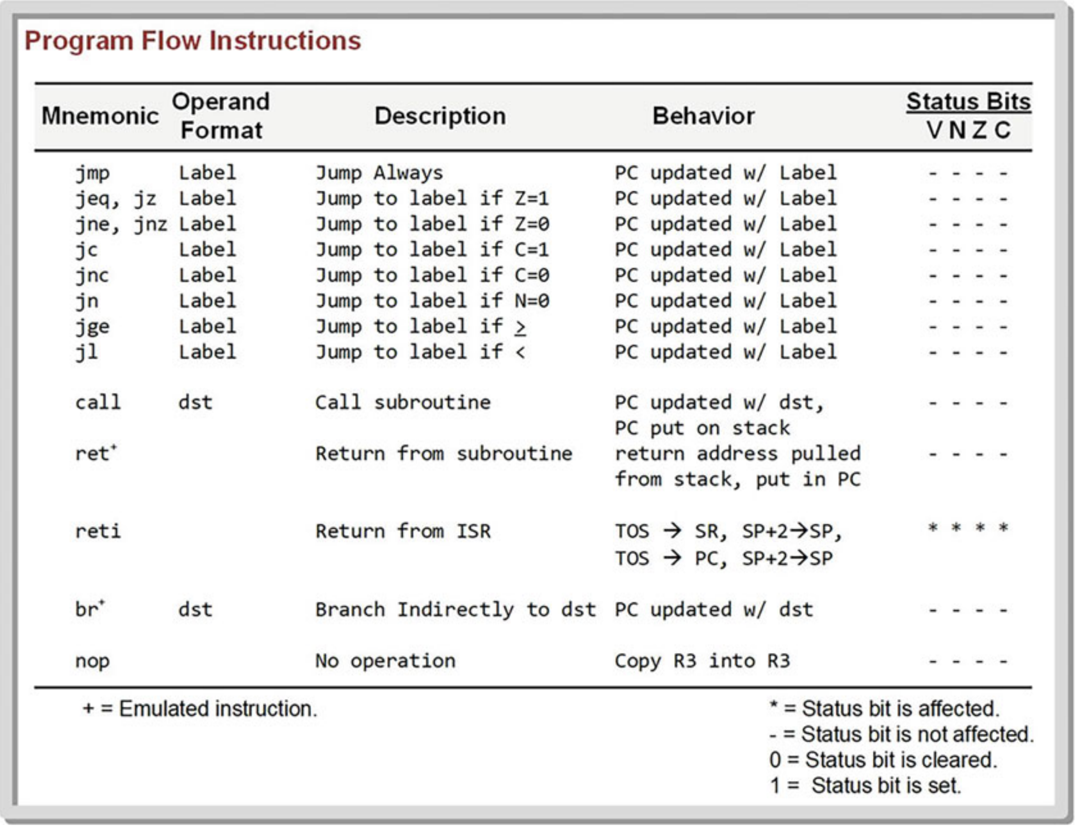
Data Manipulation Instructions

Mnemonic	Operand Format	Description	Behavior	Status Bits			
				V	N	Z	C
add	src, dst	Add	src + dst → dst	*	*	*	*
addc	src, dst	Add w/ Carry	src + dst + C → dst	*	*	*	*
adc+	dst	Add Carry to dst	dst + C → dst	*	*	*	*
dadd	src, dst	Add decimally	dst + src + C → dst(dec)	*	*	*	*
dadc+	dst	Add C to dst decimally	dst + C → dst(dec)	*	*	*	*
sub	src, dst	Subtract	dst - src → dst	*	*	*	*
subc	src, dst	Subtract w/ Borrow	dst - src - C → dst	*	*	*	*
sbc+	dst	Subtract Carry from dst	dst - C → dst	*	*	*	*
inc+	dst	Increment dst by 1	dst + 1 → dst	*	*	*	*
incd+	dst	Increment dst by 2	dst + 2 → dst	*	*	*	*
dec+	dst	Decrement dst by 1	dst - 1 → dst	*	*	*	*
decd+	dst	Decrement dst by 2	dst - 2 → dst	*	*	*	*
inv+	dst	Bitwise Inversion	NOT(dst) → dst	*	*	*	$\overline{*}$
and	src, dst	Bitwise AND	src AND dst → dst	0	*	*	$\overline{Z}$
or^	src, dst	Bitwise OR	src AND dst → dst	0	*	*	Z
xor	src, dst	Bitwise Exclusive-OR	src XOR dst → dst	*	*	*	Z
bic	src, dst	Bit Clear	NOT(src) AND dst → dst	-	-	-	-
bis	src, dst	Bit Set	src OR dst → dst	-	-	-	-
clr+	dst	Clear dst	0 → dst	-	-	-	-
clrc+		Clear Carry Bit	0 → C	-	-	-	0
clrn+		Clear Negative Bit	0 → N	-	0	-	-
clrz+		Clear Zero Bit	0 → Z	-	-	0	-
setc+		Set Carry Bit	1 → C	-	-	-	1
setn+		Set Negative Bit	1 → N	-	1	-	-
setz+		Set Zero Bit	1 → Z	-	-	1	-
rrc	dst	Rotate Right Through C	C→MSB→…→LSB→C	0	*	*	*
rra	dst	Rotate Right Arithmetic	MSB→MSB, MSB→…→LSB→C	0	*	*	*
rlc+	dst	Rotate Left Through C	C←MSB←…←LSB←C	*	*	*	*
rla+	dst	Rotate Left Arithmetic	C←MSB←…←LSB←0	*	*	*	*
cmp	src, dst	Compare	dst – src	*	*	*	$\overline{*}$
bit	src, dst	Bit Compare	src AND dst	0	*	*	Z
tst+	dst	Test, Compare to Zero	dst – 0	0	*	*	$\underline{1}$
sxt	dst	Sign Extend Low Byte	bit 7 → bit 8:15	0	*	*	Z
dint+		Disable Interrupt	0 → GIE bit in SR	-	-	-	-
eint+		Enable Interrupt	1 → GIE bit in SR	-	-	-	-

+ = Emulated instruction.
^ = Mnemonic substitution.

* = Status bit is affected.
- = Status bit is not affected.
0 = Status bit is cleared.
1 = Status bit is set.

Table 4.2
MSP430 data manipulation instructions



Program Flow Instructions

Mnemonic	Operand Format	Description	Behavior	Status Bits V N Z C
jmp	Label	Jump Always	PC updated w/ Label	- - - -
jeq, jz	Label	Jump to label if Z=1	PC updated w/ Label	- - - -
jne, jnz	Label	Jump to label if Z=0	PC updated w/ Label	- - - -
jc	Label	Jump to label if C=1	PC updated w/ Label	- - - -
jnc	Label	Jump to label if C=0	PC updated w/ Label	- - - -
jn	Label	Jump to label if N=0	PC updated w/ Label	- - - -
jge	Label	Jump to label if ≥	PC updated w/ Label	- - - -
jl	Label	Jump to label if <	PC updated w/ Label	- - - -
call	dst	Call subroutine	PC updated w/ dst, PC put on stack	- - - -
ret+		Return from subroutine	return address pulled from stack, put in PC	- - - -
reti		Return from ISR	TOS → SR, SP+2→SP, TOS → PC, SP+2→SP	* * * *
br+	dst	Branch Indirectly to dst	PC updated w/ dst	- - - -
nop		No operation	Copy R3 into R3	- - - -

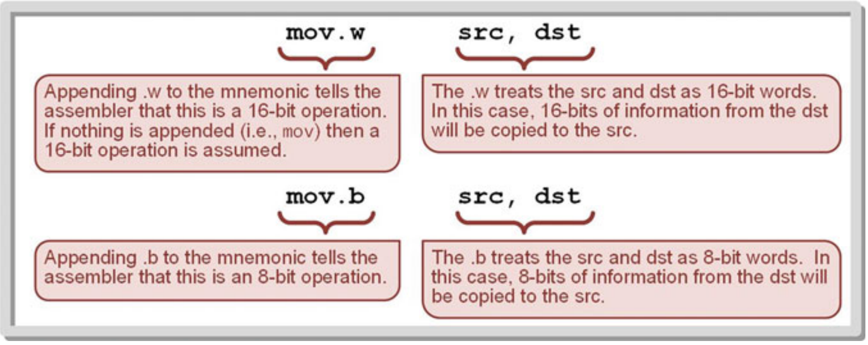
+ = Emulated instruction.

* = Status bit is affected.
- = Status bit is not affected.
0 = Status bit is cleared.
1 = Status bit is set.

Table 4.3
MSP430 program flow instructions

4.2.2 Word (.W) Versus Byte (.B) Operations

Many MSP430 instructions have the ability to perform their operation on either 16-bit words or 8-bit bytes. If the instruction mnemonic is used as is, the instruction is interpreted by the assembler to be a 16-bit operation. Special syntax can be appended to the instruction to explicitly state whether the operation is 16-bit or 8-bit. To specify a 16-bit operation, a .w is appended to the instruction mnemonic. To specify an 8-bit operation, a .b is appended to the instruction mnemonic. Figure 4.7 shows how the .w and .b syntax are used. Recall from Fig. 4.2 that 16-bit words are aligned to even address, so 16-bit access to odd addresses should be avoided.



Appending .w to the mnemonic tells the assembler that this is a 16-bit operation. If nothing is appended (i.e., mov) then a 16-bit operation is assumed.

The .w treats the src and dst as 16-bit words. In this case, 16-bits of information from the dst will be copied to the src.

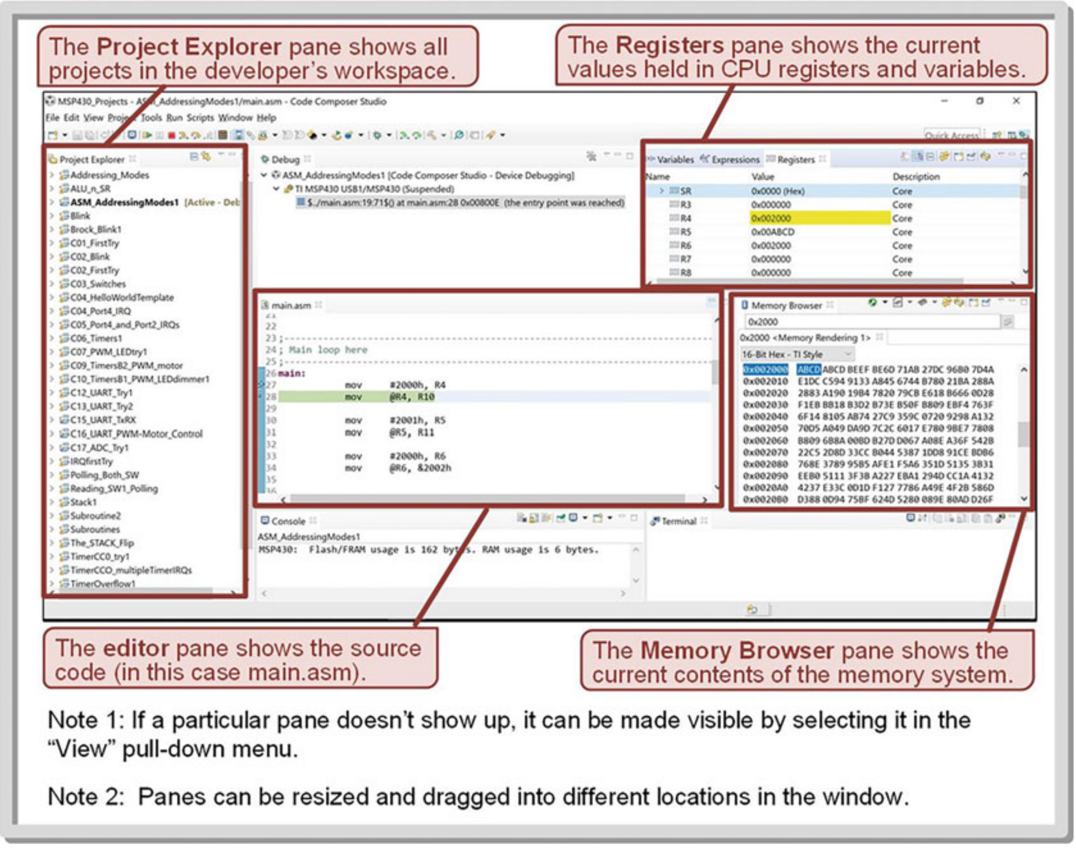
Appending .b to the mnemonic tells the assembler that this is an 8-bit operation.

The .b treats the src and dst as 8-bit words. In this case, 8-bits of information from the dst will be copied to the src.

Fig. 4.7
Word versus byte operation syntax

4.2.3 The TI Code Composer Studio Development Environment

TI provides a free development environment for MSP430 MCUs called the *Code Computer Student (CCS) Integrated Development Environment (IDE)*. The *integrated* term refers to the tool containing one environment that can be used to compile, assemble, link, download, and debug a program on an MCU. This text is using CCS version 9.x. Individual features of CCS will be covered in more details as we begin developing operational programs. Figure 4.8 gives an overview of some of the development features of CCS.



The **Project Explorer** pane shows all projects in the developer's workspace.

The **Registers** pane shows the current values held in CPU registers and variables.

MSP430_Projects - ASM_AddressingModes1/main.asm - Code Composer Studio
File Edit View Project Tools Run Scripts Window Help
Quick Access
Project Explorer
Debug
Variables
Expressions
Registers
main.asm
Memory Browser
Console
Terminal

Addressing_Modes
ALU_n_SR
ASM_AddressingModes1 [Active - Deb
Blink
Brock_Blink1
C01_FirstTry
C02_Blink
C02_FirstTry
C03_Switches
C04_HelloWorldTemplate
C04_Port4_IRQ
C05_Port4_and_Port2_IRQs
C06_Timers1
C07_PWM_LEDtry1
C09_TimersB2_PWM_motor
C10_TimersB1_PWM_LEDdimmer1
C12_UART_Try1
C13_UART_Try2
C15_UART_TxRX
C16_UART_PWM-Motor_Control
C17_ADC_Try1
IRQfirstTry
Polling_Both_SW
Reading_SW1_Polling
Stack1
Subroutine2
Subroutines
The_STACK_Flip
TimerCC0_try1
TimerCCO_multipleTimerIRQs
TimerOverflow1

```
ASM_AddressingModes1 [Code Composer Studio - Device Debugging]
  TI MSP430 USB1/MSP430 (Suspended)
    $../main.asm:19:71$() at main.asm:28 0x00800E  (the entry point was reached)
```

Name	Value	Description
SR	0x0000 (Hex)	Core
R3	0x000000	Core
R4	0x002000	Core
R5	0x00ABCD	Core
R6	0x002000	Core
R7	0x000000	Core
R8	0x000000	Core

```

;----------------------------------------------------------------------
; Main loop here
;----------------------------------------------------------------------
main:
            mov     #2000h, R4
            mov     @R4, R10

            mov     #2001h, R5
            mov     @R5, R11

            mov     #2000h, R6
            mov     @R6, &2002h

```

0x2000

0x2000 <Memory Rendering 1>

16-Bit Hex - TI Style

```
0x002000  ABCD ABCD BEEF BE6D 71AB 27DC 9680 7D4A
0x002010  E1DC C594 9133 A845 6744 B780 21BA 288A
0x002020  2883 A190 19B4 7820 79CB E618 B666 0D28
0x002030  F1EB BB18 B3D2 B73E B50F B809 EBF4 763F
0x002040  6F14 8105 AB74 27C9 359C 0720 9298 A132
0x002050  70D5 A049 DA9D 7C2C 6017 E780 9BE7 7808
0x002060  B809 688A 00BD B27D D067 A08E A36F 542B
0x002070  22C5 2D8D 33CC B044 5387 1DD8 91CE BDB6
0x002080  768E 3789 95B5 AFE1 F5A6 351D 5135 3B31
0x002090  EEB0 5111 3F38 A227 EBA1 294D CC1A 4132
0x0020A0  4237 E33C 0D1D F127 7786 A49E 4F2B 586D
0x0020B0  D388 0D94 75BF 624D 5280 089E 80AD D26F
```

```
ASM_AddressingModes1
MSP430:  Flash/FRAM usage is 162 bytes. RAM usage is 6 bytes.
```

The **editor** pane shows the source code (in this case main.asm).

The **Memory Browser** pane shows the current contents of the memory system.

Note 1: If a particular pane doesn't show up, it can be made visible by selecting it in the "View" pull-down menu.

Note 2: Panes can be resized and dragged into different locations in the window.

Fig. 4.8
TI code composer studio's development environment

Concept Check

CC4.2 If I am programming the MSP430 in assembly, do I need to care whether or not I am using core versus emulated instructions?

A) Yes. Since emulated instructions don't have op-codes, some emulated instructions may not be interpreted correctly by the assembler.

B) Yes. The emulated instructions are fake instructions that are omitted by the assembler. They are only used for readability similar to inserting comments in your code.

C) Yes. You will suffer a performance penalty when using emulated instructions because you never know what core instructions the assembler is going to use to replace them.

D) No. There is no performance penalty when using emulated instructions. They are simply used to make the assembly program easier to read.

4.3 MSP430FR2355 LaunchPad™ Development Kit

The development board that will be used throughout the rest of this book to learn about MCUs is the *MSP430FR2355 LaunchPad™ Development Kit* from Texas Instruments. This board contains an MSP430FR2355TPT MCU in addition to a variety of extra circuitry to facilitate programming, debugging, and providing power to the MCU. The board also contains some LEDs, buttons, pin headers, and additional circuitry to provide an interface to a select number of peripherals on the MCU. All code examples provided in the rest of this book are intended to be directly coded in CCS and downloaded to this development board. Figure 4.9 gives an overview of the MSP430FR2355 LaunchPad™ Development Kit.

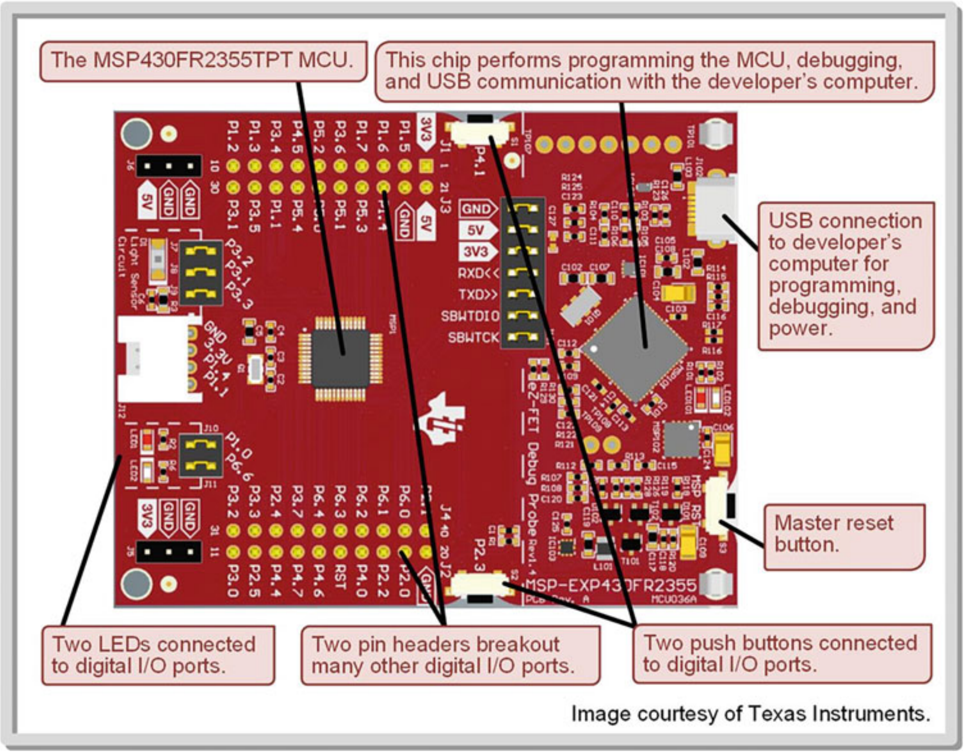

The MSP430FR2355TPT MCU.
This chip performs programming the MCU, debugging, and USB communication with the developer's computer.
USB connection to developer's computer for programming, debugging, and power.
Master reset button.
Two LEDs connected to digital I/O ports.
Two pin headers breakout many other digital I/O ports.
Two push buttons connected to digital I/O ports.

Image courtesy of Texas Instruments.

Fig. 4.9
Overview of the MSP430FR2355 LaunchPadTM Development Kit

The MSP430FR2355TPT CPU is clocked off of an internal 1 MHz MCLK by default. The clock system also produces 1 MHz SMCLK for use by the peripherals. The LaunchPad™ board contains a 32.768 kHz crystal oscillator that is fed into the MCU in order to provide the ACLK. These values represent the default frequencies upon reset and can be configured differently if so desired. The MCU memory system is 64 kB, which means it only uses an address range from 0000h to FFFFh (i.e., no 20-bit addresses). The MCU contains 32kB of FRAM program memory. It contains 4k of SRAM plus 512 bytes of FRAM for data memory. It contains six digital I/O ports labeled P1 (8-bit), P2 (8-bit), P3 (8-bit), P4 (8-bit), P5 (5-bit), and P6 (7-bit). For 16-bit port operations, the labels PA, PB, and PC are used where PA = P1:P2, PB = P3:P4, and PC = P5:P6. Each of these digital I/O bits is brought out to pins on the package and can be configured as either inputs or outputs. This MCU contains four serial communication blocks called eUSCI_A0, eUSCI_A1, eUSCI_B0, and eUSCI_B1. The “A” versions of the eUSCIs can be configured into either UART or SPI mode, while the “B” versions can be configured into either SPI or I^{2}C mode. This MCU contains four separate timers called Timer_B0, Timer_B1, Timer_B2, and Timer_B3. The clock of each timer is selectable between MCLK, SMCLK, ACLK, and an external input. Additionally, each of the timer clocks can be divided down to achieve numerous slower counting frequencies. The timer peripherals also include a 16-bit RTC and a watchdog. A single 12-bit ADC is included that can convert up to 12-input channels selected by a switch. Also, there are four smart analog combos that each contain a 12-bit DAC. These are the primary peripherals that will be discussed in this book. Additional peripherals on the MCU include comparators, backup memory, an interrupt controller, a 16-bit cyclic redundancy check (CRC) error correction module, an LCD controller, and an infrared sensor management system.

It would be impractical to have a dedicated pin for each and every peripheral signal within the MCU as too many pins would be needed on the package. Simultaneously, not all MCU applications will require every peripheral signal at the same time. Thus, to reduce the cost of the MCU, multiple internal peripheral signals are assigned to the same pin, and then the user configures the pin to be used as whichever peripheral is desired. This approach does limit some functionality because it is possible that two peripheral signals may be assigned to the same package pin internally that are both needed by the user. In this case, larger MCU is typically selected that has both desired peripheral signals on separate pins. Figure 4.10 shows a detailed block diagram for the MSP430FR2355TPT MCU used on the MSP430FR2355 LaunchPad™ Development Kit.

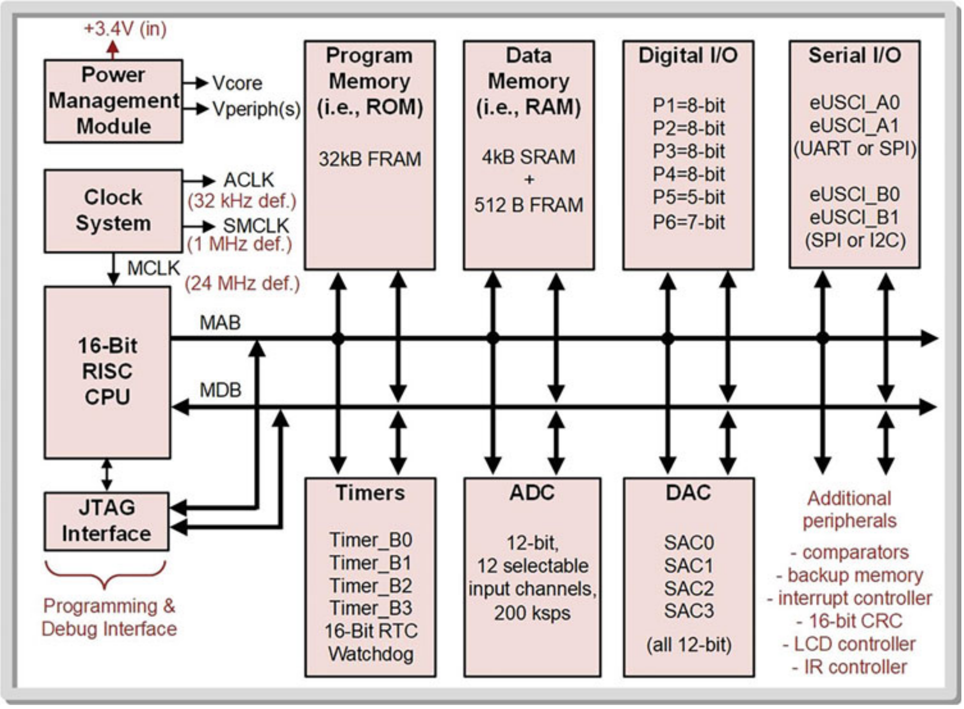


Fig. 4.10
Specific capabilities of MSP430FR2355TPT

Everything within the MSPFR2355TPT is assigned a unique address within its 64 kB address space. Figure 4.11 shows the memory map for this MCU.

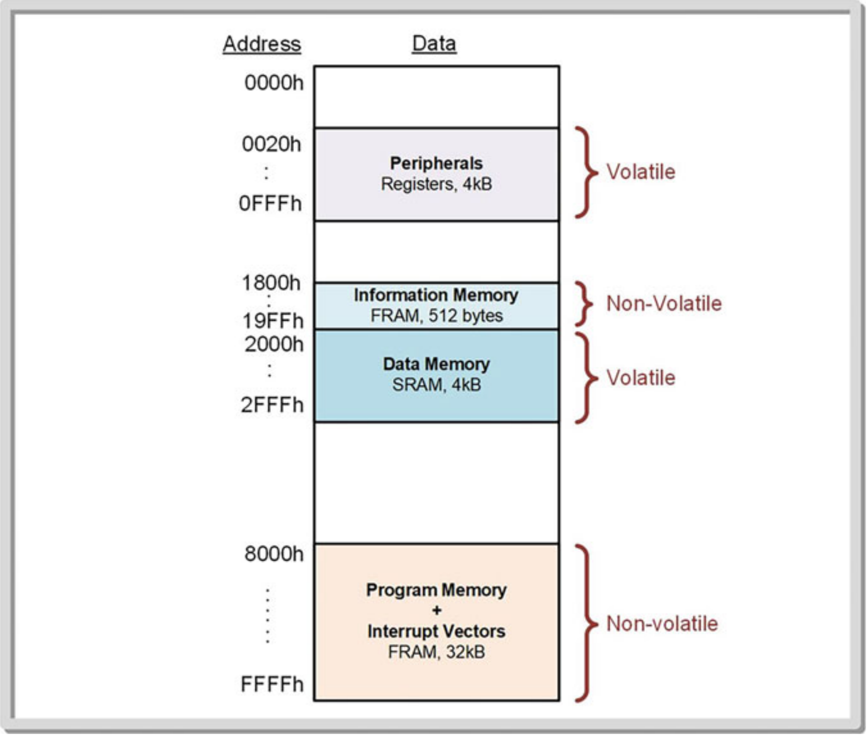


Fig. 4.11
MSP430FR235 memory map

Concept Check

CC4.3 Why does the MSP430FR2355TPT MCU share package pins across multiple internal peripheral signals?

A) It is impractical to assign each peripheral signal to a dedicated package pin because the package would be too large.

B) Not all applications require all peripherals, so the pin sharing approach reduces cost.

C) They must have run out of time when designing the packages.

D) A & B.

Summary

- The MSP430 is a family of MCUs designed for low-cost, low-power operation.
- The MSP430 has a general architecture that includes a CPU, program memory, data memory, and abundant peripherals all integrated on a single chip.
- Individual MCU devices only contain a subset of the functionality of the larger MSP430 architecture. This allows smaller versions of the architecture to be implemented that are tailored for the application and reduce the cost of the MCU.
- The MSP430's program memory ranges in size from 0 to 512 kB and can be implemented in MROM, Flash, or FRAM technologies.
- The MSP430's data memory ranges in size from 125 bytes to 66 kB and is implemented primarily with SRAM technology, although some MCUs offer a small amount of additional FRAM for data memory.
- The MSP430 is based on a 16-bit RISC CPU.
- The CPU contains 16 registers that are normally operated on 16 bits at a time; however, they are technically 20-bits wide to support addressing a 1M memory system on some MCUs. The registers are named R0→R15. Four of the registers are special-purpose registers that are used as the program counter (R0 = PC), stack pointer (R1 = SP), status register (R2 = SR), and a constant generator (R3 = CG). The remaining 12 registers are general-purpose (R4→R15).
- Data in the CPU is manipulated either 16 bits or 8 bits at a time.
- The MSP430 ALU has functionality to perform addition, subtraction, increments, decrements, bit clears, bit sets, ANDs, INVs, XOR, and rotates.
- The MSP430 ALU produces four flags that are stored in the status register. These are two's complement overflow (V), negative (N), zero (Z), and carry (C).
- The MSP430 contains digital I/O peripherals to communicate parallel data to the outside world. Each digital I/O bit can be configured as either an input or output with an optional pull-up/down resistor.
- The MSP430 contains enhanced Universal Serial Communication Interface (eUSCI) modules. These modules handle producing the protocols to transmit and receive data serially over a single pin. These modules support UART, SPI, and I^2C protocols.
- The MSP430 contains a large number of timers. A timer is a binary counter that runs independent of the CPU. The timer can trigger events when it reaches certain values or when it overflows. Timers can also store their values based on input signals. The clock frequency that drives the counters can be chosen from a variety of courses and can also be divided down to achieve a slower counter rate.
- The MSP430 contains a 12-bit analog-to-digital converter. This ADC can perform conversions on up to 12 input channels that are switched in one at a time.
- The MSP430 clock system creates all of the internal clocks used by the MCU. Some of the clocks come from off-chip oscillators, while some are produced on-chip. The three primary clocks that the CS module produces for internal use are MCLK (master clock), SMCLK (subsystem master clock), and ACLK (auxiliary clock).
- The power management module of the MSP430 takes in an external power supply voltage and creates all of the internal voltages needed by the CPU and the peripherals.
- The bus system of the MSP430 contains a 16-bit memory data bus and a 20-bit memory address bus. The 20 address lines allow the CPU to access 2^{16} = 1,048,576 unique address locations.
- All memory and peripherals are mapped to a unique address in the MSP430's unified memory map.
- The MSP430 has 51 instructions in its set. This consists of 27 core instructions and 24 emulated instructions. Emulated instructions are instructions that make code easier to write and read, but do not have op-codes themselves. Emulated instructions are translated into core instructions by the assembler.
- The MSP430 instructions can perform either 16-bit or 8-bit operations. If the mnemonic is listed as is, the assembler assumes the instruction is a 16-bit operation. Alternatively, a .w can be appended to the instruction to explicitly state that the operation is 16-bit. If an 8-bit operation is desired, a .b is appended to the mnemonic.

- ❖ The TI Code Composer Student Integrated Development Environment is a software package that allows code development in either assembly or C, compilation and error correction, linking, downloading, and debugging.
- ❖ The MSP430FR2355 LaunchPad™ Development Kit is a board that contains a MSP430FR2355TPT MCU in addition to circuitry to facilitate debugging, power regulation, and access to the MCU's peripherals.
- ❖ The MSP430FR2355TPT MCU used on the LaunchPad™ board contains 32 kB of EEPROM for program memory and 4 kB of SRAM +512 B of FRAM for data memory. This MCU also provides access to six digital I/O ports, four eUSCI modules, six timer modules, an ADC, and four smart analog combos.
- ❖ The MSP430FR2355TPT MCU contains other peripherals that won't be discussed in this text including comparators, backup memory, an interrupt controller, a 16-bit cyclic redundancy check (CRC) error correction module, an LCD controller, and an infrared sensor management system.

Exercise Problems

Section 4.1: MSP430 Hardware Overview

4.1.1 What is the MSP430 MCU optimized for?

4.1.2 What is the range of program memory sizes within the MSP430 family?

4.1.3 What types of technology exist for program memory within the MSP430 family?

4.1.4 What is the range of data memory sizes within the MSP430 family?

4.1.5 What types of technology exist for data memory within the MSP430 family?

4.1.6 How many total CPU registers does the MSP430 CPU that can be accessed?

4.1.7 How many registers within the MSP430 CPU are special-purpose?

4.1.8 How many registers within the MSP430 CPU are general-purpose?

4.1.9 Name all of the special-purpose registers within the MSP430 CPU.

4.1.10 What four status bits are altered by the MSP430 ALU?

4.1.11 What functionality does the digital I/O peripherals provide within the MSP430?

4.1.12 What functionality does the serial I/O peripherals provide within the MSP430?

4.1.13 What functionality does the timer peripherals provide within the MSP430?

4.1.14 What functionality does the ADC peripheral provide within the MSP430?

4.1.15 What functionality does the DAC peripherals provide within the MSP430?

4.1.16 What functionality does the clock system provide within the MSP430?

4.1.17 What functionality does the power management module provide within the MSP430?

4.1.18 How many unique addresses can the MSP430 bus system access?

Section 4.2: MSP430 Software Overview

4.2.1 Is the MSP430 CPU a RISC or CISC architecture?

4.2.2 How many core instructions does the MSP430 CPU support?

4.2.3 How many emulated instructions does the MSP430 CPU support?

4.2.4 Does every instruction in the MSP430's instruction set alter the VNZC status bits?

4.2.5 What special syntax is appended to an instruction mnemonic to indicate a 16-bit operation?

4.2.6 What special syntax is appended to an instruction mnemonic to indicate an 8-bit operation?

4.2.7 If no special syntax is appended to the instruction mnemonic, is the instruction treated as a 16-bit or 8-bit operation?

4.2.8 Does Code Composer Studio support development in C, assembly, or both?

4.2.9 Does Code Composer Studio provide visibility into the values in CPU registers, memory, or both?

4.2.10 Can Code Composer Studio also download a compiled program?

4.2.11 Can Code Composer Studio also debug a downloaded program?

Section 4.3: MSP430FR2355 LaunchPad™ Development Kit

4.3.1 What is the size of program memory on the MSP430FR2355TPT MCU loaded on the LaunchPad?

4.3.2 Which technology is used for the program memory on the MSP430FR2355TPT MCU loaded on the LaunchPad?

4.3.3 What is the specific size of data memory on the MSP430FR2355TPT MCU loaded on the LaunchPad?

4.3.4 What two technologies are used for the data memory on the MSP430FR2355TPT MCU loaded on the LaunchPad?

4.3.5 How many digital I/O ports are provided on the MSP430FR2355TPT MCU loaded on the LaunchPad?

4.3.6 How many serial communication blocks are provided on the MSP430FR2355TPT MCU loaded on the LaunchPad?

4.3.7 Which two serial communication protocols are available within the “A” versions of the eUSCIs on the MSP430FR2355TPT MCU loaded on the LaunchPad?

4.3.8 Which two serial communication protocols are available within the “B” versions of the eUSCIs on the MSP430FR2355TPT MCU loaded on the LaunchPad?

4.3.9 How many separate timers are provided on the MSP430FR2355TPT MCU loaded on the LaunchPad?

4.3.10 How many ADCs are provided on the MSP430FR2355TPT MCU loaded on the LaunchPad?

4.3.11 How many input channels can a single MSP430FR2355TPT ADC perform conversions on?

4.3.12 How many smart analog combo blocks are provided on the MSP430FR2355TPT MCU loaded on the LaunchPad?

Chapter 5: Getting Started Programming the MSP430 in Assembly

This chapter will introduce the logistics of programming the MSP430 MCU in assembly and running programs on the MSP430FR2355 LaunchPad™ Development Kit [1–3]. It is expected that the reader has CCS installed on their workstation and has a MSP430FR2355 LaunchPad™ Development Kit. This chapter introduces the anatomy of an assembler source file and provides some simple code to blink an LED on the LaunchPad™. The debugger tools are then explored to allow the code to be paused, restarted, and to view the contents of registers and memory. This chapter is not intended to be an exhaustive coverage of all possible assembler capabilities, but rather a means to form a basic understanding of an assembly program and get your first program running.

Learning Outcomes—After completing this chapter you will be able to:

5.1 Describe the basic anatomy of an assembly program.
5.2 Run the CCS design tools to assemble, download, and run a program on the MSP430 LaunchPad™ Development Kit.
5.3 Perform basic functions using the debugger.

5.1 The Anatomy of an Assembly Program File

An assembly source code file will have an extension of *.asm. Each line in an MSP430 assembly source file can either be empty, an *instruction*, a *comment*, an *assembler directive*, or a *macro invocation*. A line that is not empty is called a *statement*.

5.1.1 Instruction Statements

An *instruction statement* is where the developer lists the instructions that make up the functionality of the program. Assembly language source statements can contain four ordered fields: address label, mnemonic, operand list, and comment.

Address labels are used to mark a point in the program that can be referenced by other instructions. Address labels are used in program flow instructions to support functionality such as looping (i.e., jumping to the beginning of a code segment to repeat its execution) and conditional execution of code (i.e., jumping over certain instructions based on status bits). Address labels are very useful because the programmer does not need to keep track of the specific address locations within memory that are to be used. Instead, a label is inserted in the program, and then the assembler tracks the exact address of that label during assembly and inserts the physical address in the operand field of any instruction that uses that label. Address labels are always listed starting in column 1 of the source file. Labels must be legal identifiers (see Sect. 5.1.3.1). A label can be followed by an optional colon (:). The colon is not treated as part of the label name when used in subsequent instructions. The colon does indicate to CCS that a label is being inserted, and CCS will apply color coding to it in order to make the source file more readable. Note that not all statement lines require a label, and a label can also exist on its own line. For instruction statements that don’t use a label, column 1 must start with a whitespace character.

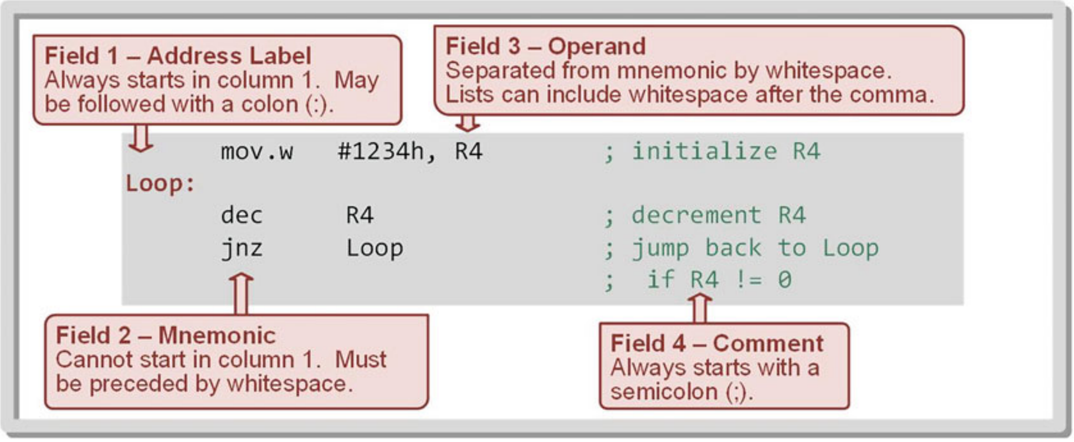
The mnemonic is the second field in the instruction statement. The mnemonic must be preceded by whitespace, either after the label or as the beginning of the line. All op-code mnemonics in Tables 4.1, 4.2, and 4.3 are supported.

B. J. LaMeres, *Embedded Systems Design using the MSP430FR2355 LaunchPad™*,
https://doi.org/10.1007/978-3-031-20888-1_5

The operand is the third field in the instruction statement (if applicable). The operand follows the mnemonic field, separated by whitespace. For operands that require a list (i.e., src, dst), the values are comma delimited with whitespace characters being optional after the comma (i.e., src,dst and src, dst are valid).

The comment is the fourth field of the instruction statement. All comments must start with a semicolon (;). Comments in the instruction field are optional but highly recommended for readability.

Figure 5.1 shows a breakdown of the four fields in an MSP430 instruction statement.



Field 1 – Address Label
Always starts in column 1. May be followed with a colon (:).

Field 3 – Operand
Separated from mnemonic by whitespace. Lists can include whitespace after the comma.

```
        mov.w   #1234h, R4          ; initialize R4
Loop:
        dec     R4                  ; decrement R4
        jnz     Loop                ; jump back to Loop
                                    ;  if R4 != 0
```

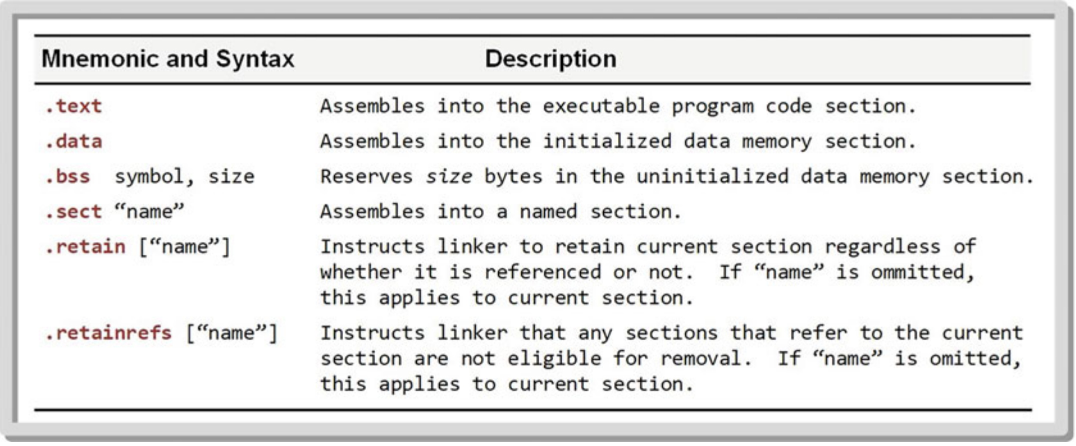
Field 2 – Mnemonic
Cannot start in column 1. Must be preceded by whitespace.

Field 4 – Comment
Always starts with a semicolon (;).

Fig. 5.1
Instrument statement fields

5.1.2 Assembler Directives

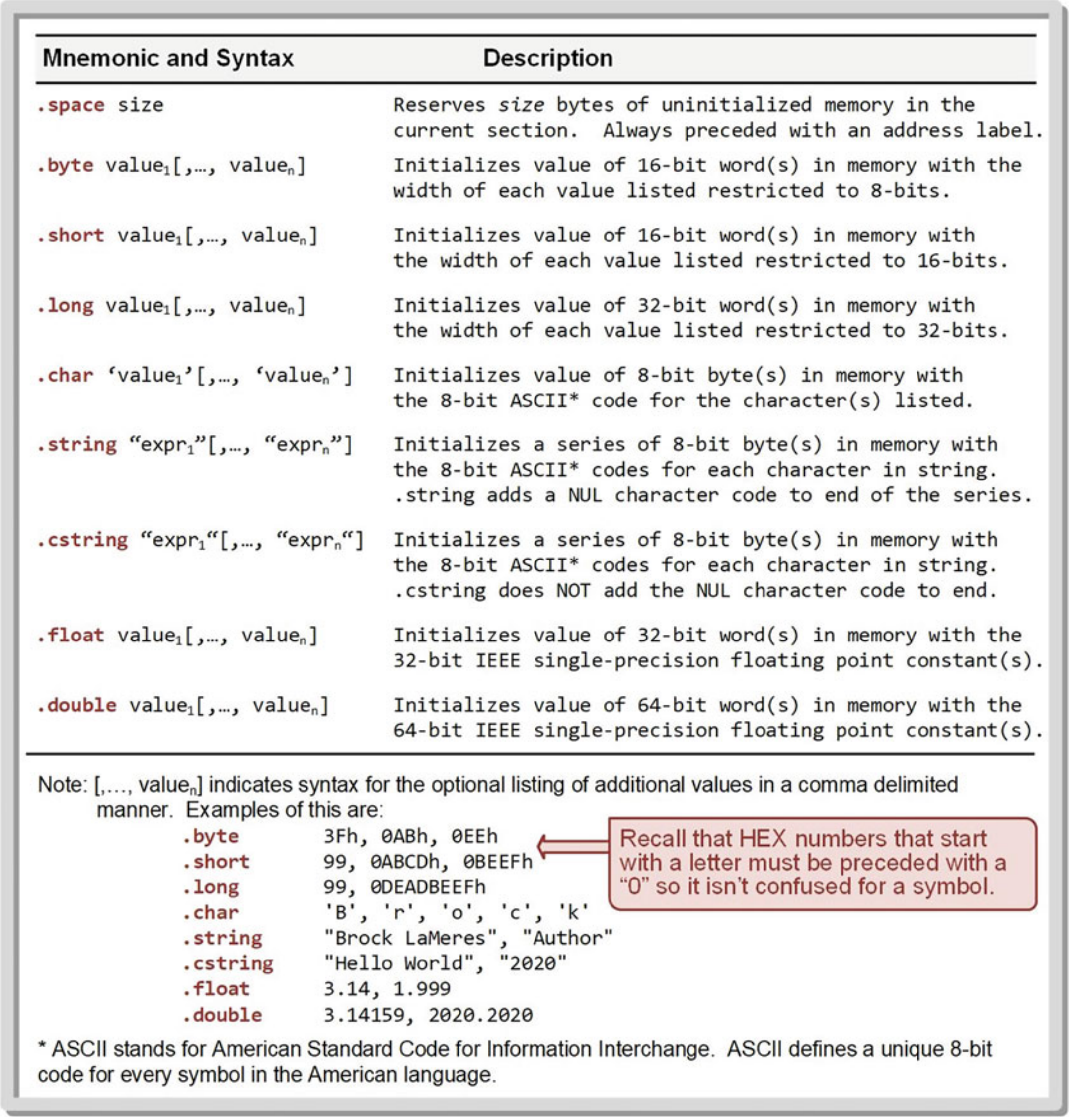
An assembler directive is a statement in the source file that tells the assembler information about the program but is not an actual instruction. Directives are used to locate instructions within program memory, allocate variable space in data memory, set up constants, and manage global variable access. A directive begins with a period (.) and must be preceded by whitespace. Directives typically are listed in the same column as the mnemonics field of instruction statements. Table 5.1 gives a list of directives that are used to control memory use. These directives are how an assembly program states that instructions go into program memory and variables go into data memory.



Mnemonic and Syntax	Description
.text	Assembles into the executable program code section.
.data	Assembles into the initialized data memory section.
.bss symbol, size	Reserves *size* bytes in the uninitialized data memory section.
.sect "name"	Assembles into a named section.
.retain ["name"]	Instructs linker to retain current section regardless of whether it is referenced or not. If "name" is ommitted, this applies to current section.
.retainrefs ["name"]	Instructs linker that any sections that refer to the current section are not eligible for removal. If "name" is omitted, this applies to current section.

Table 5.1
Directives that control memory use

Table 5.2 gives a list of directives that are used to reserve and/or initialize variables and constants in memory. These directives can be used in both data memory and program memory. Variables initialized in data memory are done so during download to the MCU and are then treated as R/W during program operation. Constants initialized in program memory are also done so during download to the MCU but are treated as ROM during program operation. Address labels are used with these directives to track the allocations memory.



Mnemonic and Syntax	Description
.space size	Reserves *size* bytes of uninitialized memory in the current section. Always preceded with an address label.
.byte value$_1$[,…, value$_n$]	Initializes value of 16-bit word(s) in memory with the width of each value listed restricted to 8-bits.
.short value$_1$[,…, value$_n$]	Initializes value of 16-bit word(s) in memory with the width of each value listed restricted to 16-bits.
.long value$_1$[,…, value$_n$]	Initializes value of 32-bit word(s) in memory with the width of each value listed restricted to 32-bits.
.char ‘value$_1$’[,…, ‘value$_n$’]	Initializes value of 8-bit byte(s) in memory with the 8-bit ASCII* code for the character(s) listed.
.string “expr$_1$”[,…, “expr$_n$”]	Initializes a series of 8-bit byte(s) in memory with the 8-bit ASCII* codes for each character in string. .string adds a NUL character code to end of the series.
.cstring “expr$_1$“[,…, “expr$_n$“]	Initializes a series of 8-bit byte(s) in memory with the 8-bit ASCII* codes for each character in string. .cstring does NOT add the NUL character code to end.
.float value$_1$[,…, value$_n$]	Initializes value of 32-bit word(s) in memory with the 32-bit IEEE single-precision floating point constant(s).
.double value$_1$[,…, value$_n$]	Initializes value of 64-bit word(s) in memory with the 64-bit IEEE single-precision floating point constant(s).

Note: [,…, value$_n$] indicates syntax for the optional listing of additional values in a comma delimited manner. Examples of this are:

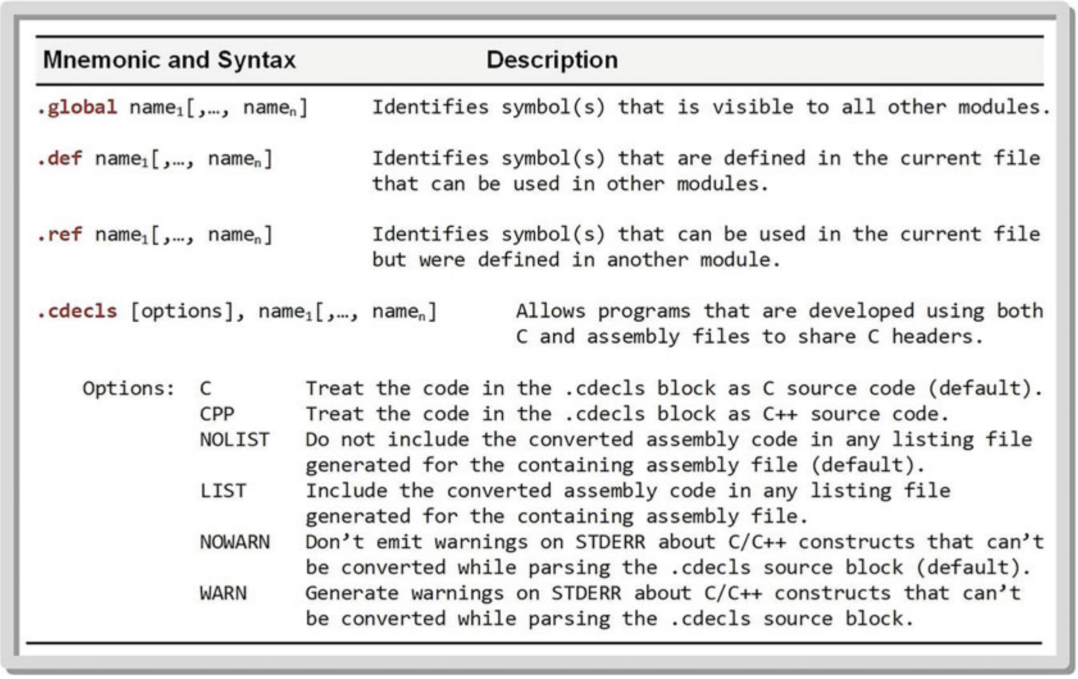
```
.byte       3Fh, 0ABh, 0EEh
.short      99, 0ABCDh, 0BEEFh
.long       99, 0DEADBEEFh
.char       'B', 'r', 'o', 'c', 'k'
.string     "Brock LaMeres", "Author"
.cstring    "Hello World", "2020"
.float      3.14, 1.999
.double     3.14159, 2020.2020
```

Recall that HEX numbers that start with a letter must be preceded with a “0” so it isn't confused for a symbol.

* ASCII stands for American Standard Code for Information Interchange. ASCII defines a unique 8-bit code for every symbol in the American language.

Table 5.2
Directives that reserve and/or initialize locations in memory (data and memory)

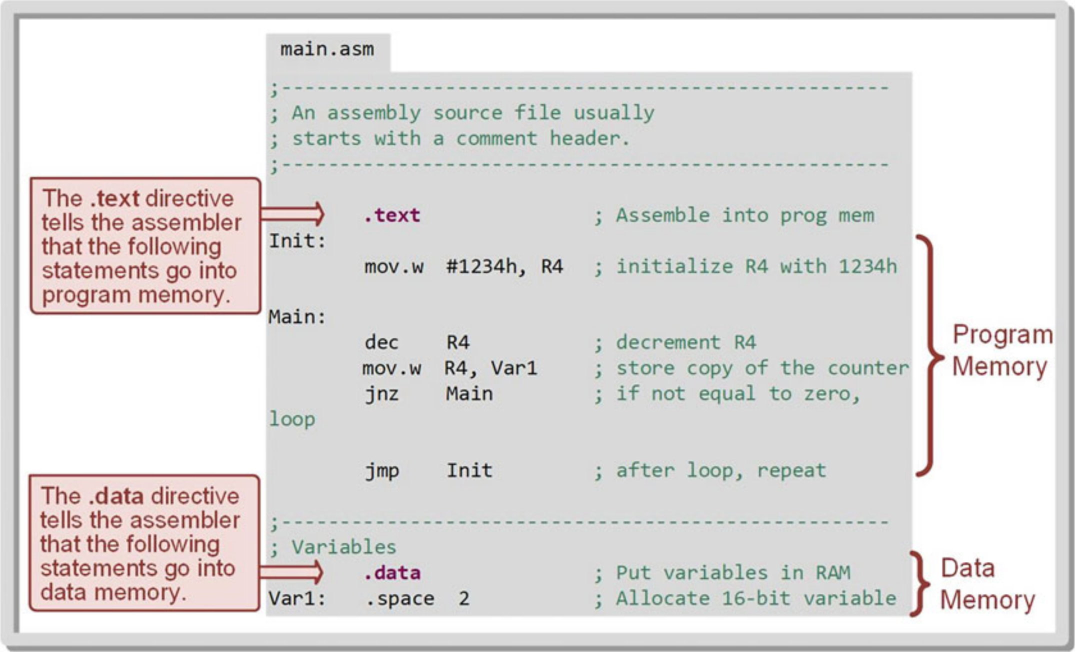
Table 5.3 gives a list of directives that are used to control interchange of information between multiple design files within a CCS project. These include directives to pass labels back and forth between files, making variables globally visible and allowing C and assembly files to work together in a mixed-language development environment.



Mnemonic and Syntax	Description
.global name$_1$[,…, name$_n$]	Identifies symbol(s) that is visible to all other modules.
.def name$_1$[,…, name$_n$]	Identifies symbol(s) that are defined in the current file that can be used in other modules.
.ref name$_1$[,…, name$_n$]	Identifies symbol(s) that can be used in the current file but were defined in another module.
.cdecls [options], name$_1$[,…, name$_n$]	Allows programs that are developed using both C and assembly files to share C headers.
Options: C	Treat the code in the .cdecls block as C source code (default).
CPP	Treat the code in the .cdecls block as C++ source code.
NOLIST	Do not include the converted assembly code in any listing file generated for the containing assembly file (default).
LIST	Include the converted assembly code in any listing file generated for the containing assembly file.
NOWARN	Don't emit warnings on STDERR about C/C++ constructs that can't be converted while parsing the .cdecls source block (default).
WARN	Generate warnings on STDERR about C/C++ constructs that can't be converted while parsing the .cdecls source block.

Table 5.3
Directives that control interchange of information between files

Figure 5.2 shows an assembly program that now includes both instructions and directives. This code snippet is beginning to take the form of a real source file that can be downloaded onto the MSP430FR2355 LaunchPad™.



main.asm

```
;-----------------------------------------------------
; An assembly source file usually
; starts with a comment header.
;-----------------------------------------------------
        .text                   ; Assemble into prog mem
Init:
        mov.w   #1234h, R4      ; initialize R4 with 1234h

Main:
        dec     R4              ; decrement R4
        mov.w   R4, Var1        ; store copy of the counter
        jnz     Main            ; if not equal to zero,
loop

        jmp     Init            ; after loop, repeat

;-----------------------------------------------------
; Variables
        .data                   ; Put variables in RAM
Var1:   .space  2               ; Allocate 16-bit variable
```

The .text directive tells the assembler that the following statements go into program memory.

The .data directive tells the assembler that the following statements go into data memory.

Fig. 5.2
Using directives in an assembly program

5.1.3 Miscellaneous Syntax Notes

5.1.3.1 Identifiers

Identifiers are names used for labels, registers, and symbols. An identifier is a string of alphanumeric characters, the dollar sign, and underscores (A-Z, a-z, 0-9, $, and _). The first character in an identifier cannot be a number, and identifiers cannot contain embedded blanks. The identifiers you define are case sensitive; for example, the assembler recognizes XYZ, Xyz, and xyz as three distinct identifiers. There are built-in identifiers for CPU register names (i.e., PC and R4).

5.1.3.2 Sections

Sections are a block of code or data that occupy a continuous space in the memory map. In an assembly file, specific syntax is used to denote the beginning of a section. The directives .text, .data, and .sect are the main ways to define the start of a section. The linker handles inserting the sections at the appropriate address in the memory map.

5.1.3.3 Case Sensitivity

The CCS environment does not make a distinction in case on mnemonics (i.e., MOV.W is the same as mov.w). In this sense, CCS is not case sensitive; however, as mentioned in Sect. 5.1.3.1, case does matter for identifiers. Mixing case is a common programming mistake (i.e., Main: $\neq$ main:).

Concept Check

CC5.1 If you accidentally inserted a space before an address label named “Main:”, what field would the assembler think the text is in and what error would you expect?

A) It would still consider it a label because of the colon (:) and no error would result.

B) It would consider the text a mnemonic (i.e., field 2) and would give an error that it did not recognize “Main” as a valid instruction mnemonic.

C) It would consider it a comment because comments start with colons (:) and no error would result.

D) It would notice that it is not in the label location and also not a valid mnemonic, so it would assume it is a directive.

5.2 Your First Program: Blinking LED

At this point, you are ready to create and download your first program. In this section you will start a new CCS project and enter the provided code to make LED1 on the LaunchPad™ blink. Follow the steps listed in Example 5.1 to create a new, blank assembly-only CCS project. Once you have done this, you should see the same code as shown in the example. When creating a blank assembly-only project, CCS creates all of the supporting files needed to build the executable object file. This includes information about the MSP430 address map so that the linker knows where to place the sections of your source code. In addition, the project sets up the reset behavior, which is where the MCU will begin executing code when powered up or reset. Finally, the project sets up the initial value of the stack pointer. The main. asm is created for you that provides supporting statements that allow you to just focus on creating the main project.

Example: Creating a Blank CCS Assembly Project

1) Launch CCS: **Start → Texas Instruments → Code Composer Studio** x.x.x.

2) Create a new project: **File → New → CCS Project**.

3) In the "Target field", select: **MSP430FR2355**.

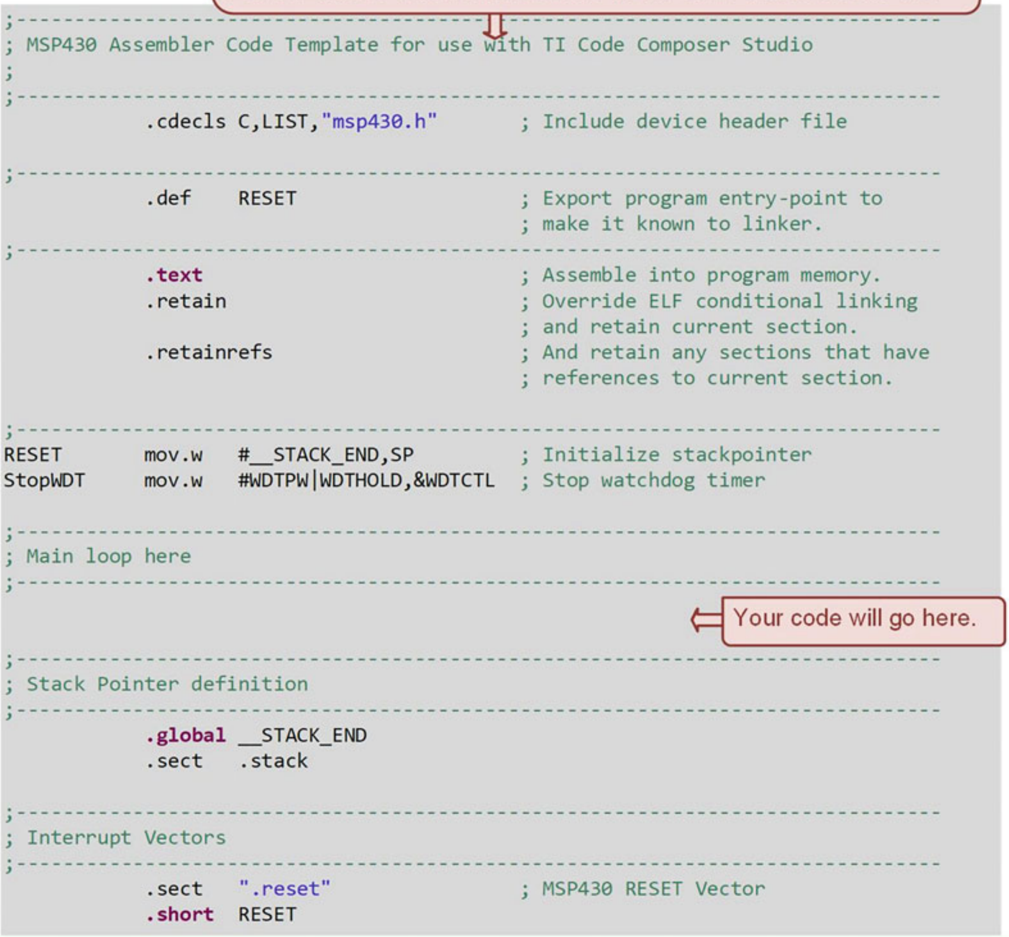
4) The first time launching CCS, you must specify the location to place your CCS workspace. Uncheck "Use Default Location" and browse to a location on your computer where you want to store your projects. After doing this, all new projects will be stored in this location by default.

5) In the "Project name" field, enter: **Asm_Blinky**.

6) In "Project templates and examples", select: **Empty Assembly-only Project**.

7) Click: **Finish**.

You should see a new main.asm file in the editor that looks like this.



```
;-------------------------------------------------------------------------------
; MSP430 Assembler Code Template for use with TI Code Composer Studio
;
;-------------------------------------------------------------------------------
            .cdecls C,LIST,"msp430.h"       ; Include device header file

;-------------------------------------------------------------------------------
            .def    RESET                   ; Export program entry-point to
                                            ; make it known to linker.
;-------------------------------------------------------------------------------
            .text                           ; Assemble into program memory.
            .retain                         ; Override ELF conditional linking
                                            ; and retain current section.
            .retainrefs                     ; And retain any sections that have
                                            ; references to current section.

;-------------------------------------------------------------------------------
RESET       mov.w   #__STACK_END,SP         ; Initialize stackpointer
StopWDT     mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog timer

;-------------------------------------------------------------------------------
; Main loop here
;-------------------------------------------------------------------------------


;-------------------------------------------------------------------------------
; Stack Pointer definition
;-------------------------------------------------------------------------------
            .global __STACK_END
            .sect   .stack

;-------------------------------------------------------------------------------
; Interrupt Vectors
;-------------------------------------------------------------------------------
            .sect   ".reset"                ; MSP430 RESET Vector
            .short  RESET
```

Your code will go here.

Example 5.1
Creating a blank assembly project in CCS

Let's inspect the provided source code provided by CCS. The **.cdecls** directive pulls in a header file that defines register and bit names for the MSP430. This will be very useful as we develop advanced programs that require setting up control registers. We can use the names from the header file instead of trying to look up the absolute addresses of the registers in memory. Note that the "C" option allows the header file to be read as C syntax. The "LIST" option tells CCS to include the converted C header in all merged assembly files created.

The **.def** directive allows the address label RESET to be seen on other project files. This is important because the RESET label will be the starting address to begin executing code when the MCU powers up. Other source files need this address in order to set up the reset condition correctly.

The **.text** directive tells the assembler that the following statements will be put into the program code portion of the memory map. This section starts with two directives: **.retain** and **.retainrefs**. These directives prevent the current section, or other sections referencing it, from being automatically removed when CCS tries to optimize the design.

The next two lines of code are two instruction statements. The first statement beginning with "RESET" initializes the SP register to the end of the data memory (i.e., 03000h). The second statement beginning with StopWDT configures some settings in a peripheral register that disable the watchdog timer. At this point of the book, it is not expected that these instructions are understood. They will be covered later.

Next is a set of comment lines that indicate where your program code will go. This is indicated by "Main loop here." The next two directives define an address called "__STACK_END" that represents the end of data memory. The **.global** directive allows all other files in the project to see this address. Finally, the interrupt vector section is defined using directives to define the starting address of program code and place it in a system called the interrupt controller that will handle retrieving the address and placing it into PC upon power-up.

The next step is to type in the code to make LED1 blink on the LaunchPad™. Follow the instructions in Example 5.2 to enter the blinking LED code, assemble, and download to the MCU. Note that you do not need to understand what the code is doing at this point.

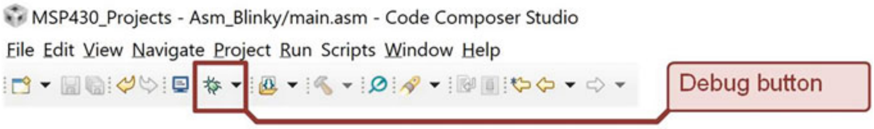
Example: Entering Code to Blink LED1

1) Type the following code into your main.asm file in the location that says "Main loop here".

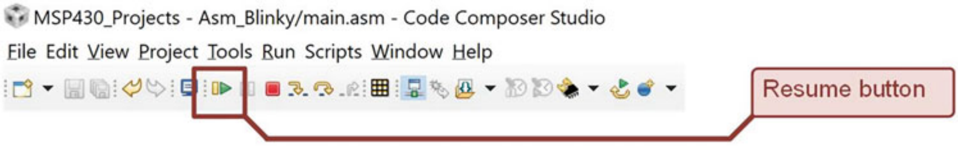
```
;-------------------------------------------------------------------------------
; Main loop here
;-------------------------------------------------------------------------------

init:
        bic.w   #0001h, &PM5CTL0 ; Disable the GPIO power-on default high-Z mode
        bis.b   #01h, &P1DIR     ; Set P1.0 as an output.  P1.0 is LED1

main:
        xor.b   #01h, &P1OUT     ; Toggle P1.0 (LED1)

        mov.w   #0FFFFh, R4      ; Put a big number into R4
delay:
        dec.w   R4               ; Decrement R4
        jnz     delay            ; Repeat until R4 is 0

        jmp     main             ; Repeat main loop forever
```

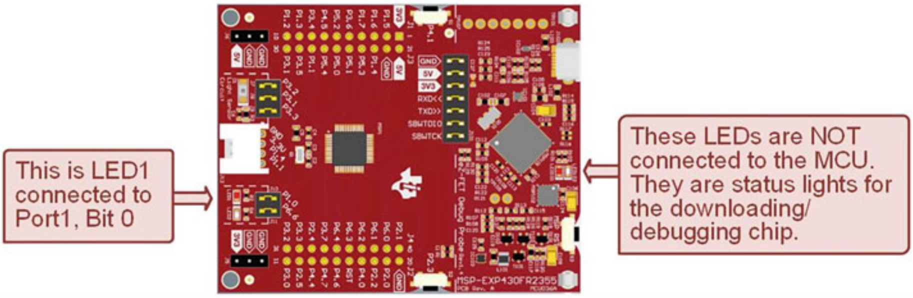
2) Save your main.asm using either File → Save, or CNTL-s.
3) Now *Debug* your program. The term "Debug" includes assembling, linking, and downloading to the LauchPad™. Make sure your LaunchPadTM is plugged in to your computer.

- Note: if you have errors, it is because you didn't type something in correctly. Syntax errors will be shown in the Console pane. You can scroll up the console and view the error location and reason. Fix errors and continue trying to debug until it works.


MSP430_Projects - Asm_Blinky/main.asm - Code Composer Studio
File Edit View Navigate Project Run Scripts Window Help
Debug button

4) Once downloaded, press the *Resume* button in CCS. Note that in debug mode, the buttons at the top of CCS change. You should now see LED1 blinking!


MSP430_Projects - Asm_Blinky/main.asm - Code Composer Studio
File Edit View Project Tools Run Scripts Window Help
Resume button


This is LED1 connected to Port1, Bit 0
These LEDs are NOT connected to the MCU. They are status lights for the downloading/ debugging chip.
MSP-EXP430FR2355

Example 5.2
Getting LED1 to blink

Concept Check

CC5.2 If you created a completely empty main.asm file and inserted the code given in Example 5.2 after a .text directive, what functionality would be missing from your program that is provided in the automatically generated main.asm files that results when you perform “Project→New CCS Project→Empty Assembly-only Project”?

A) The MCU behavior upon reset.

B) Initialization of the Stack Pointer.

C) Including the MSP430 header file.

D) All of the above.

5.3 Using the CCS Debugger

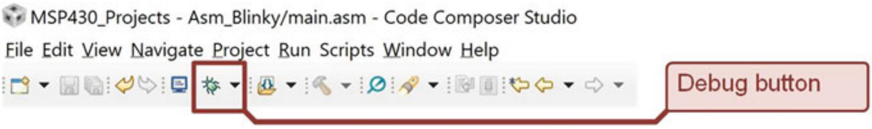
The debugger allows you to start, stop, and pause your program to observe its operation. It also allows you view the contents of registers and memory at specific points in your program. The debugger allows you to also view the binary values that your program is assembled into. When developing in C, the debugger also allows you to see the assembly file that is created during compile. This subsection will introduce some of the most used features of the CCS debugger.

5.3.1 Resume, Terminate, and Suspend

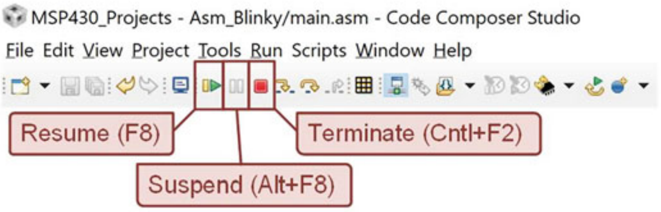
Once a program is in debug mode (i.e., the program has been assembled and downloaded using CCS), the *Resume*, *Suspend*, and *Terminate* commands become available. These commands are available either in the *Run* pull-down menu or using the buttons that appear at the top of CCS when in debug mode. The Resume command will start running your program. The Suspend command will pause your program without ending debug mode. The Terminate command will end your program and exit debug mode. Follow Example 5.3 to experiment with Resume, Suspend, and Terminate.

Example: Resume, Suspend, and Terminating in the Debugger

1) At this point, you should have your Asm_Blinky program running on the LaunchPad™ board. If it is not currently in debug mode, press the debug button to assemble, download, and enter debug mode within CCS.


MSP430_Projects - Asm_Blinky/main.asm - Code Composer Studio
File Edit View Navigate Project Run Scripts Window Help
Debug button

Once in debug mode, you can experiment with the resume, suspend, and terminate commands.


MSP430_Projects - Asm_Blinky/main.asm - Code Composer Studio
File Edit View Project Tools Run Scripts Window Help
Resume (F8)
Terminate (Cntl+F2)
Suspend (Alt+F8)

2) Press **Resume**. Your program should start running and LED1 should start blinking.

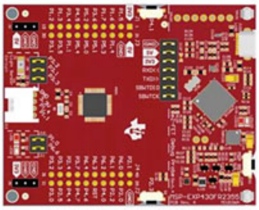
3) Press **Suspend**. This command pauses your program. LED1 will stop in either the ON or OFF state. To start your program again, press the resume button.

Can you pause your program with LED1 ON?

4) Press **Terminate**. This will stop your program and exit debug mode. If you press resume again, it will re-assemble your program, re-download, and enter debug mode.

Note that once you terminate, your program halts and you exit debug mode. But your program is still in program memory.

5) Press the **Reset button** on the LaunchPad™ board. Your program will start running again.



The master Reset will begin running whatever code is in program memory.

Example 5.3
Resume, Suspend, and Terminate in the debugger

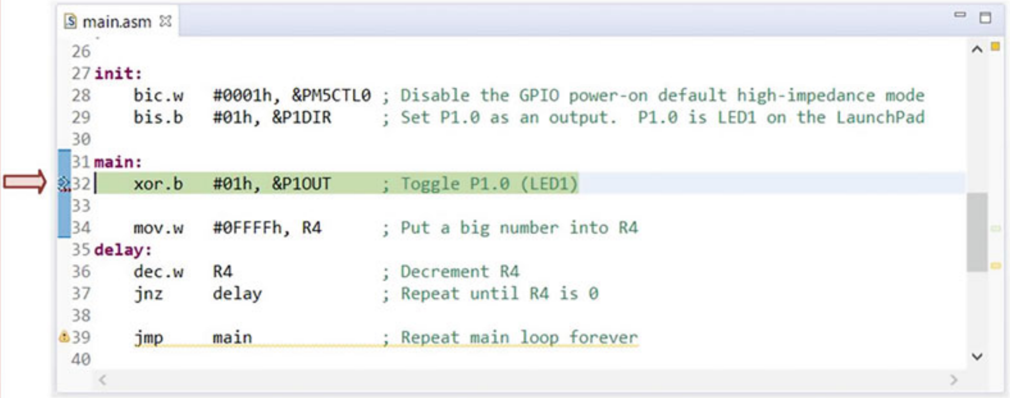
5.3.2 Breakpoints

A *breakpoint* is a stopping point in the code that can be inserted by the debugger. When a program is run and it encounters a breakpoint, it will automatically suspend. Breakpoints allow a developer to suspend a program in a specific point and then simply resume the program and monitor activity once it suspends. Breakpoints can be inserted in either editing or debug mode within CCS. Follow Example 5.4 to experiment with breakpoints.

EXAMPLE: USING BREAKPOINTS

1) At this point you should have your Asm_Blinky program running on the LaunchPad™ board. If it is not currently in debug mode, press the debug button to assemble, download, and enter debug mode within CCS.

2) Enter a breakpoint before the **xor.b** instruction statement. A breakpoint is inserted by double-clicking in the gray, vertical string to the left of the editor pane. Once the breakpoint is entered, a blue circle will appear.



```

init:
        bic.w   #0001h, &PM5CTL0 ; Disable the GPIO power-on default high-impedance mode
        bis.b   #01h, &P1DIR     ; Set P1.0 as an output.  P1.0 is LED1 on the LaunchPad

main:
        xor.b   #01h, &P1OUT     ; Toggle P1.0 (LED1)

        mov.w   #0FFFFh, R4      ; Put a big number into R4
delay:
        dec.w   R4               ; Decrement R4
        jnz     delay            ; Repeat until R4 is 0

        jmp     main             ; Repeat main loop forever

```

3) Press the **Resume** button. Your program will run to the breakpoint and pause. Press resume again and it will do the same thing. Notice that on the LaunchPad™ board each time the program suspends, LED1 toggles.

To remove a breakpoint, double-click on the blue circle.

Example 5.4
Using breakpoints

5.3.3 Viewing Register Contents

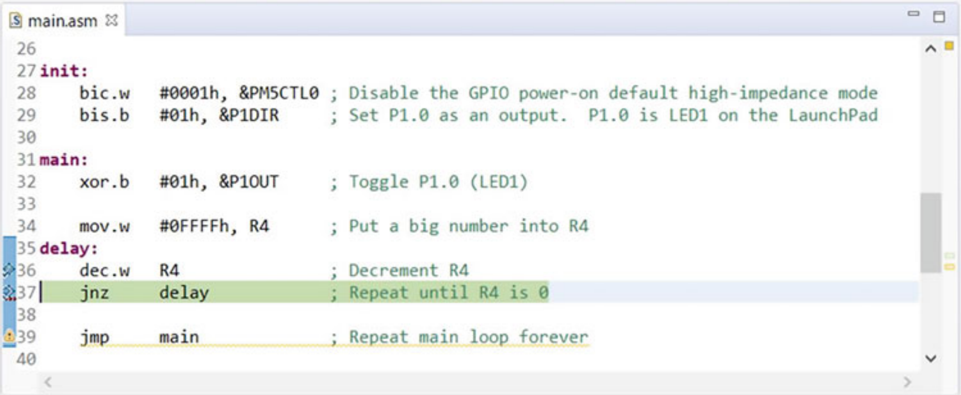
The Register Viewer within CCS allows you to see the contents of CPU registers when your program is suspended. It also allows you to expand the Status Register to see the ALU flags. Follow Example 5.5 to experiment with viewing register contents in the debugger.

Example: Viewing The Contents of Registers

1) At this point, you should have your Asm_Blinky program running on the LaunchPad^{TM} board. If it is not currently in debug mode, press the debug button to assemble, download, and enter debug mode within CCS.

2) If you have any current breakpoints, remove them by double-clicking on the blue circles associated with them.

3) Enter two new breakpoints; the first before the **dec.w** instruction and the second before the **jnz** instruction.



```
init:
    bic.w   #0001h, &PM5CTL0 ; Disable the GPIO power-on default high-impedance mode
    bis.b   #01h, &P1DIR     ; Set P1.0 as an output.  P1.0 is LED1 on the LaunchPad

main:
    xor.b   #01h, &P1OUT     ; Toggle P1.0 (LED1)

    mov.w   #0FFFFh, R4      ; Put a big number into R4
delay:
    dec.w   R4               ; Decrement R4
    jnz     delay            ; Repeat until R4 is 0

    jmp     main             ; Repeat main loop forever
```

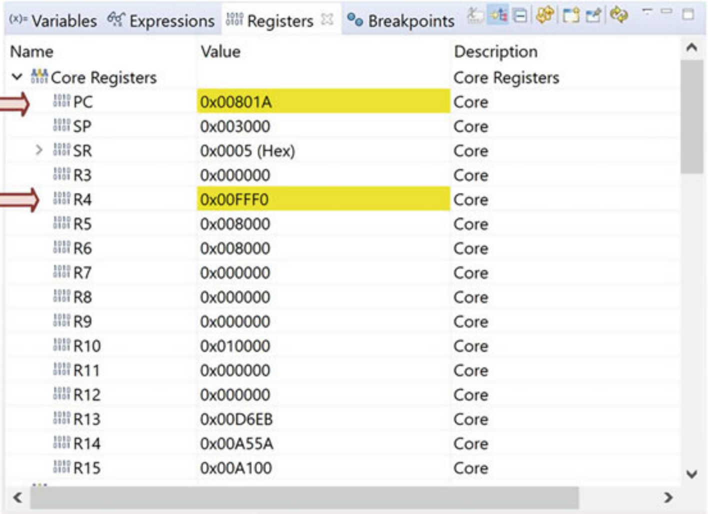
4) When in debug mode, you'll see a Registers tab in one of the panes of CCS. If it isn't visible, you can bring it up using the View→Registers pull-down menu. Expand the "Core Registers" item in the Registers pane. You can now see all of CPU registers.

5) Click the resume button over and over. This will run your program and suspend on one of the two breakpoints. As you do this, you will see the PC and R4 values change.

PC points to the address of the next instruction to be executed. Since our code is located in program memory starting at 08000h, PC will have values in this range.

The Blinky program uses a delay loop to slow down the speed at which LED1 blinks. This is done by loading R4 with FFFFh and then decrementing it until it hits zero.

As you run your program to the BPs, you'll see R4 decrement.



Name	Value	Description
Core Registers		Core Registers
PC	0x00801A	Core
SP	0x003000	Core
SR	0x0005 (Hex)	Core
R3	0x000000	Core
R4	0x00FFF0	Core
R5	0x008000	Core
R6	0x008000	Core
R7	0x000000	Core
R8	0x000000	Core
R9	0x000000	Core
R10	0x010000	Core
R11	0x000000	Core
R12	0x000000	Core
R13	0x00D6EB	Core
R14	0x00A55A	Core
R15	0x00A100	Core

Example 5.5
Viewing the contents of registers

5.3.4 Viewing the Contents of Memory

The CCS *memory browser* allows you to see the contents of the memory system. This can be opened using the View→Memory Browser pull-down menu. Follow Example 5.6 to experiment with the memory browser.

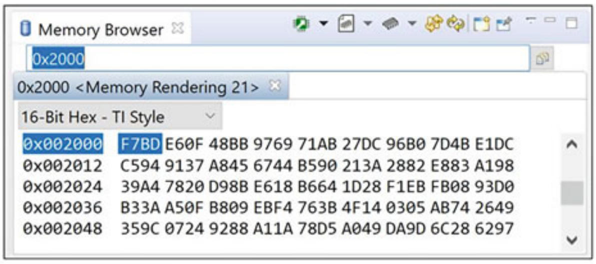
EXAMPLE: VIEWING THE CONTENTS OF MEMORY

1) At this point, you should have your Asm_Blinky program running on the LaunchPad™ board. If it is not currently in debug mode, press the debug button to assemble, download, and enter debug mode within CCS.

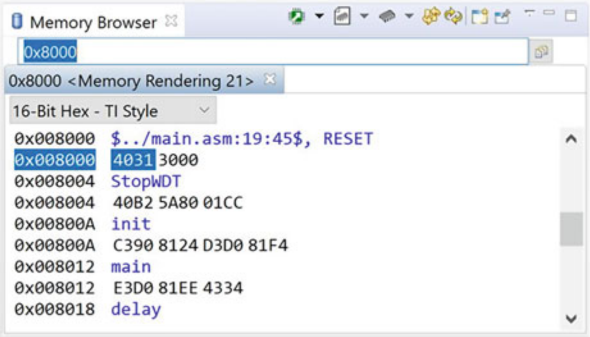
2) Open the Memory Browser. If it is not already open in CCS, you can bring it up using the View→Memory Browser pull-down memory.

3) The Memory Browser lets you see the contents of memory at each address. You can search for a certain address by typing it in the search field at the top of the window pane. Note that the format can be entered in either decimal or hex. But hex is specified using the "0x" prefix format.

Enter **0x2000** in the search field and view the contents of data memory on the MCU. Blinky doesn't use any data memory, so the contents will either be all 1's or have values from a prior program.



```
0x2000
0x2000 <Memory Rendering 21>
16-Bit Hex - TI Style
0x002000  F7BD E60F 48BB 9769 71AB 27DC 96B0 7D4B E1DC
0x002012  C594 9137 A845 6744 B590 213A 2882 E883 A198
0x002024  39A4 7820 D98B E618 B664 1D28 F1EB FB08 93D0
0x002036  B33A A50F B809 EBF4 763B 4F14 0305 AB74 2649
0x002048  359C 0724 9288 A11A 78D5 A049 DA9D 6C28 6297
```



```
0x8000
0x8000 <Memory Rendering 21>
16-Bit Hex - TI Style
0x008000  $../main.asm:19:45$, RESET
0x008000  4031 3000
0x008004  StopWDT
0x008004  40B2 5A80 01CC
0x00800A  init
0x00800A  C390 8124 D3D0 81F4
0x008012  main
0x008012  E3D0 81EE 4334
0x008018  delay
```

4) Next, let's look at our program code. Enter **0x8000** in the search field. Since the debugger knows that this is program code, it will insert the address labels into the memory browser. The values that you see are the actual opcode and operand binaries for the Blinky program.

Example 5.6
Viewing the contents of memory

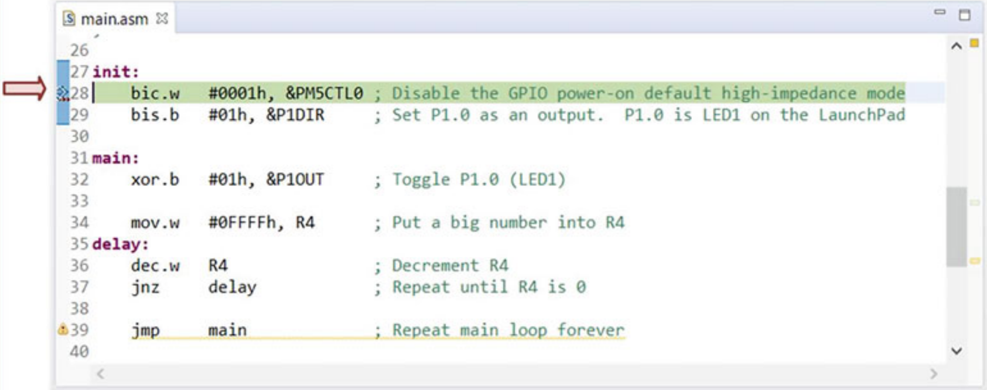
5.3.5 Stepping Your Program

Stepping your program refers to executing your program line by line. The CCS debugger supports two stepping commands: *Step Into* and *Step Over*. These two commands are functionally identical except when it comes to subroutine calls. If a subroutine exists in your program and you are using the Step Into command, when the debugger reaches the subroutine call, it will move into the subroutine as you step. If you use the Step Over command, it will simply execute the entire subroutine without entering it and continue to the next instruction in the main program. Follow Example 5.7 to experiment with stepping.

Example: Stepping a Program

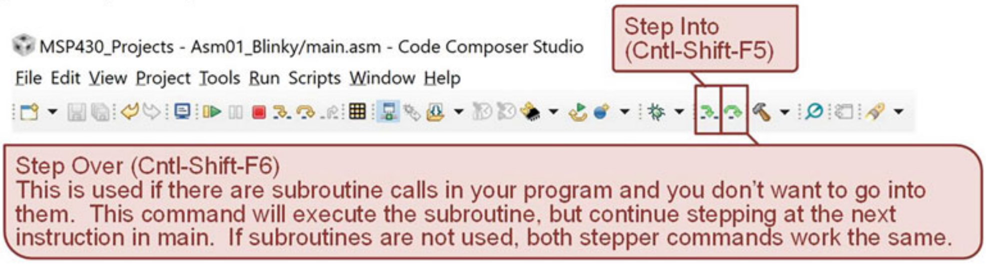
1) At this point, you should have your Asm_Blinky program running on the LaunchPad™ board. If it is not currently in debug mode, press the debug button to assemble, download, and enter debug mode within CCS.

2) Remove all breakpoints. Insert a new breakpoint at the **bic.w** instruction. Run your program to this breakpoint.



```
init:
    bic.w   #0001h, &PM5CTL0 ; Disable the GPIO power-on default high-impedance mode
    bis.b   #01h, &P1DIR     ; Set P1.0 as an output.  P1.0 is LED1 on the LaunchPad

main:
    xor.b   #01h, &P1OUT     ; Toggle P1.0 (LED1)

    mov.w   #0FFFFh, R4      ; Put a big number into R4
delay:
    dec.w   R4               ; Decrement R4
    jnz     delay            ; Repeat until R4 is 0

    jmp     main             ; Repeat main loop forever
```

3) Now press the **Step Into** button. This will execute your program instruction by instruction. Keep clicking Step Into and watch your program execute. You can also observe the values of registers as you step.


MSP430_Projects - Asm01_Blinky/main.asm - Code Composer Studio
File Edit View Project Tools Run Scripts Window Help

Step Into (Cntl-Shift-F5)

Step Over (Cntl-Shift-F6)
This is used if there are subroutine calls in your program and you don't want to go into them. This command will execute the subroutine, but continue stepping at the next instruction in main. If subroutines are not used, both stepper commands work the same.

Example 5.7
Stepping a program

Concept Check

CC5.3 If you decided to allocate and initialize a variable in data memory, what debugger tool would you use to verify they were set up correctly after downloading?

A) The stepper.

B) The Register Viewer.

C) The Memory Viewer.

D) A breakpoint.

Summary

- Assembly source files are made up of instruction and directive statements in addition to comments.
- An instruction statement has four fields: a label, the mnemonic, the operand, and the comment. Address labels are optional. Operands are only included if needed by the instruction. Comments are optional but recommended.
- Address labels mark the address of where the instruction code will reside. These are useful because the developer doesn't need to keep track of the exact address for looping or conditional jumping.
- Comments in the CCS assembler start with a semicolon (;).
- Assembler directives are statements that tell the assembler information about the program but are not instructions.
- Assembler directives can be used to control where sections are located in memory, reserving/initializing memory for constants/variables, and the behavior of information interchange between different files in the project.
- An identifier is a string of alphanumeric characters used to name labels, registers, and symbols. Identifiers cannot start with a number or have blank characters. They can include $ and _.
- A section is a block of code or data that occupy a continuous space in the memory map.
- CCS allows mixed case for instruction mnemonics (i.e., mov = MOV). CCS does interpret case in identifiers (i.e., Var1: ≠ var1).
- When creating a new blank assembly-only project in CCS, it provides a main.asm file with assembly code already inserted to handle including the MSP430 memory map details as a header, the reset condition, and the initialization of the stack pointer.
- A debugger allows the developer to control the execution of the program and observe the values held in registers and memory.
- The CCS assembler starts and stops the program using *resume*, *terminate*, and *suspend* commands.
- Breakpoints allow a location in the program code to be marked as a stopping point when the program is resumed.
- The Register Viewer shows the contents of the CPU register while debugging.
- The memory browser shows the contents of the memory map while debugging. Addresses can be searched in the memory browser, but hex addresses must follow the "0x" prefix notation.
- A program can be *stepped*, which gives the ability to run the program instruction by instruction.

Exercise Problems

Section 5.1: The Anatomy of an Assembly Program File

5.1.1 What are the four fields of an instruction statement?

5.1.2 What are address labels used for?

5.1.3 Do all instruction statements require an operand?

5.1.4 What is the purpose of a comment?

5.1.5 What is an assembler directive?

5.1.6 What assembler directive instructs the assembler to put the subsequent instruction statements into the program memory section of the memory map?

5.1.7 What assembler directive instructs the assembler to put the subsequent variable allocation statements into the data memory section of the memory map?

5.1.8 How many bytes in memory does a **.space** directive allocate if a size of 3 is provided?

5.1.9 How many bytes in memory does a **.byte** directive allocate if only one value is provided?

5.1.10 How many bytes in memory does a **.short** directive allocate if only one value is provided?

5.1.11 How many bytes in memory does a **.long** directive allocate if only one value is provided?

5.1.12 How many bytes in memory does a **.char** directive allocate if only one character is provided?

5.1.13 What is the difference between a **.string** and **.cstring** directive?

5.1.14 How many bytes in memory does a **.float** directive allocate if only one number is provided?

5.1.15 How many bytes in memory does a **.double** directive allocate if only one number is provided?

5.1.16 What does the directive **.global** do?

5.1.17 What does the directive **.def** do?

5.1.18 What does the directive **.ref** do?

5.1.19 What does the directive **.cdecls** do?

5.1.20 What two non-alphanumeric characters are allowed in a CCS assembly identifier?

5.1.21 What is a section?

5.1.22 Are instruction mnemonics case sensitive in the CCS environment?

5.1.23 Are identifiers case sensitive in the CCS environment?

Section 5.2: Your First Program: Blinking LED

5.2.1 When you create a new blank assembly-only project in CCS, a new main.asm is created with what three steps handled for you?

5.2.2 What three things does the *debug* command do for you?

5.2.3 What is the name of the MSP430 header file?

5.2.4 What is the name of the reset vector section?

5.2.5 What comment text is inserted into the main.asm to let you know where to insert your program?

Section 5.3: Using the CCS Debugger

5.3.1 What does the *resume* command do?

5.3.2 What does the *suspend* command do?

5.3.3 What does the *terminate* command do?

5.3.4 After entering debug mode, the program is downloaded to the LaunchPad™ board, but it is not running. What is an alternative way to make the program run without issuing a resume command?

5.3.5 What is the purpose of a breakpoint?

5.3.6 If you insert a breakout on the line of an instruction, does the program stop before or after the instruction on that line is executed?

5.3.7 In the Register Viewer, under what item name are the CPU registers listed?

5.3.8 In the memory browser, what is the number syntax to use if you want to go look at address 2000h?

5.3.9 What does *stepping* your program do?

5.3.10 What is the difference between *Step Into* and *Step Over*?

Chapter 6: Data Movement Instructions

This chapter delves deeper into the details of the primary data movement instructions within the MSP430 [1]. As soon as we start using an instruction where we need to specify locations of CPU registers or memory locations, the concept of an *addressing mode* needs to be introduced. As such, the majority of this chapter looks at the seven addressing modes available in the MSP430 CPU and how they are used with the **`mov`** instruction. It is intended that the reader is coding the examples using the MSP430FR2355 LaunchPad™ board as they go through this chapter.

Learning Outcomes—After completing this chapter you will be able to:

6.1 Use register mode addressing to copy data between CPU registers.
6.2 Use immediate mode addressing to put constants into registers and memory.
6.3 Use absolute mode addressing to access memory.
6.4 Use symbolic mode addressing to access memory.
6.5 Use indirect register mode addressing to access memory.
6.6 Use indirect autoincrement mode addressing to access memory.
6.7 Use index mode addressing to access memory.

6.1 The MOV Instruction with Register Mode (Rn) Addressing

The *move* instruction has a mnemonic of **`mov`** and is the primary instruction to copy information within the computer system. While the instruction is named “move,” it actually performs a copy operation. This instruction has an operand format of `src, dst` where the src is the location of where the information is to be copied from, and the dst is the location of where the information is to be copied to. The src and dst can either be CPU registers or locations in memory. The move instruction performs both 16-bit and 8-bit operations, dictated by the extensions .w and .b. If no extension is used, the instruction is assumed to be a 16-bit operation.

An *addressing mode* is the way that the src and dst locations are specified in the operand of an instruction. The MSP430 provides seven different addressing modes. The best way to learn these modes is to experiment with each while observing how the addresses are formed and data is moved in the CCS debugger. We will begin with *register mode* addressing. In register mode addressing, the operand for either the source or destination are CPU registers. The register names are provided using their unique identifiers (i.e., PC, SR, SP, R4, R5, ..., R15). The instruction operates on the data held within the register names provided. The syntax to denote register mode addressing in the MSP430 documentation is *Rn*. Follow Example 6.1 to gain experience with how register mode addressing works when programming in assembly. This example copies the values of PC and SP into the other general-purpose registers. PC and SP are used because they are initialized by the rest of the CCS automatically generated main.asm prior to entering the main loop.

B. J. LaMeres, *Embedded Systems Design using the MSP430FR2355 LaunchPad™*,
https://doi.org/10.1007/978-3-031-20888-1_6

Example: Register Mode Addressing (R_N)

In *register mode* addressing, the register identifiers are simply used in the operand. Register mode is valid in both the src and dst. Let's look at how this mode works on the LaunchPad™ board. This example will move information between CPU registers.

1) Create a new Empty Assembly-only CCS project titled: **Asm_AddrMode1_Register**.

2) Type in the following code into the main.asm file where the comments say "Main loop here".

Set breakpoint here

```
main:
        mov.w   PC, R4     ; copy PC into R4
        mov.w   R4, R5     ; copy R4 into R5
        mov.w   R5, R6     ; copy R5 into R6

        mov.b   PC, R7     ; copy LB of PC into R7
        mov.b   R7, R8     ; copy LB of R7 into R8
        mov.b   R8, R9     ; copy LB of R8 into R9

        mov.w   SP, R10    ; copy SP into R10
        mov.w   R10, R11   ; copy R10 into R11
        mov.w   R11, R12   ; copy R11 into R12

        mov.b   SP, R13    ; copy LB of SP into R13
        mov.b   R13, R14   ; copy LB of R13 into R14
        mov.b   R14, R15   ; copy LB of R14 into R15

        jmp     main
```

16-bit operations

8-bit operations

16-bit operations

8-bit operations

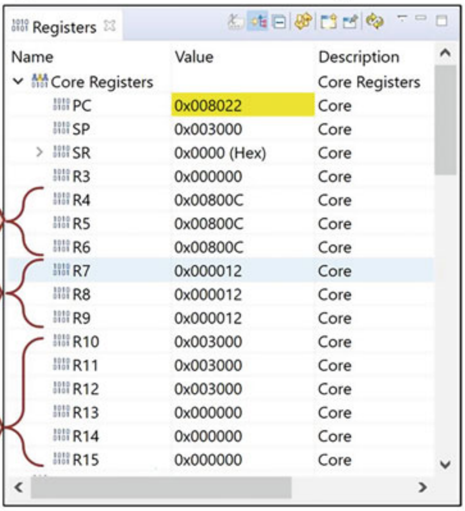
3) Debug your program. If you have errors, correct them and continue debugging until your program is successfully downloaded to the LaunchPad™ board.

4) Set a breakpoint before the first instruction (**mov.w PC, R4**).

5) Open the register viewer and expand the Core Registers item to see the CPU registers.

6) Run your program to the breakpoint.

7) Step your program through each instruction and observe the data movement in the register viewer.



Name	Value	Description
Core Registers		Core Registers
PC	0x008022	Core
SP	0x003000	Core
SR	0x0000 (Hex)	Core
R3	0x000000	Core
R4	0x00800C	Core
R5	0x00800C	Core
R6	0x00800C	Core
R7	0x000012	Core
R8	0x000012	Core
R9	0x000012	Core
R10	0x003000	Core
R11	0x003000	Core
R12	0x003000	Core
R13	0x000000	Core
R14	0x000000	Core
R15	0x000000	Core

Notice how the value of PC when the first mov.w is executed is copied into R4, then R4 is copied into R5, then R5 is copied into R6.

The next three moves only copy the lower 8-bits of PC into R7, then R7→R8, R8→R9. Notice that when these instructions are executed, PC has incremented to a new location in program memory.

The same functionality is then performed on the SP. SP is initialized to 3000h, but isn't altered by this program.

Example 6.1
Register mode addressing

Concept Check

CC6.1 When executing an instruction using register mode addressing, is there any point during the execution that memory is accessed if the intent of the instruction is to move data only between CPU registers?

A) Yes. Even though register mode addressing only moves data within the CPU, every instruction has to access memory at least once in order to fetch its op-code.

B) No. There is no need to access memory because data is only moved within the CPU.

6.2 The MOV Instruction with Immediate Mode (#N) Addressing

In *immediate mode* addressing, the operand for the src is a numeric constant. Since the operand now contains a number that could be interpreted as either a constant or address, there needs to be a way to indicate how the number is to be used. In immediate mode, a # is placed in front of the number to indicate it is to be used as a constant. Follow Example 6.2 to experiment with immediate mode addressing. Notice in this example that we are now mixing two types of addressing modes. The src in all of the **`mov`** instructions uses immediate mode, while the destination address uses register mode. This is perfectly acceptable for the **`mov`** instruction but does bring up some limitations on where certain addressing modes can be used. We cannot use immediate mode for the dst because it wouldn't make sense to move a numerical constant into another numerical constant because the immediate mode constant in the src is not a storage element of any type. The constant is instead embedded as part of the operand code in program memory. This means immediate mode is only valid for the src field of the operation. This brings up the fact that not all modes are valid in both the src and dst fields. Limitations of subsequent addressing modes will be highlighted.

Example: Immediate Mode Addressing (#)

In *immediate mode* addressing, the src is a numeric constant. To indicate that the number being provided is intended to be used as a constant, a # is placed before the number. Let's look at how this mode works on the LaunchPad™ board. This program moves a series of constants into CPU registers using different formats.

1) Create a new Empty Assembly-only CCS project titled: **Asm_AddrMode2_Immediate**.

2) Type in the following code into the main.asm file where the comments say "Main loop here".

```
main:
        mov.w   #1234h, R4          ; put the value 1234h into R4
        mov.w   #0FACEh, R5         ; put the value FACEh into R5

        mov.b   #99h, R6            ; put the value 99h into R6
        mov.b   #0EEh, R7           ; put the value EEh into R7

        mov.w   #371, R8            ; 371 decimal into R8
        mov.b   #10101010b, R9      ; 10101010 binary into R9
        mov.b   #'B', R10           ; ASCII code for B (42h) into R10

        jmp     main
```

Set breakpoint here

Remember to add a "0" to the beginning of any HEX value starting with a letter.

16-bit operations

8-bit operations

Showing decimal, binary, and char formats

3) Debug your program. If you have errors, correct them and continue debugging until your program is successfully downloaded to the LaunchPad™ board.

4) Set a breakpoint before the first instruction (**mov.w #1234h, R4**).

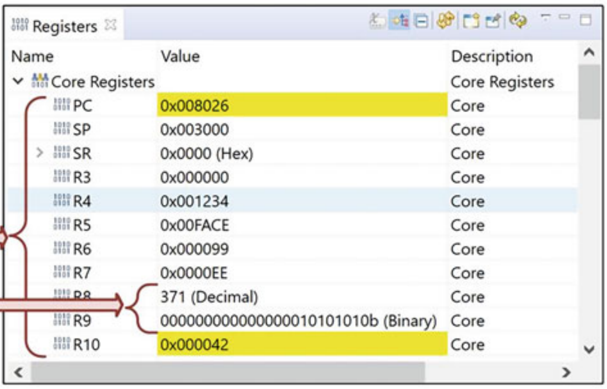
5) Open the register viewer and expand Core Registers item so you can see the CPU registers.

6) Run your program to the breakpoint.

7) Step your program and observe the constants appear in each register.

The constants will appear in each register as the mov instructions are executed.

You can change the format of the numbers in the register viewer by Right-Clicking on the number and selecting Number Format.



Registers

Name	Value	Description
Core Registers		Core Registers
PC	0x008026	Core
SP	0x003000	Core
SR	0x0000 (Hex)	Core
R3	0x000000	Core
R4	0x001234	Core
R5	0x00FACE	Core
R6	0x000099	Core
R7	0x0000EE	Core
R8	371 (Decimal)	Core
R9	000000000000000010101010b (Binary)	Core
R10	0x000042	Core

Example 6.2
Immediate mode addressing

Concept Check

CC6.2 Can immediate mode addressing be used for the dst? Why or why not?

A) Yes. All addressing modes are supported for both the src and dst.

B) No. It doesn't make sense to move a number into a number.

6.3 The MOV Instruction with Absolute Mode (&ADDR) Addressing

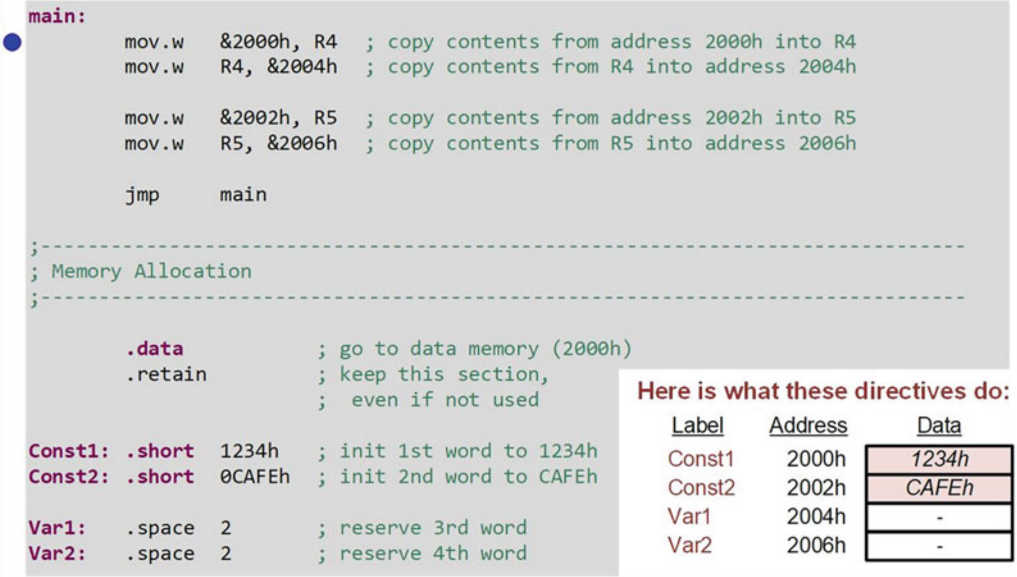
In *absolute mode* addressing, the src and/or dst is a 16-bit address value. This mode is the first to allow us to access the memory system of the MSP430. This mode only supports 16-bit addressing, meaning it can access $2^{16} = 65{,}536$ unique address locations (i.e., address $0_{10} \rightarrow 65{,}535_{10}$ or $0000_{16} \rightarrow \text{FFFF}_{16}$). This is often stated in the MSP430 documentation as being able to access the first 64K addresses in memory. Recall that “64K” is shorthand for the actual value 65,536. The term *absolute* means that the address provided is the *actual* address to access and not a variable name or label. To denote that we are providing an absolute address, the value must be preceded by an &. Addresses are most commonly listed in hexadecimal format due to their large sizes. It is important to remember to precede hex values that start with a letter with a “0” or the assembler will interpret the value as a symbol and not a number. Follow Example 6.3 to gain experience with absolute mode addressing. Note that since we are beginning to move information into and out of memory, we will need to use the .data directive to allocate some variable space in data memory. Example 6.3 also shows how to allocate and initialize memory.

Example: Absolute Mode Addressing (&)

In *absolute mode* addressing, the operand is a 16-bit address. This 16-bit address is the actual address that will be accessed. To indicate that the number being provided is intended to be used as a 16-bit address, an & is placed before the number. Let's look at how this mode works on the LaunchPad™ board. This program will move information back and forth between memory and CPU registers.

1) Create a new Empty Assembly-only CCS project titled: **Asm_AddrMode3_Absolute.**

2) We are going to create a main program and also initialize some space in memory that we can access. We will put the .data section directly after the main program code. Type in the following code into the main.asm file where the comments say "Main loop here".



```
main:
        mov.w   &2000h, R4   ; copy contents from address 2000h into R4
        mov.w   R4, &2004h   ; copy contents from R4 into address 2004h

        mov.w   &2002h, R5   ; copy contents from address 2002h into R5
        mov.w   R5, &2006h   ; copy contents from R5 into address 2006h

        jmp     main

;-------------------------------------------------------------------------------
; Memory Allocation
;-------------------------------------------------------------------------------

        .data           ; go to data memory (2000h)
        .retain         ; keep this section,
                        ;  even if not used

Const1: .short  1234h   ; init 1st word to 1234h
Const2: .short  0CAFEh  ; init 2nd word to CAFEh

Var1:   .space  2       ; reserve 3rd word
Var2:   .space  2       ; reserve 4th word
```

Here is what these directives do:

Label	Address	Data
Const1	2000h	*1234h*
Const2	2002h	*CAFEh*
Var1	2004h	-
Var2	2006h	-

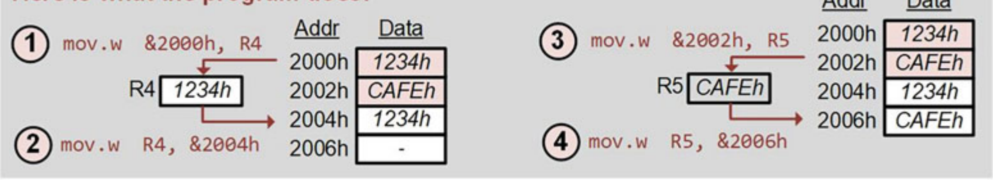
3) Debug your program. Fix errors as needed.

4) Set a breakpoint before the first instruction (**mov.w &2000h, R4**).

5) Open the register viewer and memory browser. Inspect the memory starting at 2000h.

6) Run & step your program while observing the registers and memory.

Here is what the program does:



Example 6.3
Absolute mode addressing

CONCEPT CHECK

CC6.3 If absolute addressing uses a numeric address, why don't we just list the number of the address and skip the & sign?

A) Since hex numbers can start with letters, we need a way to distinguish letters that represent address values versus letters that represent address labels.

B) It really doesn't matter if you use the & sign. The assembler will figure out whether the operand is an absolute address or an address label.

C) Since absolute addressing can also be used to access CPU registers, we need a way to distinguish between registers and memory locations.

D) The answer to this concept check is A.

6.4 The MOV Instruction with Symbolic Mode (ADDR) Addressing

One of the downsides of absolute mode addressing covered in Sect. 6.3 is that the developer needs to keep track of the exact address values being used in memory. This becomes very difficult as programs get larger and multiple developers are involved. In order to overcome this issue, assemblers support the use of *address labels*. An address label is a sequence of characters that identifies a location in memory. This can be thought of as a unique name associated with an address location. The MSP430 allows the use of address labels as operands in *symbolic mode*. In symbolic mode, the address label is simply inserted in either the src or dst fields without any preceding indicator (i.e., no "&" is needed as in absolute mode). If the assembler sees any non-numeric character in an operand, it treats it as an address label. This is why when we use hex constant literal, we need to precede values starting with letters with a 0 (i.e., ABCDh must be entered as 0ABCDh). Symbolic mode supports labels for 16-bit addresses so that just like absolute mode, it can only access memory between 0 → 65,535 and 0000h → FFFFh. Follow Example 6.4 to gain some experience with symbolic mode. Note that this example is functionally identical to Example 6.3 with the exception that address labels are used to access data memory instead of absolute addresses.

Example: Symbolic Mode Addressing

In *symbolic mode* addressing, an address label is used as the operand. An address label is a sequence of characters that represents a 16-bit address in memory. No special syntax is needed to indicate symbolic mode. Let's look at how this mode works on the LaunchPad™ board. This program will move information back and forth between memory and CPU registers.

1) Create a new Empty Assembly-only CCS project titled: **Asm_AddrMode4_Symbolic**.

2) We are going to create a main program and also initialize some space in memory that we can access. We will put the .data section directly after the main program code. Type in the following code into the main.asm file where the comments say "Main loop here".

```
main:
        mov.w    Const1, R4       ; copy contents at address label Const1 into R4
        mov.w    R4, Var1         ; copy contents from R4 into address label Var1

        mov.w    Const2, R5       ; copy contents at address label Const2 into R5
        mov.w    R5, Var2         ; copy contents from R5 into address label Var2

        jmp      main

;-------------------------------------------------------------------------------
; Memory Allocation
;-------------------------------------------------------------------------------

        .data              ; go to data memory (2000h)
        .retain            ; keep this section,
                           ;  even if not used

Const1: .short   1234h     ; init 1st word to 1234h
Const2: .short   0CAFEh    ; init 2nd word to CAFEh

Var1:   .space   2         ; reserve 3rd word
Var2:   .space   2         ; reserve 4th word
```

Here is what these directives do:

Label	Address	Data
Const1	2000h	*1234h*
Const2	2002h	*CAFEh*
Var1	2004h	-
Var2	2006h	-

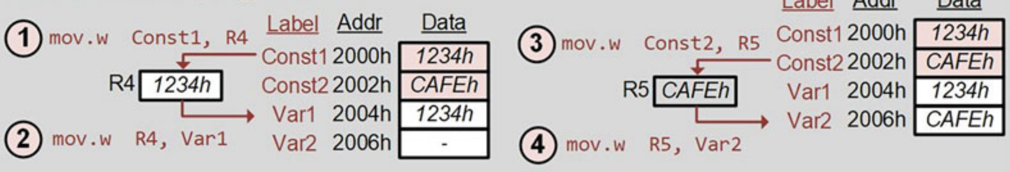
3) Debug your program. Fix errors as needed.

4) Set a breakpoint before the first instruction (**mov.w Const1, R4**).

5) Open the register viewer and memory browser. Inspect the memory starting at 2000h.

6) Run & step your program while observing the registers and memory.

Here is what the program does:



Example 6.4
Symbolic mode addressing

CONCEPT CHECK

CC6.4 Can symbolic mode addressing be used to access CPU registers?

A) Yes. Since the operand is simply an alphanumeric identifier, CPU registers are legal.

B) No. Symbolic addressing is only for address labels. If you enter the name of a CPU register as an operand, it will use register mode addressing.

6.5 The MOV Instruction with Indirect Register Mode (@Rn) Addressing

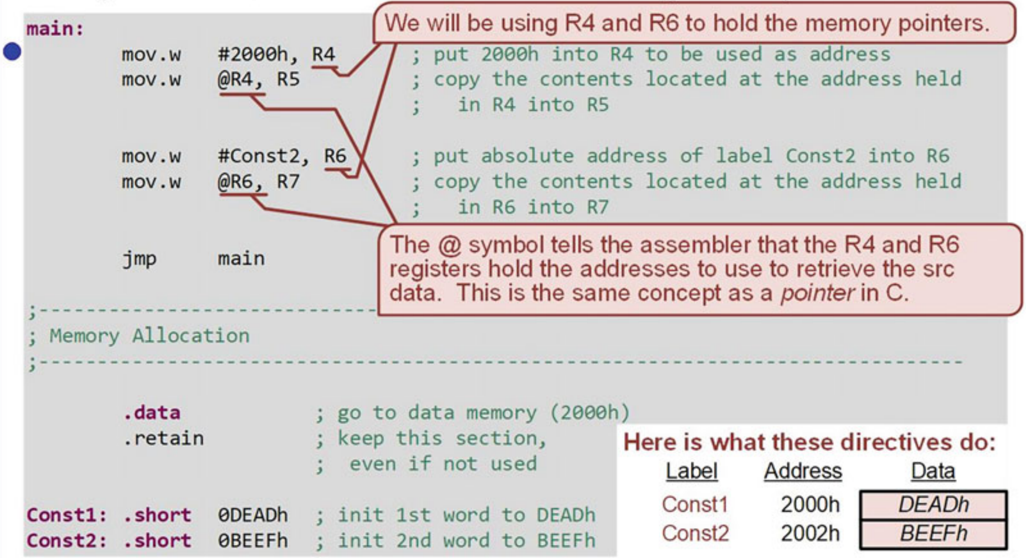
In *indirect register* mode, a CPU register is used to provide the address of where the information to be accessed is stored. This is the same concept as a *pointer* in the language C. The motivation for providing the address in this way is that the CPU register can be modified by subsequent instructions. This provides a way to access blocks of memory by incrementing the address pointer in a loop. To indicate that a register is to be used as an address pointer, an @ is inserted before the register name. In order to use indirect register mode, the program must first initialize the register to be used with the address of where the information is stored using a separate instruction, typically a `mov`. Indirect register mode is only valid in the src of an instruction. This mode supports full 20-bit addressing, meaning it can access $2^{20} = $ 1,048,576 unique address locations (i.e., address 0 → 1,048,575 or 00000h → FFFFFh). This is often stated in the MSP430 documentation as being able to access the first 1M addresses in memory. Recall that "1M" is shorthand for the actual value 1,048,576; however, keep in mind that the MSP430FR2355 MCU only has a 64 kB memory system, so 20-bit addressing will never be used in this book. Indirect register mode is the reason that the MSP430 registers are actually 20-bits wide. Indirect register mode is only allowed in the src. Follow Example 6.5 to gain experience using indirect register mode.

Example: Indirect Register Mode Addressing (@Rn)

In *indirect register mode* addressing, a CPU register name is provided that contains the address of where the information to be accessed is held. An @ is placed in front of the register name to indicate the use of indirect register mode. Let's look at how this mode works on the LaunchPad™ board. This program moves information from memory into CPU registers using indirect register mode to provide the address.

1) Create a new Empty Assembly-only CCS project titled: **Asm_AddrMode5_Indirect_Register**.

2) We are going to create a main program and also initialize some space in memory that we can access. We will put the .data section directly after the main program code. Type in the following code into the main.asm file where the comments say "Main loop here".



```
main:
        mov.w   #2000h, R4      ; put 2000h into R4 to be used as address
        mov.w   @R4, R5         ; copy the contents located at the address held
                                ;   in R4 into R5

        mov.w   #Const2, R6     ; put absolute address of label Const2 into R6
        mov.w   @R6, R7         ; copy the contents located at the address held
                                ;   in R6 into R7

        jmp     main

;-------------------------------------------------------------------------------
; Memory Allocation
;-------------------------------------------------------------------------------

        .data                   ; go to data memory (2000h)
        .retain                 ; keep this section,
                                ;  even if not used

Const1: .short  0DEADh          ; init 1st word to DEADh
Const2: .short  0BEEFh          ; init 2nd word to BEEFh
```

We will be using R4 and R6 to hold the memory pointers.

The @ symbol tells the assembler that the R4 and R6 registers hold the addresses to use to retrieve the src data. This is the same concept as a *pointer* in C.

Here is what these directives do:

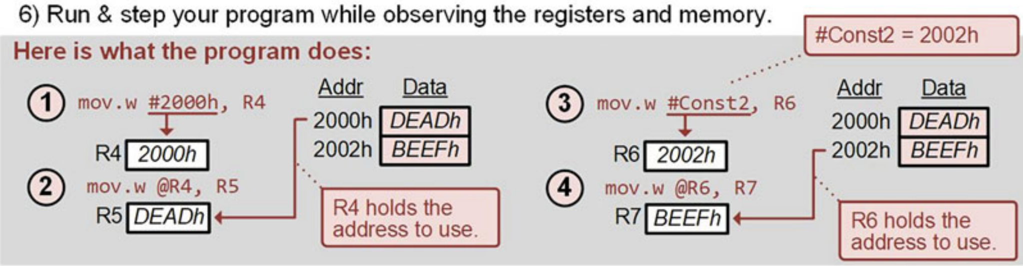
Label	Address	Data
Const1	2000h	*DEADh*
Const2	2002h	*BEEFh*

3) Debug your program. Fix errors as needed.

4) Set a breakpoint before the first instruction (**mov.w #2000h, R4**).

5) Open the register viewer and memory browser. Inspect the memory starting at 2000h.

6) Run & step your program while observing the registers and memory.



#Const2 = 2002h

R4 holds the address to use.

R6 holds the address to use.

Example 6.5
Indirect register mode addressing

CONCEPT CHECK

CC6.5 Can I use the PC and SP in indirect register mode addressing?

A) Yes. All CPU registers can be used to hold addresses; go for it.

B) No. While PC and SP hold addresses, they are dedicated for other purposes. If you use them you will mess up the operation of the CPU.

6.6 The MOV Instruction with Indirect Autoincrement Mode (@Rn+) Addressing

One useful behavior of indirect register mode is its amenability to access large blocks of information in memory by incrementing the address pointer in a loop. For example, consider a **`mov`** instruction that is placed inside of a looping structure that repeats n-times. If after the **`mov`** instruction completes, the pointing register is incremented to the next location in memory, the loop can repeat, and the next time the **`mov`** instruction is executed, it will access the next location in memory. In this way, a block of information in memory that is n-words long can be copied or modified within a loop. This functionality is such a common use of indirect register mode that the MSP430 actually contains a dedicated addressing mode to automate the process of incrementing the pointer register. In *indirect autoincrement* mode, the pointing register is automatically incremented after the completion of the instruction. The pointing register can be incremented by 1 or 2 depending on the type of the size of the operation dictated by .w and .b. Indirect autoincrement mode is only allowed in the src. Follow Example 6.6 to gain experience with indirect autoincrement mode addressing.

Example: Indirect Autoincrement Mode Addressing (@Rn+)

In *indirect autoincrement mode* addressing, a CPU register name is provided that contains the address of where the information to be accessed is held, preceded by an @ (just like in indirect register mode). However, a + sign is placed after the register name indicating that after the address is used, the address register should be incremented. Instructions with .b extensions will increment by 1. Instructions with .w extensions will increment by 2. Let's look at how this mode works on the LaunchPad™ board. This program moves information from memory into CPU registers allowing the autoincrement feature to provide the addresses.

1) Create an Empty Assembly-only CCS project titled: **Asm_AddrMode6_Indirect_Autoincrement**.

2) Type in the following code into the main.asm file where the comments say "Main loop here".

```
main:
        mov.w   #Block1, R4  ; put the value 2000h into R6 to be used as address

        mov.w   @R4+, R5     ; copy data at addr held in R4 into R5, then R4+2-->R4
        mov.w   @R4+, R6     ; copy data at addr held in R4 into R6, then R4+2-->R4
        mov.w   @R4+, R7     ; copy data at addr held in R4 into R7, then R4+2-->R4

        mov.b   @R4+, R8     ; copy data at addr held in R4 into R8, then R4+1-->R4
        mov.b   @R4+, R9     ; copy data at addr held in R4 into R9, then R4+1-->R4
        mov.b   @R4+, R10    ; copy data at addr held in R4 into R10, then R4+1-->R4

        jmp     main

;-------------------------------------------------------------------------------
; Memory Allocation
;-------------------------------------------------------------------------------

        .data       ; allocate vars in data memory
        .retain     ; keep these statements
                    ;    even if not used

Block1: .short   1122h, 3344h, 5566h, 7788h, 99AAh
```

The @ symbol tells the assembler that the register holds the address to use to retrieve the src data. The + symbol says that after the address is used, increment the register.

Here is what these directives do:

Label	Address	Data
Block1	2000h	*1122h*
	2002h	*3344h*
	2004h	*5566h*
	2006h	*7788h*
	2008h	*99AAh*

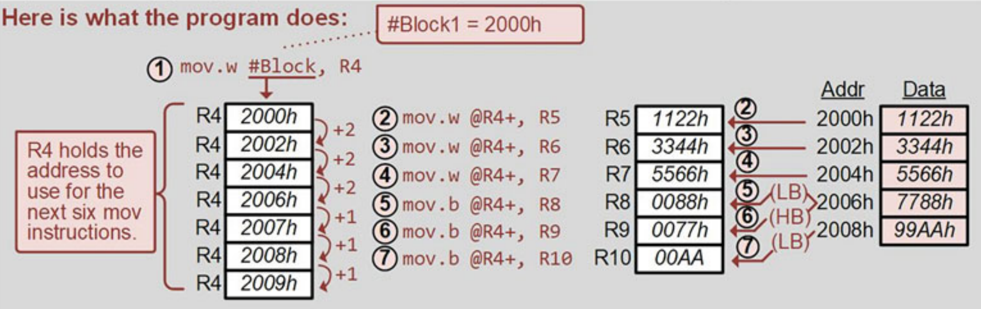
3) Debug your program. Fix errors as needed.

4) Set a breakpoint before the first instruction (**mov.w #Block1, R4**).

5) Open the register viewer and memory browser. Inspect the memory starting at 2000h.

6) Run & step your program while observing the registers and memory.

Here is what the program does:



Example 6.6
Indirect autoincrement mode addressing

Concept Check

CC6.6 Is it possible to accomplish the same functionality as indirect autoincrement mode using indirect register mode? If so, how?

A) Yes. You would simply need to add an increment instruction after the instruction using indirect register mode.

B) No.

6.7 The MOV Instruction with Indexed Mode (X(Rn)) Addressing

Indexed register mode is similar to indirect register mode in that a register name is provided that holds the address of where to access the information. Index register mode extends this functionality by allowing a numeric constant to be added to the contents of the register. This constant is called an *offset*. The offset is a 16-bit, signed number that can be provided in any base. The syntax to indicate indexed register mode is to put the name of the register in parenthesis with the numeric constant in front (i.e., X(R_N)). Indexed addressing is useful when accessing blocks of data in memory where the offset between blocks can be calculated. Indexed mode works in both the src and dst. Follow Example 6.7 to gain experience with indexed mode addressing.

Example: Indexed Mode Addressing X(Rn)

In *indexed mode* addressing, a CPU register name plus a numerical offset is provided to form the address where the information is to be accessed. The register name is provided within a parenthesis and the offset is placed in front of it. Let's look at how this mode works on the LaunchPad^{TM} board. This program copies a block of four words in memory from one location to another location in memory.

1) Create an Empty Assembly-only CCS project titled: **Asm_AddrMode7_Indexed**.

2) Type in the following code into the main.asm file where the comments say "Main loop here".

```
main:
        mov.w   #Block1, R4     ; put the value 2000h into R4 to be used as address

        mov.w   0(R4),  8(R4)   ; copy 1st word in Block1 into 1st word into Block2
        mov.w   2(R4), 10(R4)   ; copy 2nd word in Block1 into 2nd word into Block2
        mov.w   4(R4), 12(R4)   ; copy 3rd word in Block1 into 3rd word into Block2
        mov.w   6(R4), 14(R4)   ; copy 4th word in Block1 into 4th word into Block2

        jmp     main

;-------------------------------------------------------------------------------
; Memory Allocation
;-------------------------------------------------------------------------------

        .data   ; allocate vars in data memory
        .retain ; keep these statements
                ;   even if not used

Block1: .short  0AAAAh, 0BBBBh, 0CCCCh, 0DDDDh
Block2: .space  8
```

A parenthesis around a register name preceded by an offset indicates indexed addressing. The offset is added to the value held in the register to form the operand address.

Here is what these directives do:

Label	Address	Data	Offset Relative to 2000h
Block1	2000h	AAAAh	0
	2002h	BBBBh	+2
	2004h	CCCCh	+4
	2006h	DDDDh	+6
Block2	2008h	-	+8
	200Ah	-	+10
	200Ch	-	+12
	200Eh	-	+14

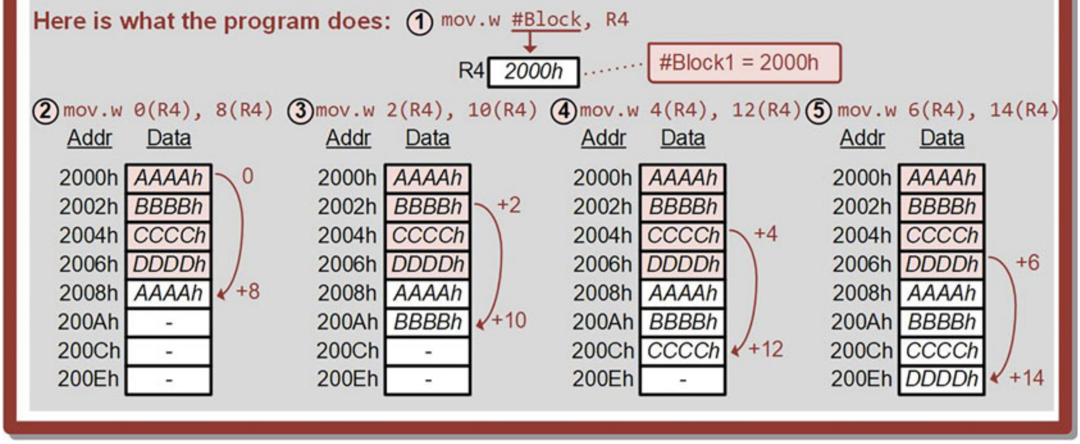
3) Debug your program. Fix errors as needed.

4) Set a breakpoint before the first instruction (**mov.w #Block1, R4**).

5) Open the register viewer and memory browser. Inspect the memory starting at 2000h.

6) Run & step your program while observing the registers and memory.

Here is what the program does:



Addr	② Data	③ Data	④ Data	⑤ Data
2000h	AAAAh (0)	AAAAh	AAAAh	AAAAh
2002h	BBBBh	BBBBh (+2)	BBBBh	BBBBh
2004h	CCCCh	CCCCh	CCCCh (+4)	CCCCh
2006h	DDDDh	DDDDh	DDDDh	DDDDh (+6)
2008h	AAAAh (+8)	AAAAh	AAAAh	AAAAh
200Ah	-	BBBBh (+10)	BBBBh	BBBBh
200Ch	-	-	CCCCh (+12)	CCCCh
200Eh	-	-	-	DDDDh (+14)

Example 6.7
Indexed mode addressing

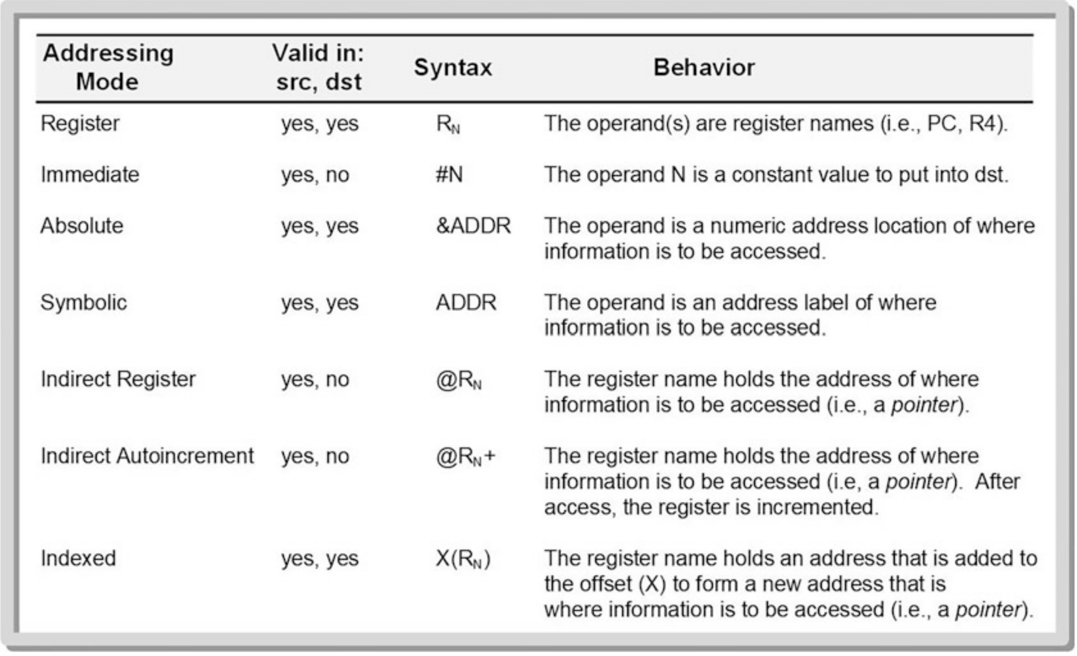
Concept Check

CC6.7 Would it be possible to calculate the offset between two addresses for use in indexed addressing?

A) Yes. You would simply need to subtract the smaller address from the larger address.

B) No. MCUs don't have the ability to perform subtraction. (Hint: MCUs absolutely can perform subtraction.)

Table 6.1 gives a summary of the seven addressing modes used on the MS430.



Addressing Mode	Valid in: src, dst	Syntax	Behavior
Register	yes, yes	R_N	The operand(s) are register names (i.e., PC, R4).
Immediate	yes, no	#N	The operand N is a constant value to put into dst.
Absolute	yes, yes	&ADDR	The operand is a numeric address location of where information is to be accessed.
Symbolic	yes, yes	ADDR	The operand is an address label of where information is to be accessed.
Indirect Register	yes, no	@R_N	The register name holds the address of where information is to be accessed (i.e., a *pointer*).
Indirect Autoincrement	yes, no	@R_N+	The register name holds the address of where information is to be accessed (i.e, a *pointer*). After access, the register is incremented.
Indexed	yes, yes	X(R_N)	The register name holds an address that is added to the offset (X) to form a new address that is where information is to be accessed (i.e., a *pointer*).

Table 6.1
MSP430 addressing mode summary

Summary

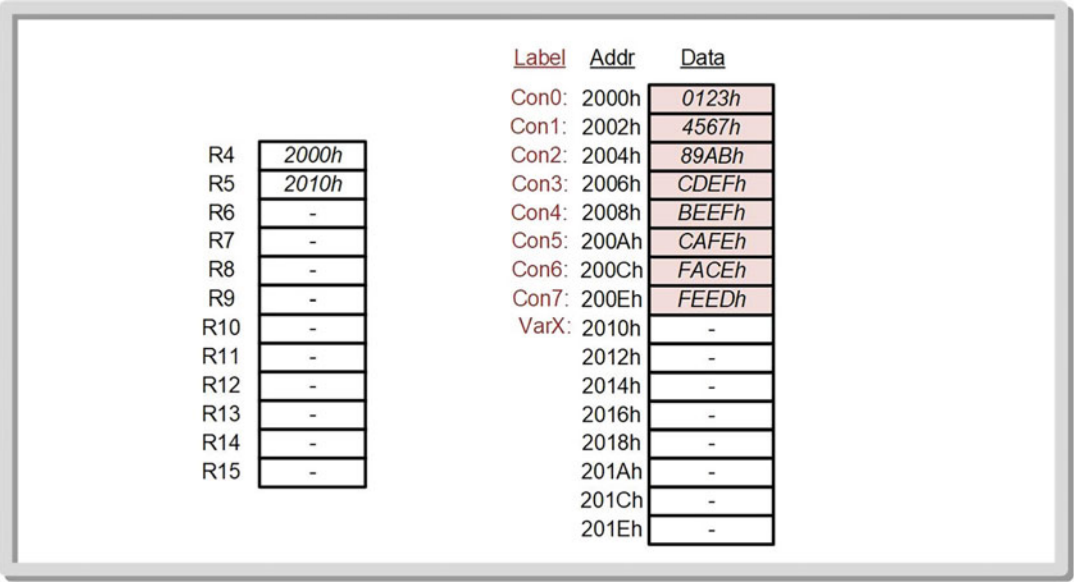
- A move instruction has a mnemonic of **`mov`** and an operand of src, dst. While called a move, this instruction actually copies the information from the src into the dst.
- An addressing mode is the way that the locations of the src and dst are provided for the instruction. The MSP430 provides seven different addressing modes.
- Register mode addressing uses the CPU names for the src and/or dst (i.e., PC, R4, and R5).
- Immediate mode addressing provides a numeric constant for the src. This mode is indicated by preceding the numeric constant with a #.
- Absolute mode addressing uses a numeric address in the src and/or dst. This mode is

indicated by preceding the address with an ampersand (&).

- Symbolic mode addressing uses an address label in the src and/or dst. No special syntax is needed for symbolic mode.
- Indirect register mode addressing uses a CPU register to provide the address of where the information to be accessed resides. This mode is indicated by preceding the register name with an @. This mode is only valid in the src.
- Indirect autoincrement mode works the same as indirect register mode with the additional feature that after memory is accessed, the address register is incremented. This mode is indicated by preceding the register name with an @ and adding a + afterward. This mode is only valid in the src.
- Indexed mode addressing works the same as indirect register mode, except a numeric constant can be applied to the register to form the absolute address. This mode is indicated by placing the address register within parenthesis and a numeric offset before the parenthesis. This mode is valid in both the src and dst.

Exercise Problems

Figure 6.1 provides the initial values of CPU registers and memory that will be used in some of the exercise problems.



R4	*2000h*
R5	*2010h*
R6	-
R7	-
R8	-
R9	-
R10	-
R11	-
R12	-
R13	-
R14	-
R15	-

Label	Addr	Data
Con0:	2000h	*0123h*
Con1:	2002h	*4567h*
Con2:	2004h	*89ABh*
Con3:	2006h	*CDEFh*
Con4:	2008h	*BEEFh*
Con5:	200Ah	*CAFEh*
Con6:	200Ch	*FACEh*
Con7:	200Eh	*FEEDh*
VarX:	2010h	-
	2012h	-
	2014h	-
	2016h	-
	2018h	-
	201Ah	-
	201Ch	-
	201Eh	-

Fig. 6.1
Initial values of CPU registers and memory for exercise problems

Section 6.1: The MOV Instruction with Register Mode (Rn) Addressing

6.1.1 Provide the new value of R8 if the following instruction is executed:

```
mov    R4, R8
```

6.1.2 Provide the new value of R8 if the following instruction is executed:

```
mov.w    R4, R8
```

6.1.3 Provide the new value of R8 if the following instruction is executed:

```
mov.b    R4, R8
```

6.1.4 Provide the new value of R9 if the following instruction is executed:

```
mov    R5, R9
```

6.1.5 Provide the new value of R9 if the following instruction is executed:

```
mov.w    R5, R9
```

6.1.6 Provide the new value of R9 if the following instruction is executed:

```
mov.b    R5, R9
```

6.1.7 Provide the instruction that will move the value in R4 into R6.

6.1.8 Provide the instruction that will move the value in R5 into R7.

6.1.9 Provide the instruction that will move the value in R4 into R8.

6.1.10 Provide the instruction that will move the value in R5 into R9.

Section 6.2: The MOV Instruction with Immediate Mode (#N) Addressing

6.2.1 Provide the new value of R8 if the following instruction is executed:

```
mov.w    #3711h, R8
```

6.2.2 Provide the new value of R8 if the following instruction is executed:

```
mov.w    #0A999h, R8
```

6.2.3 Provide the new value of R8 if the following instruction is executed:

```
mov.b    #67h, R8
```

6.2.4 Provide the new value of R8 if the following instruction is executed:

```
mov.w    #100, R8
```

6.2.5 Provide the new value of R8 if the following instruction is executed:

```
mov.w    #371, R8
```

6.2.6 Provide the instruction that will put the numeric value 1234_{16} into R6.

6.2.7 Provide the instruction that will put the numeric value ABCD_{16} into R7.

6.2.8 Provide the instruction that will put the numeric value 99_{10} into R8.

6.2.9 Provide the instruction that will put the numeric value 1100_2 into R9.

6.2.10 Provide the instruction that will put the numeric value 1100_2 into R10.

Section 6.3: The MOV Instruction with Absolute Mode (&) Addressing

6.3.1 Using the values from Fig. 6.1, provide the new value of R15 if the following instruction is executed:

```
mov    &2000h, R15
```

6.3.2 Using the values from Fig. 6.1, provide the new value of R15 if the following instruction is executed:

```
mov.w    &2002h, R15
```

6.3.3 Using the values from Fig. 6.1, provide the new value of R15 if the following instruction is executed:

```
mov.b    &2004h, R15
```

6.3.4 Using the values from Fig. 6.1, provide the new value of R15 if the following instruction is executed:

```
mov    &200Ah, R15
```

6.3.5 Using the values from Fig. 6.1, provide the new value of R15 if the following instruction is executed:

```
mov.w    &200Ch, R15
```

6.3.6 Using the values from Fig. 6.1, provide the new value of R15 if the following instruction is executed:

```
mov.b    &200Eh, R15
```

6.3.7 Using the values from Fig. 6.1, provide the new value of R15 if the following instruction is executed:

```
mov.b    &2001h, R15
```

6.3.8 Using the values from Fig. 6.1, provide the new value of R15 if the following instruction is executed:

```
mov.b    &200bh, R15
```

6.3.9 Using the values from Fig. 6.1, provide the new value of R15 if the following instruction is executed:

```
mov.w    &2001h, R15
```

6.3.10 Using the values from Fig. 6.1, provide the new value of R15 if the following instruction is executed:

```
mov.w    &200bh, R15
```

Section 6.4: The MOV Instruction with Symbolic Mode Addressing

6.4.1 Using the values from Fig. 6.1, provide the new value of R7 if the following instruction is executed:

```
mov    Con3, R7
```

6.4.2 Using the values from Fig. 6.1, provide the new value of R7 if the following instruction is executed:

```
mov.w    Con7, R7
```

6.4.3 Using the values from Fig. 6.1, provide the new value of R7 if the following instruction is executed:

```
mov.b    Con1, R7
```

6.4.4 Using the values from Fig. 6.1, provide the new value of R7 if the following instruction is executed:

```
mov    Con2, R7
```

6.4.5 Using the values from Fig. 6.1, provide the new value of R7 if the following instruction is executed:

```
mov.w    Con6, R7
```

6.4.6 Using the values from Fig. 6.1, provide the new value of R7 if the following instruction is executed:

```
mov.b    Con0, R7
```

Section 6.5: The MOV Instruction with Indirect Register Mode (@Rn) Addressing

6.5.1 Using the values from Fig. 6.1, provide the new value of R9 if the following instruction is executed:

```
mov    @R4, R9
```

6.5.2 Using the values from Fig. 6.1, provide the new value of R9 if the following instruction is executed:

```
mov.w    @R4, R9
```

6.5.3 Using the values from Fig. 6.1, provide the new value of R9 if the following instruction is executed:

```
mov.b    @R4, R9
```

6.5.4 Using the values from Fig. 6.1, provide the new value of R8 if the following instructions are executed:

```
mov.w    #200Ah, R7
mov.w    @R7, R8
```

6.5.5 Using the values from Fig. 6.1, provide the new value of R8 if the following instructions are executed:

```
mov.w    #200Ah, R7
mov.b    @R7, R8
```

6.5.6 Using the values from Fig. 6.1, provide the new value of R8 if the following instructions are executed:

```
mov.w    #200Ch, R7
mov.w    @R7, R8
```

6.5.7 Using the values from Fig. 6.1, provide the new value of R8 if the following instructions are executed:

```
mov.w    #200Ch, R7
mov.b    @R7, R8
```

6.5.8 Using the values from Fig. 6.1, provide the new value of R8 if the following instructions are executed:

```
mov.w    #200Eh, R7
mov.w    @R7, R8
```

6.5.9 Using the values from Fig. 6.1, provide the new value of R8 if the following instructions are executed:

```
mov.w    #200Eh, R7
mov.b    @R7, R8
```

Section 6.6: The MOV Instruction with Indirect Autoincrement Mode (@Rn+) Addressing

6.6.1 Using the values from Fig. 6.1, provide the new value of R9 if the following instruction is executed:

```
mov.w    @R4+, R9
```

6.6.2 Using the values from Fig. 6.1, provide the new value of R4 after following instruction is executed:

```
mov.w    @R4+, R9
```

6.6.3 Using the values from Fig. 6.1, provide the new value of R9 if the following instruction is executed:

```
mov.b    @R4+, R9
```

6.6.4 Using the values from Fig. 6.1, provide the new value of R4 after following instruction is executed:

```
mov.b    @R4+, R9
```

6.6.5 Using the values from Fig. 6.1, provide the new value of R7 after following instructions are executed:

```
mov.w    @R4+, R5
mov.w    @R4+, R6
mov.w    @R4+, R7
```

6.6.6 Using the values from Fig. 6.1, provide the new value of R4 after following instructions are executed:

```
mov.w    @R4+, R5
mov.w    @R4+, R6
mov.w    @R4+, R7
```

6.6.7 Using the values from Fig. 6.1, provide the new value of R9 after following instructions are executed:

```
mov.w    @R4+, R7
mov.w    @R4+, R8
mov.w    @R4+, R9
```

6.6.8 Using the values from Fig. 6.1, provide the new value of R4 after following instructions are executed:

```
mov.w    @R4+, R5
mov.w    @R4+, R6
mov.w    @R4+, R7
```

6.6.9 Using the values from Fig. 6.1, provide the new value of R7 after following instructions are executed:

```
mov.b    @R4+, R5
mov.b    @R4+, R6
mov.b    @R4+, R7
```

6.6.10 Using the values from Fig. 6.1, provide the new value of R4 after following instructions are executed:

```
mov.b    @R4+, R5
mov.b    @R4+, R6
mov.b    @R4+, R7
```

6.6.11 Using the values from Fig. 6.1, provide the new value of R9 after following instructions are executed:

```
mov.b    @R4+, R7
mov.b    @R4+, R8
mov.b    @R4+, R9
```

6.6.12 Using the values from Fig. 6.1, provide the new value of R4 after following instructions are executed:

```
mov.b    @R4+, R5
mov.b    @R4+, R6
mov.b    @R4+, R7
```

Section 6.7: The MOV Instruction with Indexed Mode (X(Rn)) Addressing

6.7.1 Using the values from Fig. 6.1, provide the new value of R7 after following instruction is executed:

```
mov.w    2(R4),R7
```

6.7.2 Using the values from Fig. 6.1, provide the new value of R7 after following instruction is executed:

```
mov.w    6(R4),R7
```

6.7.3 Using the values from Fig. 6.1, provide the new value of R7 after following instruction is executed:

```
mov.w    4(R4),R7
```

6.7.4 Using the values from Fig. 6.1, provide the new value of R7 after following instruction is executed:

```
mov.w    0(R4),R7
```

6.7.5 Using the values from Fig. 6.1, provide the new value at label VarX after following instruction is executed:

```
mov.w    12(R4),0(R5)
```

6.7.6 Using the values from Fig. 6.1, provide the new value at label VarX after following instruction is executed:

```
mov.w    10(R4),0(R5)
```

6.7.7 Using the values from Fig. 6.1, provide the new value at label VarX after following instruction is executed:

```
mov.w    14(R4),0(R5)
```

6.7.8 Using the values from Fig. 6.1, provide the new value at label VarX after following instruction is executed:

```
mov.w    8(R4),0(R5)
```

6.7.9 Using the values from Fig. 6.1, provide the new value at label VarX after following instruction is executed:

```
mov.w    4(R4),16(R4)
```

6.7.10 Using the values from Fig. 6.1, provide the new value at label VarX after following instruction is executed:

```
mov.w    2(R4),16(R4)
```

Chapter 7: Data Manipulation Instructions

This chapter introduces the MSP430 data manipulation instructions performed by the ALU and the impact on the corresponding status bits in the status register [1]. Examples are provided for some of the data manipulation instructions to help in understanding how ALU operations work and how the SR is updated. It is intended that the reader is coding the examples using the MSP430FR2355 LaunchPad™ board as they go through this chapter.

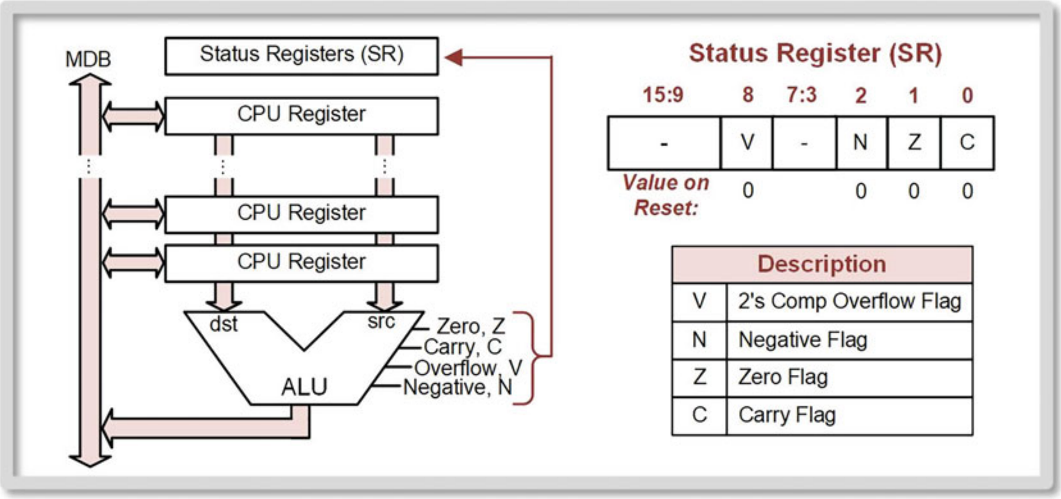
Learning Outcomes—After completing this chapter you will be able to:

7.1 Use arithmetic instructions to manipulate data within the CPU and explain how the status flags are altered.
7.2 Use logic instructions to manipulate data within the CPU and explain how the status flags are altered.
7.3 Use bit set and bit clear instructions to set and clear individual bits within an operand.
7.4 Use test instructions to determine information about an operand from the status bits.
7.5 Explain the operation of rotate arithmetic and rotate through carry instructions.

7.1 Arithmetic Instructions

The ALU can be thought of as a collection of combinational logic circuits, each that can perform a desired operation on the data coming from CPU registers. For every operation that is desired when designing the CPU, a new circuit is inserted into the ALU. Every effort is taken to try to optimize the amount of logic in the ALU, but conceptually, instructions that use the ALU can be thought of as accessing separate circuits. The output of the ALU is not registered, so its output must be put back into either a CPU register or memory. A key feature of the ALU is that it has circuits that monitor the operations and produce status flags (or status bits). These flags are stored back into the status register in the CPU and then can potentially be used by subsequent instructions. The flags are two's complement overflow (V), negative (N), zero (Z), and carry (C). Each of these flags is asserted when the condition exists. The carry flag can also be used to indicate a *borrow* when performing subtraction. Figure 7.1 shows a conceptual model of the ALU and SR.

B. J. LaMeres, *Embedded Systems Design using the MSP430FR2355 LaunchPad™*,
https://doi.org/10.1007/978-3-031-20888-1_7



Status Register (SR)

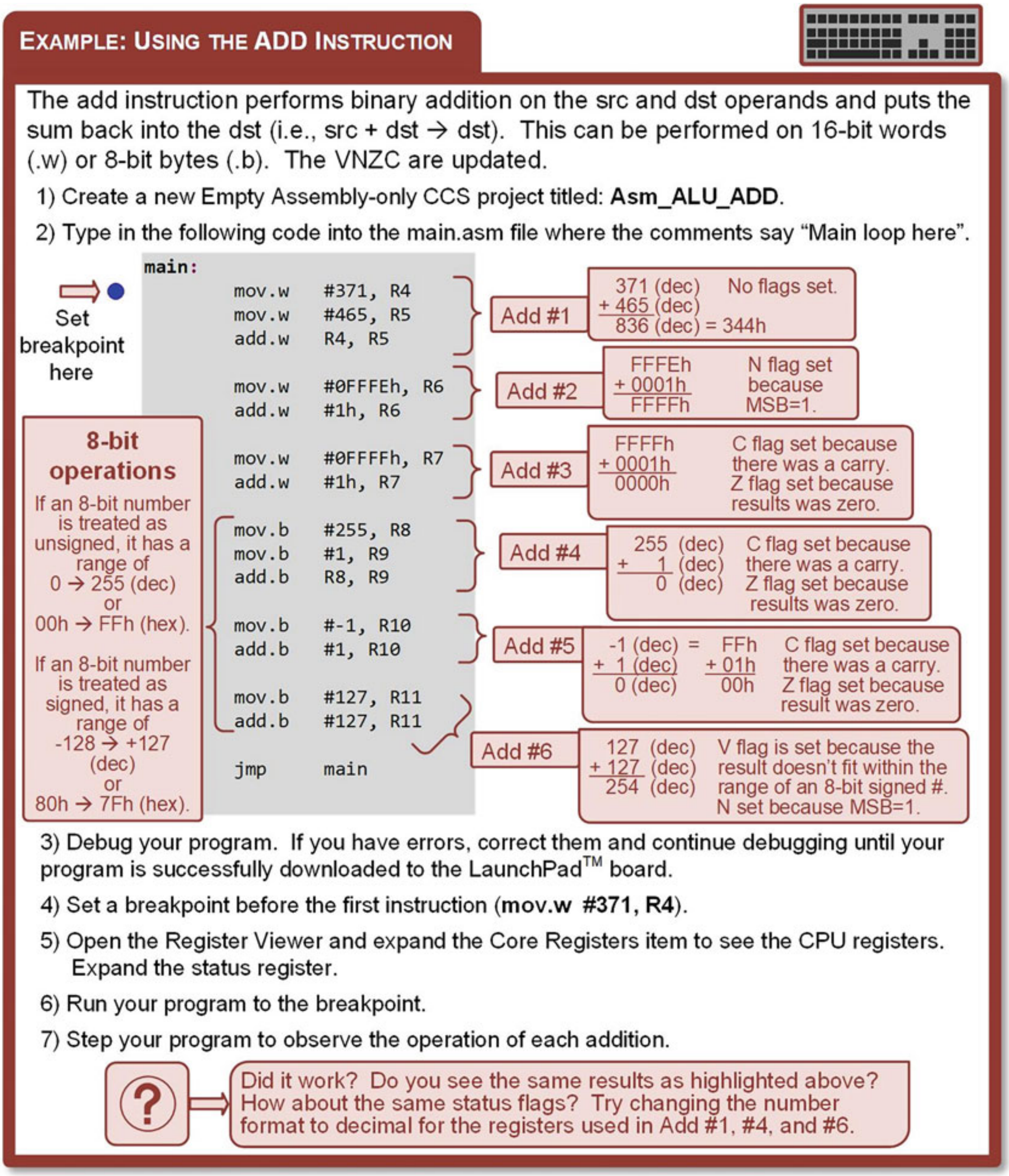
15:9	8	7:3	2	1	0
-	V	-	N	Z	C
Value on Reset:	0		0	0	0

Description	
V	2's Comp Overflow Flag
N	Negative Flag
Z	Zero Flag
C	Carry Flag

Fig. 7.1
ALU and status register operation

7.1.1 Addition Instructions

The **`add`** instruction performs binary addition on two inputs, the src and dst, and puts the sum back into the dst (i.e., src + dst → dst). The four status flags are updated in this operation. This operation can be performed on both 8-bit and 16-bit words by appending .w or .b. This operation works the same regardless of whether the src or dst is treated as unsigned or signed numbers. Follow Example 7.1 to gain some experience using the **`add`** instruction.



Example: Using the ADD Instruction

The add instruction performs binary addition on the src and dst operands and puts the sum back into the dst (i.e., src + dst → dst). This can be performed on 16-bit words (.w) or 8-bit bytes (.b). The VNZC are updated.

1) Create a new Empty Assembly-only CCS project titled: **Asm_ALU_ADD**.

2) Type in the following code into the main.asm file where the comments say "Main loop here".

```
main:
            mov.w   #371, R4
            mov.w   #465, R5
            add.w   R4, R5

            mov.w   #0FFFEh, R6
            add.w   #1h, R6

            mov.w   #0FFFFh, R7
            add.w   #1h, R7

            mov.b   #255, R8
            mov.b   #1, R9
            add.b   R8, R9

            mov.b   #-1, R10
            add.b   #1, R10

            mov.b   #127, R11
            add.b   #127, R11

            jmp     main
```

Set breakpoint here

8-bit operations

If an 8-bit number is treated as unsigned, it has a range of 0 → 255 (dec) or 00h → FFh (hex).

If an 8-bit number is treated as signed, it has a range of -128 → +127 (dec) or 80h → 7Fh (hex).

Add #1: 371 (dec) + 465 (dec) = 836 (dec) = 344h. No flags set.

Add #2: FFFEh + 0001h = FFFFh. N flag set because MSB=1.

Add #3: FFFFh + 0001h = 0000h. C flag set because there was a carry. Z flag set because results was zero.

Add #4: 255 (dec) + 1 (dec) = 0 (dec). C flag set because there was a carry. Z flag set because results was zero.

Add #5: -1 (dec) + 1 (dec) = 0 (dec); FFh + 01h = 00h. C flag set because there was a carry. Z flag set because result was zero.

Add #6: 127 (dec) + 127 (dec) = 254 (dec). V flag is set because the result doesn't fit within the range of an 8-bit signed #. N set because MSB=1.

3) Debug your program. If you have errors, correct them and continue debugging until your program is successfully downloaded to the LaunchPad™ board.

4) Set a breakpoint before the first instruction (**mov.w #371, R4**).

5) Open the Register Viewer and expand the Core Registers item to see the CPU registers. Expand the status register.

6) Run your program to the breakpoint.

7) Step your program to observe the operation of each addition.

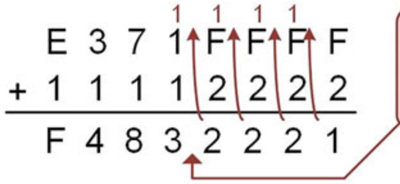
? Did it work? Do you see the same results as highlighted above? How about the same status flags? Try changing the number format to decimal for the registers used in Add #1, #4, and #6.

Example 7.1
Using the ADD instruction

One of the limitations of the add instruction is that it only operates on 16-bit words. When numbers are larger than 16-bit words, a different algorithm must be used. This is accomplished by adding the lower 16-bit words of the input numbers first using add, and then including the carry that was potentially generated in the addition of the next upper 16-bit words of the inputs. The addition of higher order words that include the carry from prior additions is repeated until all of the bits in the inputs have been added. The **addc** instruction gives us the functionality to include the carry in the addition. This instruction performs src + dst + C → dst. Follow Example 7.2 to see how numbers larger than 16 bits can be added together with the functionality provided by the **addc** instruction.

Example: Using the ADDC Instruction

The addc instruction performs binary addition on the src and dst operands, but also adds in the carry bit from the status register. The sum is put back into the dst (i.e., src + dst + C → dst). This instruction is useful when trying to perform addition on numbers that are larger than 16-bits. Let's consider why this is needed when adding the 32-bit words E371FFFF + 11112222h.



The lower 16-bit word addition results in a carry that must be considered in the upper 16-bit word addition. To perform this operation, we can use add on the lower words and addc on the upper words.

Let's see how this looks in assembly.

1) Create a new Empty Assembly-only CCS project titled: **Asm_ALU_ADDC**.

2) Type in the following code into the main.asm file where the comments say "Main loop here".

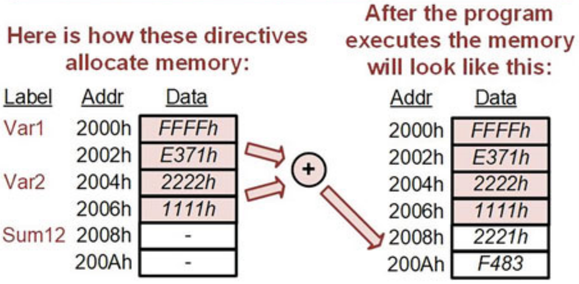
BP

```
main:
        mov.w   #Var1, R4
        mov.w   #Var2, R5
        mov.w   #Sum12, R6

        mov.w   0(R4), R7
        mov.w   0(R5), R8
        add.w   R7, R8
        mov.w   R8, 0(R6)

        mov.w   2(R4), R7
        mov.w   2(R5), R8
        addc.w  R7, R8
        mov.w   R8, 2(R6)

        jmp     main

;-------------------------
; Memory Allocation
;-------------------------
        .data
        .retain

Var1:   .long   0E371FFFFh
Var2:   .long   11112222h

Sum12:  .space  4
```

First, let's load registers with the addresses of where the data that we want to add is located and where the sum will go. This will allow us to use indexed mode addressing to access the information.

Now we add the lower 16-bit words of the 32-bit inputs using the add instruction. Keep in mind that the MSP430 puts the lower 16-bits of the 32-bit word in memory before the upper 16-bits. See below for a graphical depiction. That is why our offset is "0".

Now we add the upper 16-bit words of the 32-bit inputs using the addc instruction. This addition considers the carry bit from the prior addition. Note that the mov instructions don't alter the status register flags, so the carry from the prior add is preserved.



Here is how these directives allocate memory:

Label	Addr	Data
Var1	2000h	FFFFh
	2002h	E371h
Var2	2004h	2222h
	2006h	1111h
Sum12	2008h	-
	200Ah	-

After the program executes the memory will look like this:

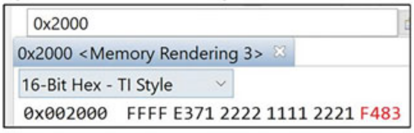
Addr	Data
2000h	FFFFh
2002h	E371h
2004h	2222h
2006h	1111h
2008h	2221h
200Ah	F483

3) Debug your program. Set a breakpoint before the first instruction (**mov.w #Var1, R4**).

4) Open the Register Viewer and expand the Core Registers item to see the CPU registers. Expand the status register. Also go to address 0x2000 in the Memory Browser.

5) Run your program to the breakpoint. Step your program to observe its operation.

Did it work? Does your memory browser look like this after execution? Viewing the memory browser can help understand how we used indexed addressing.

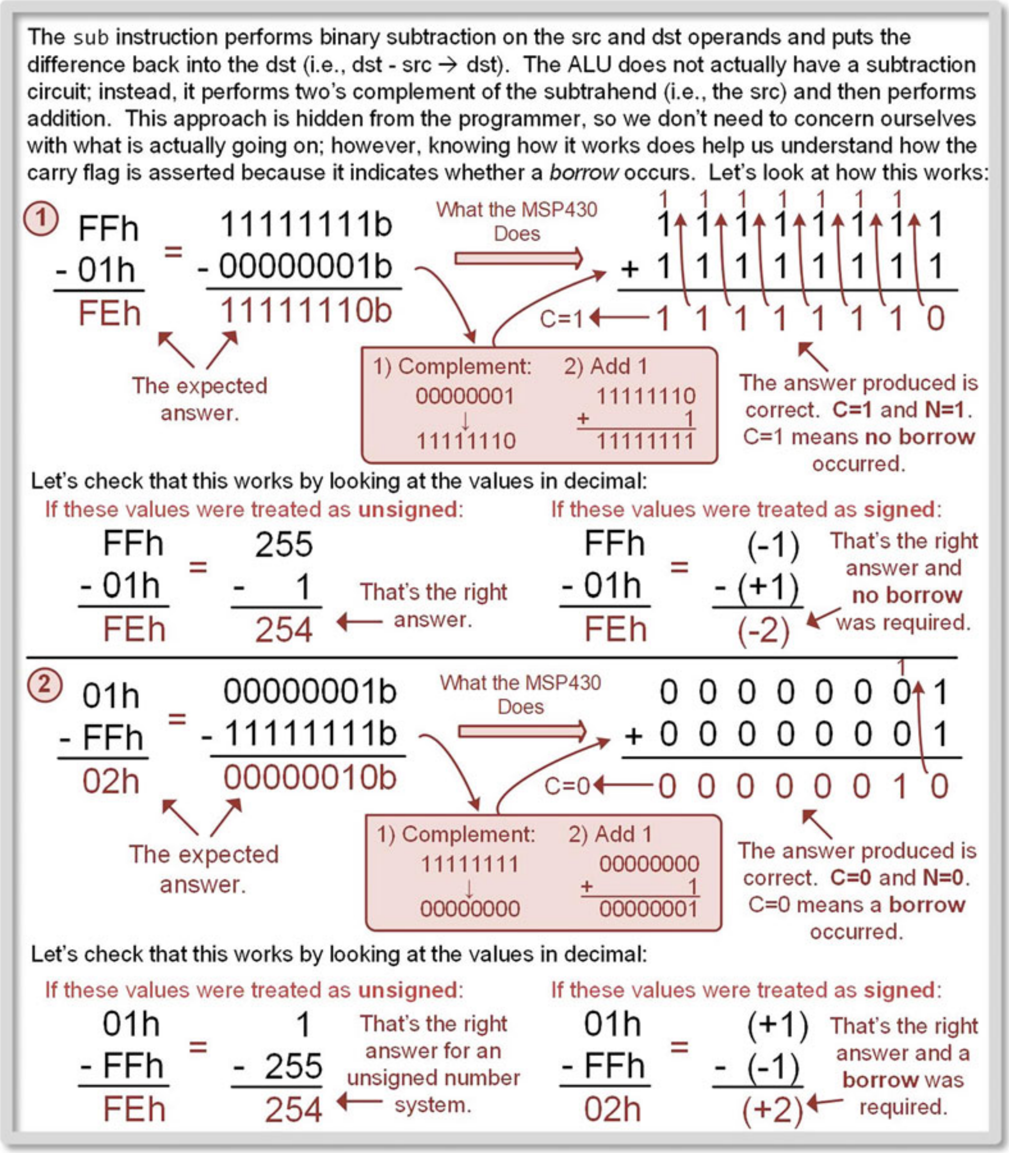


```
0x002000   FFFF E371 2222 1111 2221 F483
```

Example 7.2
Using the ADDC instruction

7.1.2 Subtraction Instructions

The **sub** instruction performs binary addition on two operands and puts the difference back into the dst (i.e., dst – src → dst). The four status flags are updated. This operation can be performed on both 8-bit and 16-bit words by appending .w or .b. This operation works the same regardless of whether the src or dst is treated as unsigned or signed numbers. The MSP430 does not actually contain a subtraction circuit; instead, it converts the src into its negative equivalent by performing two's complement on it, and then it adds the src to the dst. Taking advantage of A−B = A+(−B) allows the ALU circuitry within the MSP430 to be minimized. This approach is hidden from the end user, but understanding it helps us understand how the carry flag is asserted. When C = 1, then no borrow was required for the subtraction, while when C = 0, a borrow was required. Figure 7.2 gives a description of how the subtraction works and how the C-flag is asserted.



The sub instruction performs binary subtraction on the src and dst operands and puts the difference back into the dst (i.e., dst - src → dst). The ALU does not actually have a subtraction circuit; instead, it performs two's complement of the subtrahend (i.e., the src) and then performs addition. This approach is hidden from the programmer, so we don't need to concern ourselves with what is actually going on; however, knowing how it works does help us understand how the carry flag is asserted because it indicates whether a *borrow* occurs. Let's look at how this works:

The answer produced is correct. **C=1** and **N=1**. C=1 means **no borrow** occurred.

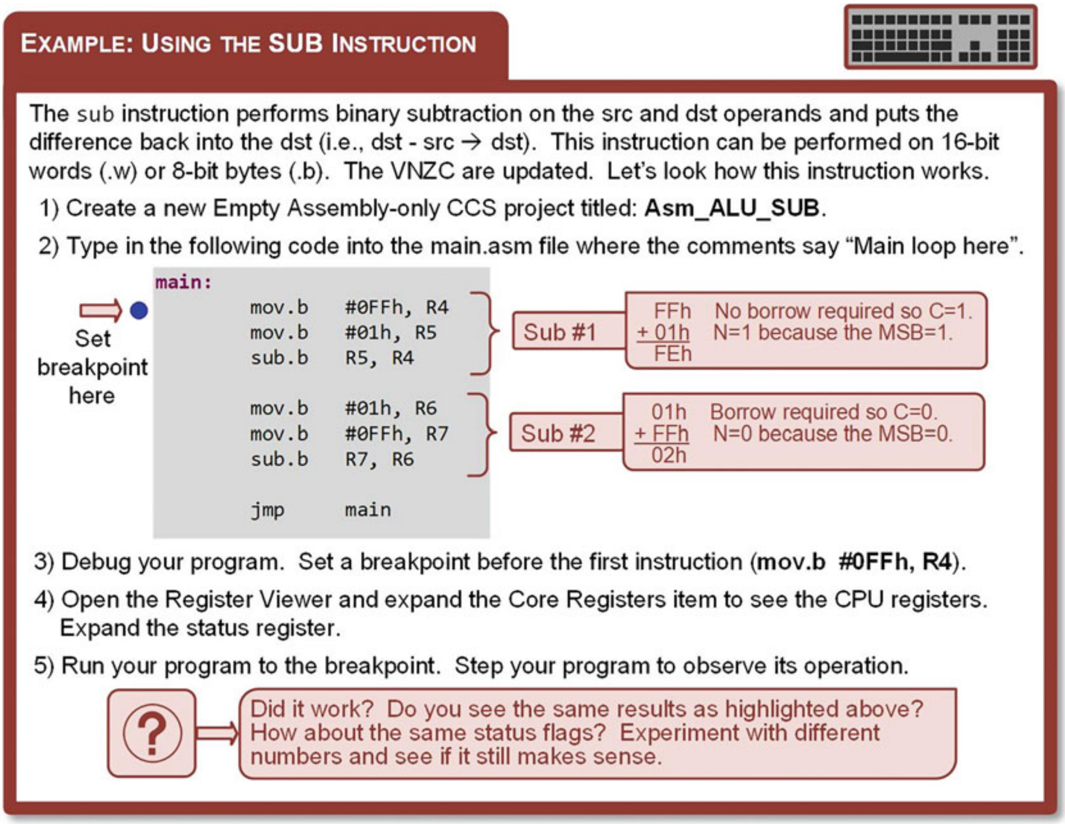
Let's check that this works by looking at the values in decimal:

The answer produced is correct. **C=0** and **N=0**. C=0 means a **borrow** occurred.

Let's check that this works by looking at the values in decimal:

Fig. 7.2
Explanation of subtraction and the C-flag

Now let's see if this works in assembly. Follow Example 7.3 to view how the **sub** instruction works and how the C-flag is asserted.



EXAMPLE: USING THE SUB INSTRUCTION

The sub instruction performs binary subtraction on the src and dst operands and puts the difference back into the dst (i.e., dst - src → dst). This instruction can be performed on 16-bit words (.w) or 8-bit bytes (.b). The VNZC are updated. Let's look how this instruction works.

1) Create a new Empty Assembly-only CCS project titled: **Asm_ALU_SUB**.

2) Type in the following code into the main.asm file where the comments say "Main loop here".

```
main:
        mov.b   #0FFh, R4
        mov.b   #01h, R5
        sub.b   R5, R4

        mov.b   #01h, R6
        mov.b   #0FFh, R7
        sub.b   R7, R6

        jmp     main
```

Set breakpoint here

Sub #1

$$\begin{array}{r} FFh \\ \underline{+\,01h} \\ FEh \end{array}$$

No borrow required so C=1.
N=1 because the MSB=1.

Sub #2

$$\begin{array}{r} 01h \\ \underline{+\,FFh} \\ 02h \end{array}$$

Borrow required so C=0.
N=0 because the MSB=0.

3) Debug your program. Set a breakpoint before the first instruction (**mov.b #0FFh, R4**).

4) Open the Register Viewer and expand the Core Registers item to see the CPU registers. Expand the status register.

5) Run your program to the breakpoint. Step your program to observe its operation.

?

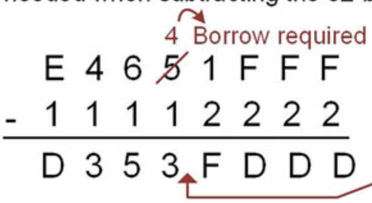
Did it work? Do you see the same results as highlighted above? How about the same status flags? Experiment with different numbers and see if it still makes sense.

Example 7.3
Using the SUB instruction

The **subc** instruction is used when subtracting numbers that are larger than 16-bit words. This is accomplished by subtracting the lower 16-bit words of the input numbers first using **sub** and then including the borrow that was potentially generated in the subtraction of the next upper 16-bit words of the inputs. The subtraction of higher-order words that include the borrow from prior subtractions is repeated until all of the bits in the inputs have been subtracted. The MSP430 tracks borrows using the C-flag. The logic is if C = 0, then a borrow occurred. If C = 1, then no borrow occurred. See Fig. 7.2 for an explanation of this logic. The description of the **subc** instruction is dst – src – not(C) → dst. Follow Example 7.4 to see how numbers larger than 16 bits can be subtracted with the functionality provided by the **subc** instruction.

Example: Using the SUBC Instruction

The `subc` instruction performs binary subtraction on the src and dst operands, but also subtracts not(C) from the status register that may have occurred from a prior subtraction (i.e., dst – src - not(C) → dst). This instruction is useful when trying to perform subtraction on numbers that are larger than 16-bits. Let's consider why this is needed when subtracting the 32-bit words E4651FFF + 11112222h.



The lower 16-bit word subtraction requires a borrow, which is indicated by C=0. This borrow must be considered in the upper 16-bit word subtraction. To perform this operation, we can use **sub** on the lower words and **subc** on the upper words.

Let's see how this looks in assembly.

1) Create a new Empty Assembly-only CCS project titled: **Asm_ALU_SUBC**.

2) Type in the following code into the main.asm file where the comments say "Main loop here".

```
main:
●BP         mov.w    #Var1, R4
            mov.w    #Var2, R5
            mov.w    #Diff12, R6

            mov.w    0(R4), R7
            mov.w    0(R5),  R8
            sub.w    R8, R7
            mov.w    R7, 0(R6)

            mov.w    2(R4), R7
            mov.w    2(R5),  R8
            subc.w   R8, R7
            mov.w    R7, 2(R6)

            jmp      main

;-------------------------
; Memory Allocation
;-------------------------
            .data
            .retain

Var1:       .long    0E4651FFFh
Var2:       .long    11112222h

Diff12:     .space   4
```

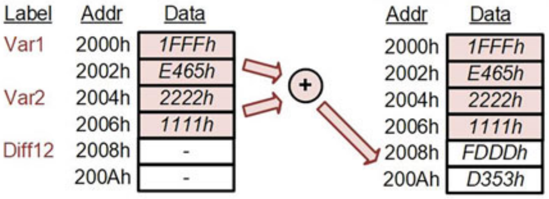
First, let's load registers with the addresses of where the data that we want to subtract is located and where the difference will go. This will allow us to use indexed mode addressing to access the information.

Now we subtract the lower 16-bit words of the 32-bit inputs using the **sub** instruction. Keep in mind that the MSP430 puts the lower 16-bits of the 32-bit word in memory before the upper 16-bits. See below for a graphical depiction. That is why our offset is "0".

Now subtract the upper 16-bit words of the 32-bit inputs using **subc**. This subtraction considers the borrow from the prior subtraction (C=0). Note that the mov instructions don't alter the status register flags, so the borrow from the prior **sub** is preserved.

Here is how these directives allocate memory:

After the program executes the memory will like this:



Label	Addr	Data
Var1	2000h	*1FFFh*
	2002h	*E465h*
Var2	2004h	*2222h*
	2006h	*1111h*
Diff12	2008h	-
	200Ah	-

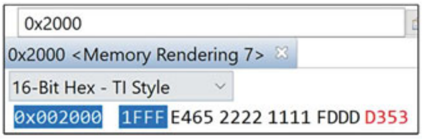
Addr	Data
2000h	*1FFFh*
2002h	*E465h*
2004h	*2222h*
2006h	*1111h*
2008h	*FDDDh*
200Ah	*D353h*

3) Debug your program. Set a breakpoint before the first instruction (**mov.w #Var1, R4**).

4) Open the Register Viewer and expand the Core Registers item to see the CPU registers. Expand the status register. Also go to address 0x2000 in the Memory Browser.

5) Run your program to the breakpoint. Step your program to observe its operation.

Did it work? Does your memory browser look like this after execution? Can you see C=0 from the sub instruction being used in the subc instruction?

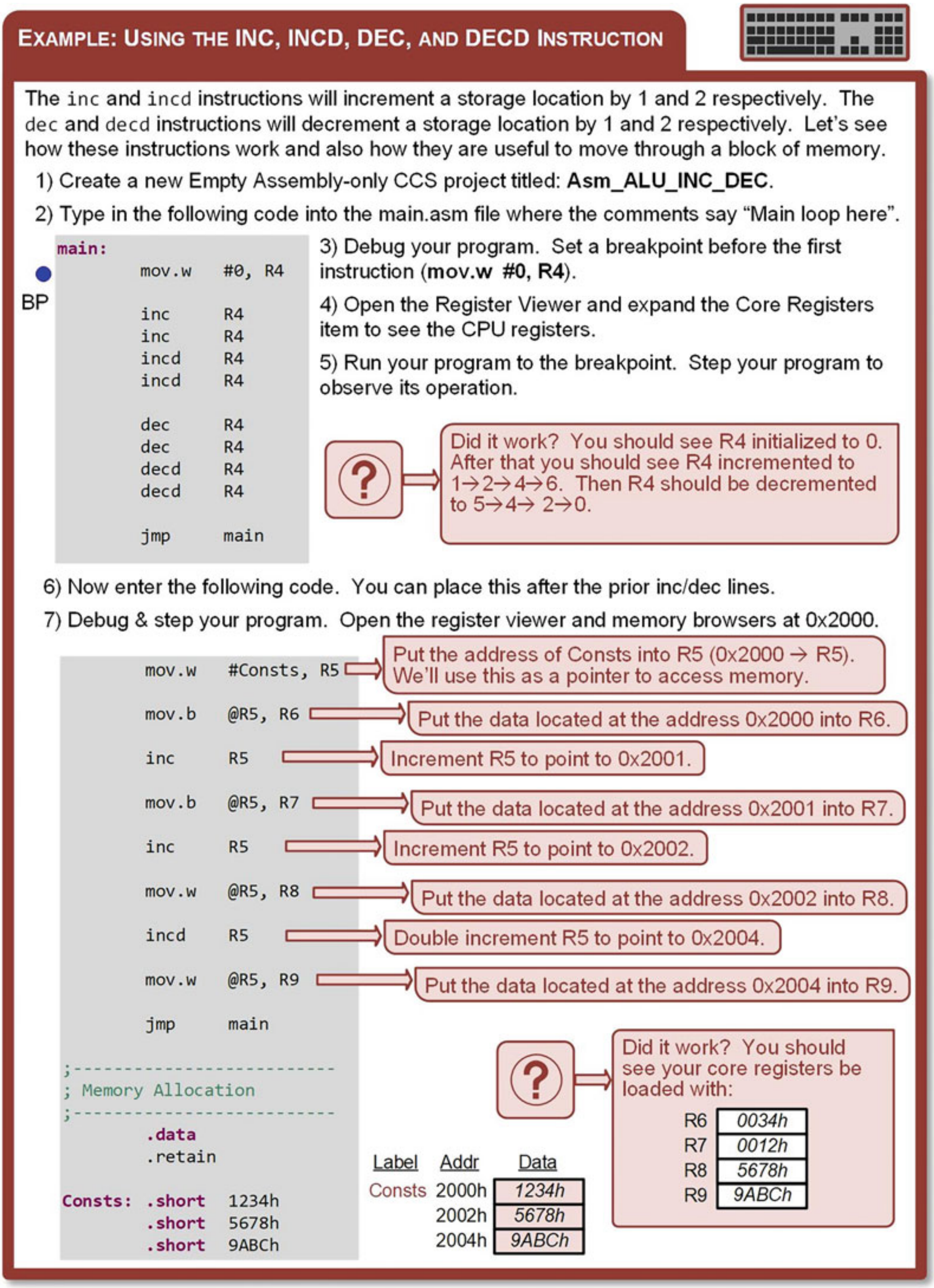


```
0x002000  1FFF E465 2222 1111 FDDD D353
```

Example 7.4
Using the SUBC instruction

7.1.3 Increments and Decrements

The **inc** and **incd** instruction will increment a storage location by 1 and 2, respectively. The **dec** and **decd** will decrement a storage location by 1 and 2, respectively. Follow Example 7.5 to see how these instructions work and also how they are useful for moving through a block of memory.



EXAMPLE: USING THE INC, INCD, DEC, AND DECD INSTRUCTION

The inc and incd instructions will increment a storage location by 1 and 2 respectively. The dec and decd instructions will decrement a storage location by 1 and 2 respectively. Let's see how these instructions work and also how they are useful to move through a block of memory.

1) Create a new Empty Assembly-only CCS project titled: **Asm_ALU_INC_DEC**.

2) Type in the following code into the main.asm file where the comments say "Main loop here".

```
main:
●           mov.w   #0, R4
BP
            inc     R4
            inc     R4
            incd    R4
            incd    R4

            dec     R4
            dec     R4
            decd    R4
            decd    R4

            jmp     main
```

3) Debug your program. Set a breakpoint before the first instruction (**mov.w #0, R4**).

4) Open the Register Viewer and expand the Core Registers item to see the CPU registers.

5) Run your program to the breakpoint. Step your program to observe its operation.

Did it work? You should see R4 initialized to 0. After that you should see R4 incremented to 1→2→4→6. Then R4 should be decremented to 5→4→ 2→0.

6) Now enter the following code. You can place this after the prior inc/dec lines.

7) Debug & step your program. Open the register viewer and memory browsers at 0x2000.

```
            mov.w   #Consts, R5
            mov.b   @R5, R6
            inc     R5
            mov.b   @R5, R7
            inc     R5
            mov.w   @R5, R8
            incd    R5
            mov.w   @R5, R9
            jmp     main

;-------------------------
; Memory Allocation
;-------------------------
            .data
            .retain

Consts:     .short  1234h
            .short  5678h
            .short  9ABCh
```

Put the address of Consts into R5 (0x2000 → R5). We'll use this as a pointer to access memory.

Put the data located at the address 0x2000 into R6.

Increment R5 to point to 0x2001.

Put the data located at the address 0x2001 into R7.

Increment R5 to point to 0x2002.

Put the data located at the address 0x2002 into R8.

Double increment R5 to point to 0x2004.

Put the data located at the address 0x2004 into R9.

Label	Addr	Data
Consts	2000h	1234h
	2002h	5678h
	2004h	9ABCh

Did it work? You should see your core registers be loaded with:

R6	0034h
R7	0012h
R8	5678h
R9	9ABCh

Example 7.5
Using the INC, INCD, DEC, and DECD instructions

Concept Check

CC7.1 When incrementing through 16-bit words of data in memory, which increment would be best suited for updating a register holding the address of the data?

A) INC. We want to step through each piece of information in memory one by one.

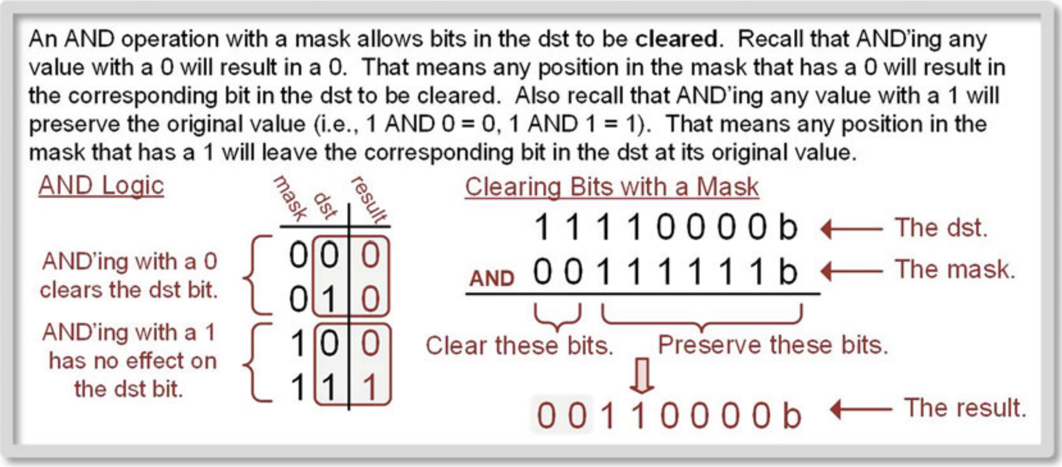
B) INCD. Since 16-bit words are aligned to even addresses in the MSP430 memory system, we need to increment by 2 if we want to access the data 16-bits at a time.

7.2 Logic Instructions

The MSP430 provides three basic logic operations: **`inv`**, **`and`**, and **`xor`**. The CCS environment also supports the mnemonic `or`, which it will substitute with the **bis** instruction (next section). The **`inv`** instruction performs an inversion on each bit of the operand (!dst → dst). The **`and`** operation performs a logical AND on the two operands and places the result back into the dst (src AND dst → dst). The **`or`** operation performs a logical OR on the two operands and places the result back into the dst (src OR dst → dst). The **`xor`** operation performs an exclusive-or operation on the src and dst and places the result back into the dst (src XOR dst → dst). These operations are called *bitwise* operations because they take place on the individual bits of the src and dst independent of each other. For example, for an **`and`** operation, bit 0 of the src will be AND'd with bit 0 of the dst, bit 1 of the src will be and'd with bit 1 of the dst, etc.

With these logic operations, we are now able to accomplish *bit masking*. Bit masking refers to the act of setting, clearing, toggling, or testing the value of a bit within a word. Bit masking is such a common task within an MCU that the MSP430 contains dedicated instructions to perform these tasks. Understanding the logic behind those instructions is useful, so we will look at bit masking using the **`and`**, **`or`**, and **`xor`** instructions.

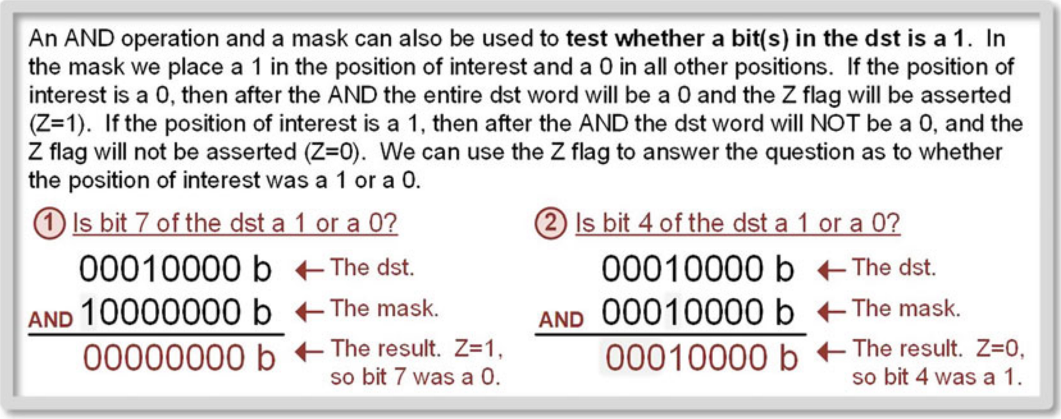
Let's start with what we can do with the **`and`** instruction. A logical AND will produce a 1 only when both inputs are a 1; otherwise it will produce a 0. Thus, we can use the AND operation with a 0 to clear any bit within a word. A *mask* is a bit pattern that is used to indicate which bit we wish to manipulate. When using a mask with an AND, the mask indicates which bit(s) to clear and which to preserve. Any position within the mask that contains a 0 will result in the corresponding bit in the dst being cleared. Any position within the mask that contains a 1 will simply leave the original value within the dst. Figure 7.3 shows a graphical depiction of using the AND operation and a mask to clear bits in the dst.



An AND operation with a mask allows bits in the dst to be **cleared**. Recall that AND'ing any value with a 0 will result in a 0. That means any position in the mask that has a 0 will result in the corresponding bit in the dst to be cleared. Also recall that AND'ing any value with a 1 will preserve the original value (i.e., 1 AND 0 = 0, 1 AND 1 = 1). That means any position in the mask that has a 1 will leave the corresponding bit in the dst at its original value.

Fig. 7.3
Using an AND operation and a mask to clear bits

An AND operation with a mask can also check whether a particular bit(s) is a 1. This is accomplished by using a mask with a 1 in the position of interest and 0 s everywhere else. The 0 s of the mask will clear all bits except the position of interest. If the position of interest is a 0, then the entire dst word will be a 0 and the Z-flag will be asserted (Z = 1). If the position of interest is a 1, then the entire dst word will NOT be a 0, and the Z-flag will not be asserted (Z = 0). We can use the Z-flag to answer the question as to whether the position of interest was a 1 or a 0. Figure 7.4 shows a graphical depiction of using the AND operation and a mask to test whether a particular bit is a 0 in the dst.



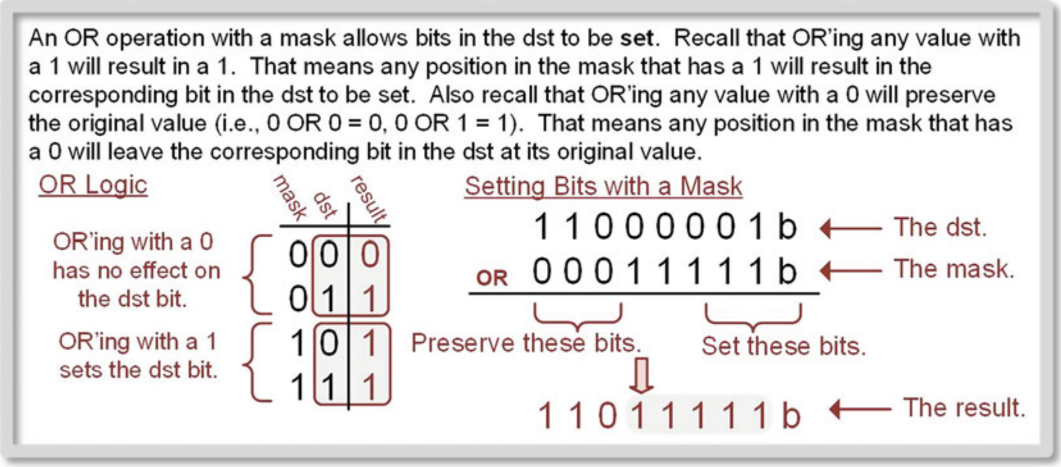
An AND operation and a mask can also be used to **test whether a bit(s) in the dst is a 1**. In the mask we place a 1 in the position of interest and a 0 in all other positions. If the position of interest is a 0, then after the AND the entire dst word will be a 0 and the Z flag will be asserted (Z=1). If the position of interest is a 1, then after the AND the dst word will NOT be a 0, and the Z flag will not be asserted (Z=0). We can use the Z flag to answer the question as to whether the position of interest was a 1 or a 0.

1 Is bit 7 of the dst a 1 or a 0?
00010000 b
The dst.
AND 10000000 b
The mask.
00000000 b
The result. Z=1, so bit 7 was a 0.
2 Is bit 4 of the dst a 1 or a 0?
00010000 b
The dst.
AND 00010000 b
The mask.
00010000 b
The result. Z=0, so bit 4 was a 1.

Fig. 7.4
Using an AND operation and a mask to test whether bits are 1 s

If we wish to test whether a bit in the dst is a 0, we can first complement the dst using the **inv** instruction and then use the same AND operation and mask as in Fig. 7.4.

Now let's see what type of masking we can do with the **or** instruction. A logical OR will produce a 1 when either of the bits of the inputs is a 1. OR'ing with a 0 preserves the original value of the dst. Thus, we can use the OR operation with a 1 to set any bit within a word. When using a mask with an OR, the mask indicates which bit(s) to set and which to preserve. Any position within the mask that contains a 1 will result in the corresponding bit in the dst being set. Any position within the mask that contains a 0 will simply leave the original value within the dst. Figure 7.5 shows a graphical depiction of using the OR operation and a mask to set bits in the dst.

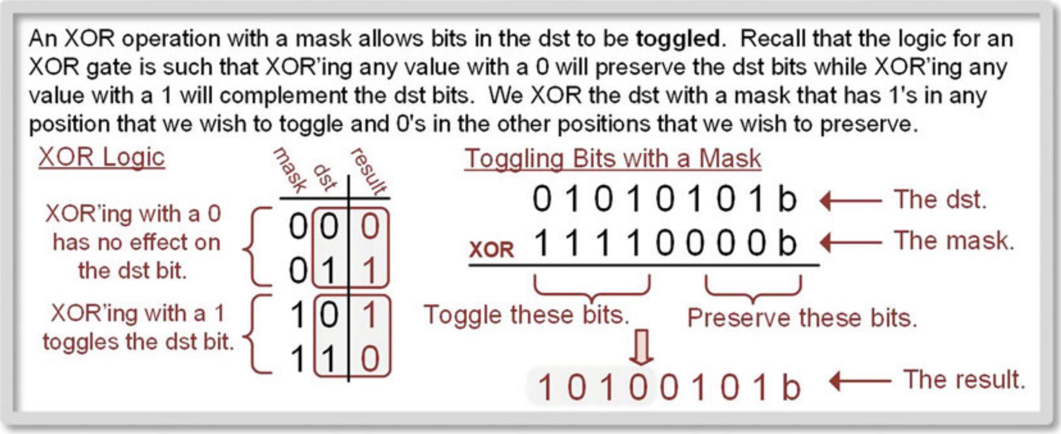


An OR operation with a mask allows bits in the dst to be **set**. Recall that OR'ing any value with a 1 will result in a 1. That means any position in the mask that has a 1 will result in the corresponding bit in the dst to be set. Also recall that OR'ing any value with a 0 will preserve the original value (i.e., 0 OR 0 = 0, 0 OR 1 = 1). That means any position in the mask that has a 0 will leave the corresponding bit in the dst at its original value.

mask	dst	result
0	0	0
0	1	1
1	0	1
1	1	1

Fig. 7.5
Using an OR operation and a mask to set bits

The **xor** instruction can be used to toggle bits within the dst. When XOR'ing a dst bit with a 0, the dst bit will remain unchanged. When XOR'ing a dst bit with a 1, the dst bit will be complemented. Thus, if we wish to toggle bit(s) in the dst, we simply XOR it with a mask where any bits to be toggled contain a 1 and all other bits to be preserved are 0 s. Figure 7.6 shows a graphical depiction of using the XOR operation and a mask to toggle bits in the dst.



An XOR operation with a mask allows bits in the dst to be **toggled**. Recall that the logic for an XOR gate is such that XOR'ing any value with a 0 will preserve the dst bits while XOR'ing any value with a 1 will complement the dst bits. We XOR the dst with a mask that has 1's in any position that we wish to toggle and 0's in the other positions that we wish to preserve.

mask	dst	result
0	0	0
0	1	1
1	0	1
1	1	0

Fig. 7.6
Using an XOR operation and a mask to toggle bits

Now let's look at these instructions in assembly. Follow Example 7.6 to gain experience with these logic operations and the concept of bit masks for clearing, testing, setting, and toggling.

Example: Using Logic Instructions to Manipulate Bits

The inv, and, or, and xor instructions can be used to perform bitwise manipulations on information. Let's see how these instructions work and how they can be used with masks to clear, test, set, and toggle bits in the dst.

1) Create a new Empty Assembly-only CCS project titled: **Asm_ALU_Logic**.

2) Type in the following code into the main.asm file where the comments say "Main loop here".

```
main:
●           mov.b    #10101010b, R4
BP          inv.b    R4

            mov.b    #11110000b, R5
            and.b    #00111111b, R5

            mov.b    #00010000b, R6
            and.b    #10000000b, R6

            mov.b    #00010000b, R7
            and.b    #00010000b, R7

            mov.b    #11000001b, R8
            or.b     #00011111b, R8

            mov.b    #01010101b, R9
            xor.b    #11110000b, R9
            xor.b    #00001111b, R9

            jmp      main
```

```
INV 10101010
    01010101

Use AND to clear bits 7:6            11110000
                                 AND 00111111
                                     00110000

Use AND to test whether              00010000
bit 7 is a one.                  AND 10000000
                                     00000000  Z=1, "no"

Use AND to test whether              00010000
bit 4 is a one.                  AND 00010000
                                     00010000  Z=0, "yes"

Use OR to set bits 5:0               11000000
                                 OR  00011111
                                     11011111

Use XOR to toggle bits 7:4           01010101
                                 XOR 11110000
                                     10100101

Then toggle bits 3:0                 10100101
                                 XOR 00001111
                                     10101010
```

3) Debug your program. Set a breakpoint before the first instruction (**mov.b #10101010, R4**).

4) Open the Register Viewer and expand the Core Registers item to see the CPU registers. Also expand SR so you can see the Z flag. Change the number format of R4→R9 to binary.

5) Run your program to the breakpoint. Step your program to observe its operation.

? → Did it work? Did you see the results listed above?

Example 7.6
Using logic instructions to manipulate bits

Concept Check

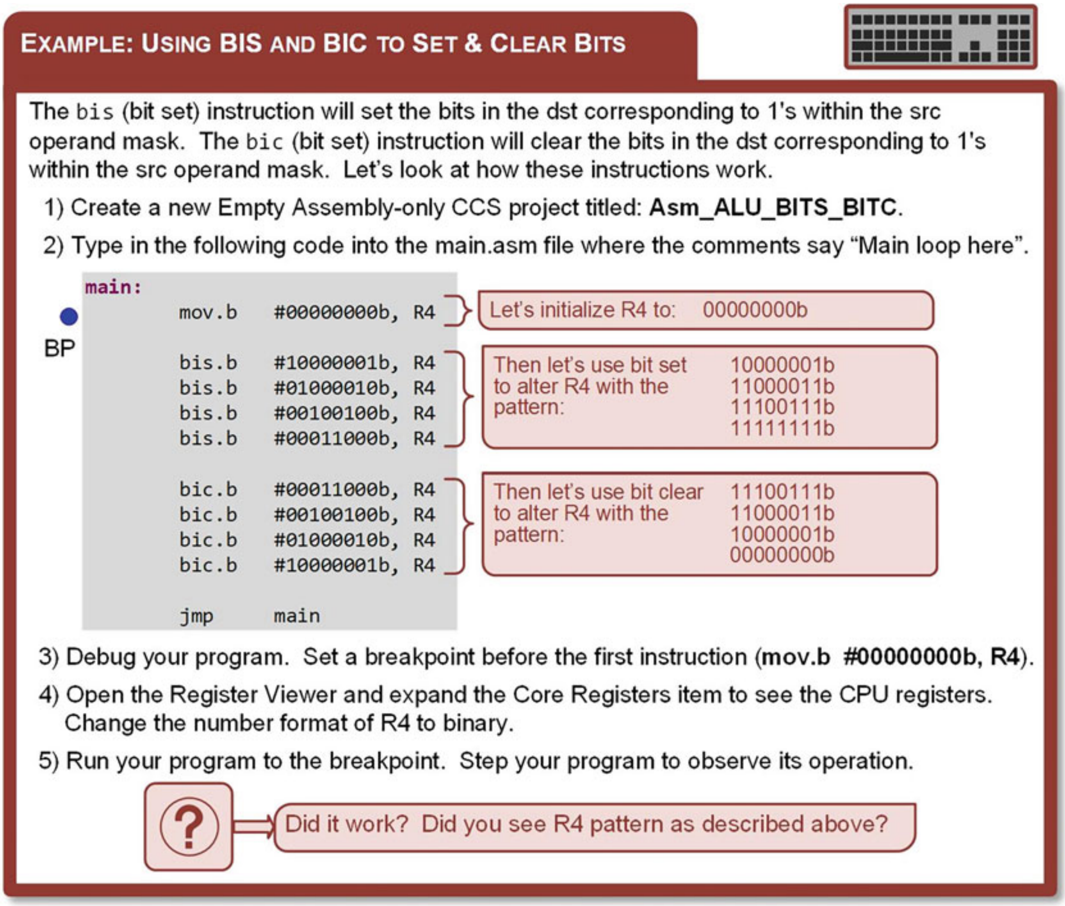
CC7.2 Can we use an XOR operation to create the same functionality as an INV? If so, how?

A) No. XOR operations are bitwise, meaning that they operate on each bit independent from the other bits. An INV complements the entire operand.

B) Yes. If we XOR'd an operand with a mask of all 1s, then each bit would be toggled. This is the same functionality as an INV operation. In fact, the INV instruction is actually emulated, and in reality, the MSP430 does use an XOR operation to perform an inversion of an operand.

7.3 Bit Set and Bit Clear Instructions

Setting and clearing bits is such a common task in an MCU that the MSP430 provides dedicated instructions to perform these operations. The **`bis`** (bit set) instruction will set the bits in the dst corresponding to 1 s within the src operand mask. The **`bic`** (bit clear) instruction will clear the bits in the dst corresponding to 1 s within the src operand mask. All bits within the mask for each instruction that are 0 s will leave the dst bits unaltered. Providing the bits to be altered as 1 s within the mask make the programming more straightforward. If we used the `and` instruction to perform bit clears, we would have to create a mask that contained 0 s in the locations of the bits to be cleared and 1 s where bits were to be preserved. This would result in a mask that was different for setting bit-n of a word versus clearing bit-n of a word. By providing a masking approach that works the same for **`bis`** and **`bic`**, the same masks can be used throughout the program to alter the same bit location. The **`bis`** and **`bic`** instructions work on both 16-bit words (.w) and 8-bit bytes (.b). While multiple addressing modes are supported for the src, the common approach is to use immediate mode for the src mask. Follow Example 7.7 to see how the bit set and bit clear instructions operate.



Example: Using BIS and BIC to Set & Clear Bits

The bis (bit set) instruction will set the bits in the dst corresponding to 1's within the src operand mask. The bic (bit set) instruction will clear the bits in the dst corresponding to 1's within the src operand mask. Let's look at how these instructions work.

1) Create a new Empty Assembly-only CCS project titled: **Asm_ALU_BITS_BITC**.

2) Type in the following code into the main.asm file where the comments say "Main loop here".

```
main:
●         mov.b    #00000000b, R4
BP
          bis.b    #10000001b, R4
          bis.b    #01000010b, R4
          bis.b    #00100100b, R4
          bis.b    #00011000b, R4

          bic.b    #00011000b, R4
          bic.b    #00100100b, R4
          bic.b    #01000010b, R4
          bic.b    #10000001b, R4

          jmp      main
```

Let's initialize R4 to: 00000000b

Then let's use bit set to alter R4 with the pattern: 10000001b 11000011b 11100111b 11111111b

Then let's use bit clear to alter R4 with the pattern: 11100111b 11000011b 10000001b 00000000b

3) Debug your program. Set a breakpoint before the first instruction (**mov.b #00000000b, R4**).

4) Open the Register Viewer and expand the Core Registers item to see the CPU registers. Change the number format of R4 to binary.

5) Run your program to the breakpoint. Step your program to observe its operation.

Did it work? Did you see R4 pattern as described above?

Example 7.7
Using BIS and BIC to set and clear bits

Concept Check

CC7.3 What is the advantage of bit set/clear instructions over using AND/OR to perform sets and clears?

A) Bit sets/clears allow the same mask to be used for both sets and clears.

B) There is no advantage. They act the same.

7.4 Test Instructions

Testing a bit or word is where you are trying to determine something about it such as whether a bit is 1, whether the value is 0, whether the value is negative, or whether the value is the same as another number. The MSP430 provides three test instructions: **bit**, **cmp**, and **tst**. Each of these instructions is unique in that it performs an operation using the values held in the dst and updates the VNZC flags;

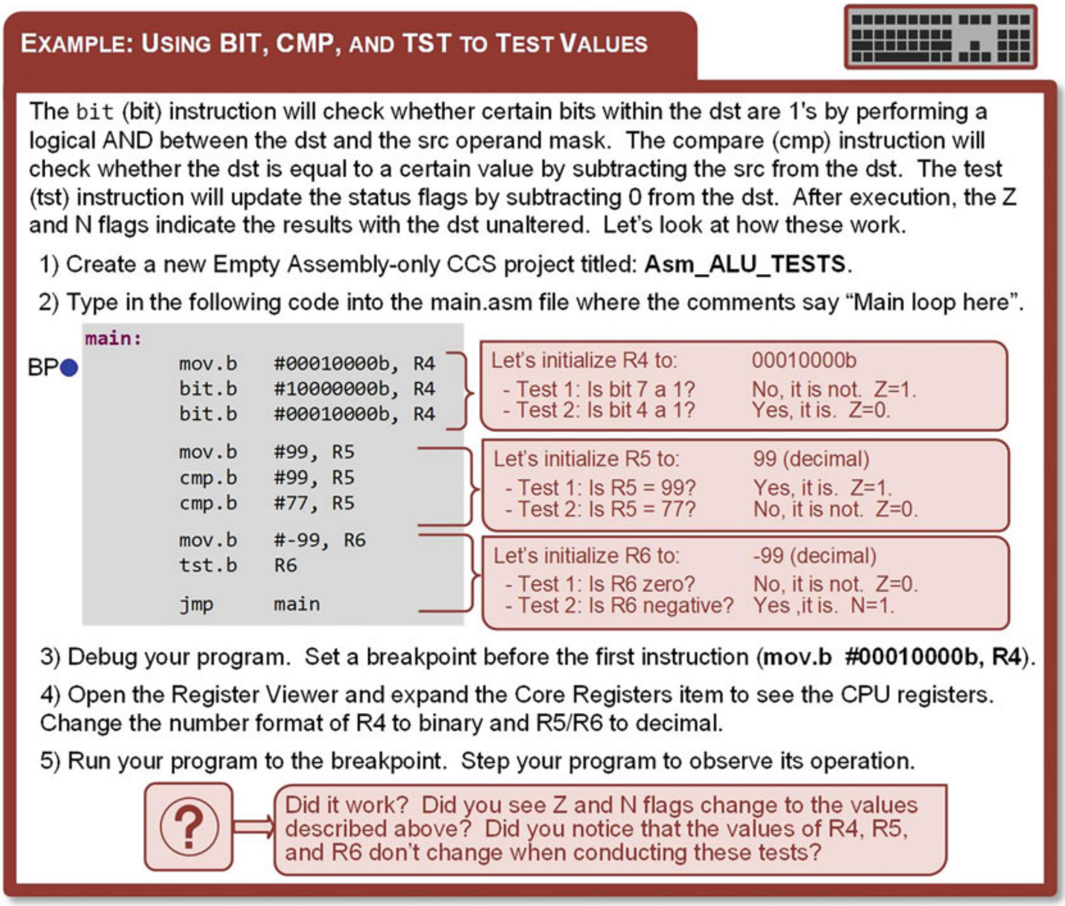
however, it does not alter the value of the dst itself. This allows information to be determined about the dst without destroying its contents.

The bit test (**`bit`**) instruction will perform a logical AND with the mask provided in the src and the value held in the dst. This is used to determine whether certain bits within the dst are 1 s by checking the Z-flag after the operation. If the bit location in the dst dictated by the mask is a 1, then the result of the AND will be a value that is not 0 and the Z-flag will not be asserted (Z = 0). If the Z-flag is asserted (Z = 1), then the bit value of interest in the dst was a 0.

The compare (**`cmp`**) instruction will subtract the src from the dst and update the status flags, but leave the dst intact. This instruction is used to determine if the dst is equal to a specific value by checking the Z-flag after the operation. If the src and dst are equal, the result of the subtraction will result in 0, and the Z-flag will be asserted (Z = 1). If the Z-flag is not asserted (Z = 0), then the src and dst were not the same.

The test (**tst**) instruction simply subtracts 0 from the dst and updates the status flags. This instruction allows us to determine whether the value in the dst is 0 (Z = 1) or negative (N = 1).

Follow Example 7.8 to see how the **`bit`**, **`cmp`**, and **`tst`** instructions work.



EXAMPLE: USING BIT, CMP, AND TST TO TEST VALUES

The bit (bit) instruction will check whether certain bits within the dst are 1's by performing a logical AND between the dst and the src operand mask. The compare (cmp) instruction will check whether the dst is equal to a certain value by subtracting the src from the dst. The test (tst) instruction will update the status flags by subtracting 0 from the dst. After execution, the Z and N flags indicate the results with the dst unaltered. Let's look at how these work.

1) Create a new Empty Assembly-only CCS project titled: **Asm_ALU_TESTS**.

2) Type in the following code into the main.asm file where the comments say "Main loop here".

```
main:
BP●         mov.b     #00010000b, R4
            bit.b     #10000000b, R4
            bit.b     #00010000b, R4

            mov.b     #99, R5
            cmp.b     #99, R5
            cmp.b     #77, R5

            mov.b     #-99, R6
            tst.b     R6

            jmp       main
```

Let's initialize R4 to: 00010000b
- Test 1: Is bit 7 a 1? No, it is not. Z=1.
- Test 2: Is bit 4 a 1? Yes, it is. Z=0.

Let's initialize R5 to: 99 (decimal)
- Test 1: Is R5 = 99? Yes, it is. Z=1.
- Test 2: Is R5 = 77? No, it is not. Z=0.

Let's initialize R6 to: -99 (decimal)
- Test 1: Is R6 zero? No, it is not. Z=0.
- Test 2: Is R6 negative? Yes ,it is. N=1.

3) Debug your program. Set a breakpoint before the first instruction (**mov.b #00010000b, R4**).

4) Open the Register Viewer and expand the Core Registers item to see the CPU registers. Change the number format of R4 to binary and R5/R6 to decimal.

5) Run your program to the breakpoint. Step your program to observe its operation.

Did it work? Did you see Z and N flags change to the values described above? Did you notice that the values of R4, R5, and R6 don't change when conducting these tests?

Example 7.8
Using BIT, CMP, and TST to test values

Concept Check

CC7.4 What is an example of when you need to use a test instruction to find out information about an operand because you can't, or don't want, to alter that operand?

A) Reading from an input port. The input is what it is and isn't altered by the computer, but we often want to know information about the value of the input.

B) Using the value of an operand multiple times in a segment of code.

C) Both B & C.

7.5 Rotate Operations

The MSP430 also contains a variety of rotate instructions. Rotate instructions can be used to shift in serial data into a parallel word, to sign extend negative numbers, or to perform multiply/divide by two arithmetic operations. Figure 7.7 shows a graphical description of the four main rotate instructions: **`rla`**, **`rra`**, **`rlc`**, and **`rrc`**.

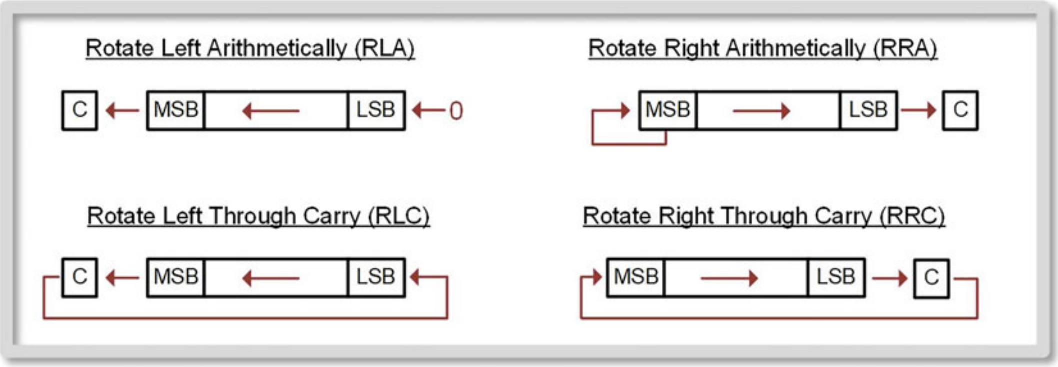

Rotate Left Arithmetically (RLA)
C
MSB
LSB
0
Rotate Right Arithmetically (RRA)
MSB
LSB
C
Rotate Left Through Carry (RLC)
C
MSB
LSB
Rotate Right Through Carry (RRC)
MSB
LSB
C

Fig. 7.7
Graphical depiction of MSP430 rotate instructions

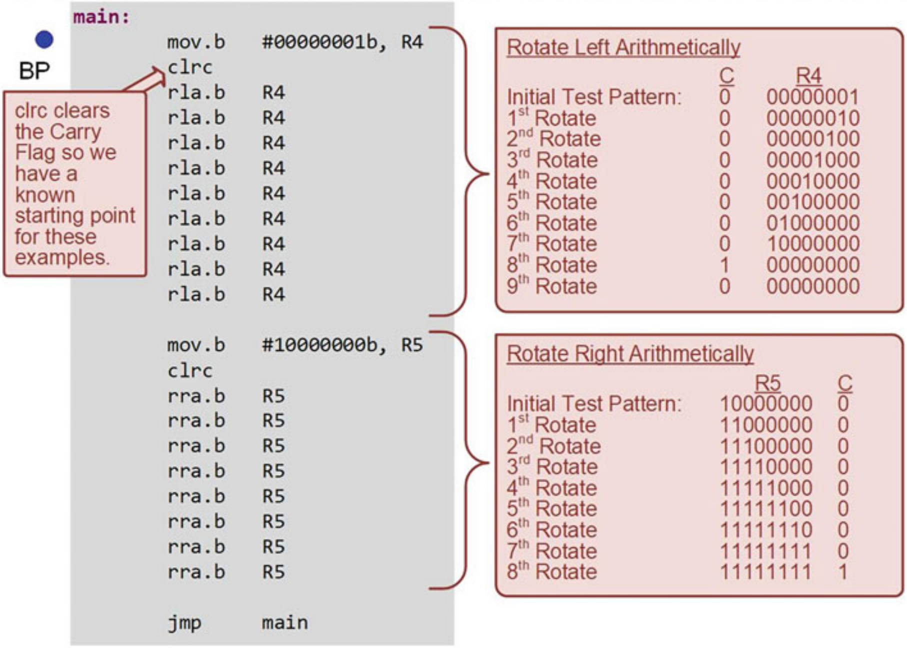
Follow Example 7.9 to see how the rotate arithmetically instructions work.

EXAMPLE: USING ROTATE ARITHMETICALLY

Rotate arithmetically shifts the bits of the dst either right or left and fills the vacated bit position with either a 1 or a 0. The C-bit in the SR is treated as an additional bit within the shift. These instructions support 16-bit (.w) or 8-bit (.b) rotates. Let's look at how these instructions work.

1) Create a new Empty Assembly-only CCS project titled: **Asm_ALU_ROTATE_ARITH**.

2) Type in the following code into the main.asm file where the comments say "Main loop here".



```
main:
            mov.b   #00000001b, R4
            clrc
            rla.b   R4
            rla.b   R4
            rla.b   R4
            rla.b   R4
            rla.b   R4
            rla.b   R4
            rla.b   R4
            rla.b   R4
            rla.b   R4

            mov.b   #10000000b, R5
            clrc
            rra.b   R5
            rra.b   R5
            rra.b   R5
            rra.b   R5
            rra.b   R5
            rra.b   R5
            rra.b   R5
            rra.b   R5

            jmp     main
```

clrc clears the Carry Flag so we have a known starting point for these examples.

Rotate Left Arithmetically

	C	R4
Initial Test Pattern:	0	00000001
1st Rotate	0	00000010
2nd Rotate	0	00000100
3rd Rotate	0	00001000
4th Rotate	0	00010000
5th Rotate	0	00100000
6th Rotate	0	01000000
7th Rotate	0	10000000
8th Rotate	1	00000000
9th Rotate	0	00000000

Rotate Right Arithmetically

	R5	C
Initial Test Pattern:	10000000	0
1st Rotate	11000000	0
2nd Rotate	11100000	0
3rd Rotate	11110000	0
4th Rotate	11111000	0
5th Rotate	11111100	0
6th Rotate	11111110	0
7th Rotate	11111111	0
8th Rotate	11111111	1

3) Debug your program. Set a breakpoint before the first instruction (**mov.b #00000001b, R4**).

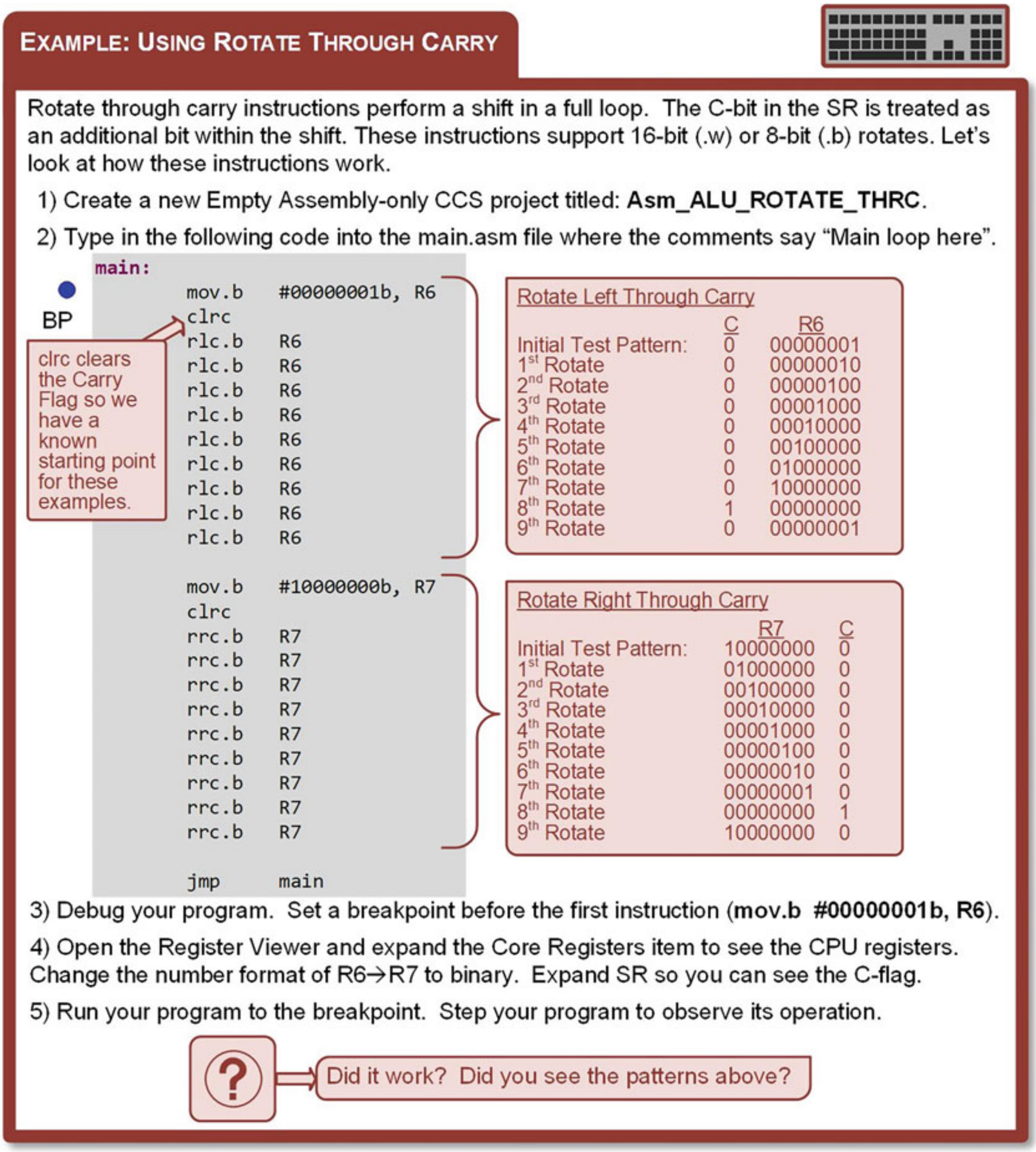
4) Open the Register Viewer and expand the Core Registers item to see the CPU registers. Change the number format of R4→R5 to binary. Expand SR so you can see the C-flag.

5) Run your program to the breakpoint. Step your program to observe its operation.

? Did it work? Did you see the patterns above?

Example 7.9
Using rotate arithmetically

Follow Example 7.10 to see how the rotate through carry instructions work.



EXAMPLE: USING ROTATE THROUGH CARRY

Rotate through carry instructions perform a shift in a full loop. The C-bit in the SR is treated as an additional bit within the shift. These instructions support 16-bit (.w) or 8-bit (.b) rotates. Let's look at how these instructions work.

1) Create a new Empty Assembly-only CCS project titled: **Asm_ALU_ROTATE_THRC**.

2) Type in the following code into the main.asm file where the comments say "Main loop here".

```
main:
            mov.b    #00000001b, R6
            clrc
            rlc.b    R6
            rlc.b    R6
            rlc.b    R6
            rlc.b    R6
            rlc.b    R6
            rlc.b    R6
            rlc.b    R6
            rlc.b    R6
            rlc.b    R6

            mov.b    #10000000b, R7
            clrc
            rrc.b    R7
            rrc.b    R7
            rrc.b    R7
            rrc.b    R7
            rrc.b    R7
            rrc.b    R7
            rrc.b    R7
            rrc.b    R7
            rrc.b    R7

            jmp      main
```

clrc clears the Carry Flag so we have a known starting point for these examples.

Rotate Left Through Carry

	C	R6
Initial Test Pattern:	0	00000001
1st Rotate	0	00000010
2nd Rotate	0	00000100
3rd Rotate	0	00001000
4th Rotate	0	00010000
5th Rotate	0	00100000
6th Rotate	0	01000000
7th Rotate	0	10000000
8th Rotate	1	00000000
9th Rotate	0	00000001

Rotate Right Through Carry

	R7	C
Initial Test Pattern:	10000000	0
1st Rotate	01000000	0
2nd Rotate	00100000	0
3rd Rotate	00010000	0
4th Rotate	00001000	0
5th Rotate	00000100	0
6th Rotate	00000010	0
7th Rotate	00000001	0
8th Rotate	00000000	1
9th Rotate	10000000	0

3) Debug your program. Set a breakpoint before the first instruction (**mov.b #00000001b, R6**).

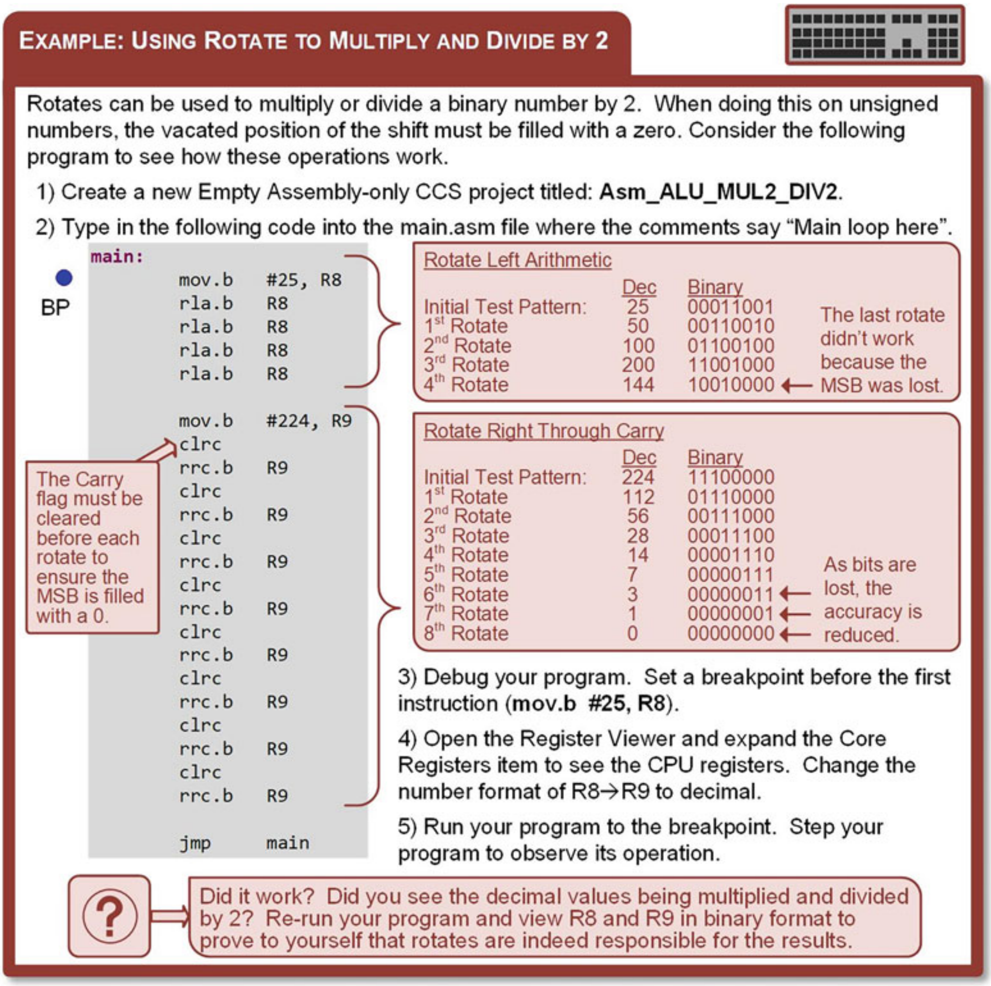
4) Open the Register Viewer and expand the Core Registers item to see the CPU registers. Change the number format of R6→R7 to binary. Expand SR so you can see the C-flag.

5) Run your program to the breakpoint. Step your program to observe its operation.

Did it work? Did you see the patterns above?

Example 7.10
Using rotate through carry

Rotate instructions provide the ability to perform simply multiply-by-2 and divide-by-2 operations. When a binary number is rotated to the left by one bit and the vacated LSB position is filled with a 0, it doubles the original value. When a binary number is rotated to the right by one bit and the vacated MSB position is filled with a 0, it has the original value. Follow Example 7.11 to see how this multiply-by-2 and divide-by-2 functionality is implemented using rotates.



Example: Using Rotate to Multiply and Divide by 2

Rotates can be used to multiply or divide a binary number by 2. When doing this on unsigned numbers, the vacated position of the shift must be filled with a zero. Consider the following program to see how these operations work.

1) Create a new Empty Assembly-only CCS project titled: **Asm_ALU_MUL2_DIV2**.

2) Type in the following code into the main.asm file where the comments say "Main loop here".

● BP

```
main:
        mov.b   #25, R8
        rla.b   R8
        rla.b   R8
        rla.b   R8
        rla.b   R8

        mov.b   #224, R9
        clrc
        rrc.b   R9
        clrc
        rrc.b   R9
        clrc
        rrc.b   R9
        clrc
        rrc.b   R9
        clrc
        rrc.b   R9
        clrc
        rrc.b   R9
        clrc
        rrc.b   R9
        clrc
        rrc.b   R9

        jmp     main
```

The Carry flag must be cleared before each rotate to ensure the MSB is filled with a 0.

Rotate Left Arithmetic

	Dec	Binary
Initial Test Pattern:	25	00011001
1st Rotate	50	00110010
2nd Rotate	100	01100100
3rd Rotate	200	11001000
4th Rotate	144	10010000 ←

The last rotate didn't work because the MSB was lost.

Rotate Right Through Carry

	Dec	Binary
Initial Test Pattern:	224	11100000
1st Rotate	112	01110000
2nd Rotate	56	00111000
3rd Rotate	28	00011100
4th Rotate	14	00001110
5th Rotate	7	00000111
6th Rotate	3	00000011 ←
7th Rotate	1	00000001 ←
8th Rotate	0	00000000 ←

As bits are lost, the accuracy is reduced.

3) Debug your program. Set a breakpoint before the first instruction (**mov.b #25, R8**).

4) Open the Register Viewer and expand the Core Registers item to see the CPU registers. Change the number format of R8→R9 to decimal.

5) Run your program to the breakpoint. Step your program to observe its operation.

? Did it work? Did you see the decimal values being multiplied and divided by 2? Re-run your program and view R8 and R9 in binary format to prove to yourself that rotates are indeed responsible for the results.

Example 7.11
Using rotate to multiply and divide by 2

Concept Check

CC7.5 What constraint do you need to keep in mind when using rotates to perform multiply-by-2 or divide-by-2?

A) You need to make sure that bits of the number aren't being rotated out of the register. When this happens, the value is either incorrect or has diminished accuracy.

B) You can only use this technique on even numbers, so you need to monitor the input value to ensure it is a valid number to be operated on.

C) You can only use this technique on odd numbers, so you need to monitor the input value to ensure it is a valid number to be operated on.

D) You need to be careful that the other bits in the status register aren't accidentally included in the shift.

Summary

- The MSP430 ALU performs a variety of operations. Each operation can be thought of as an independent combinational logic circuit that is selected for use by the control unit.
- The ALU instructions set the VNZC flags in the status register to provide information about the result of the operation.
- The addition instruction (**`add`**) works the same on unsigned versus signed numbers. It is up to the programmer to decide how to treat the binary values.
- The add with carry instruction (**`addc`**) allows numbers to be added that are larger than the 16-bit ALU can handle. The larger numbers are broken down into 16-bit words, and then the carry from prior additions is included in the next higher addition using the `addc` instruction.
- The subtraction instruction (**`sub`**) will modify the C-flag to indicate whether a borrow occurred. The logic is if C = 1, then no borrow occurred, while if C = 0, then a borrow did occur. The C-flag logic is due to the way that the MSP430 performs subtraction by taking the two's complement of the src and then adding it to the dst.
- The subtract with carry instruction (**`subc`**) allows numbers to be subtracted that are larger than the 16-bit ALU can handle. The larger numbers are broken down into 16-bit words, and then the borrow from a prior subtraction is included in the next higher subtraction using the `subc` instruction.
- Increment (**`inc`**) and decrement (**`dec`**) instructions add or subtract 1 from the dst, respectively.
- Increment double (**`incd`**) and decrement double (**`decb`**) instructions add or subtract 2 from the dst, respectively. These instructions are useful when accessing 16-bit words in memory that are aligned to even addresses.
- The MSP430 provides the logic operation mnemonics **`inv`**, **`and`**, **`or`**, and **`xor`**. These are considered bitwise operations because the individual bits of the src and/or dst are acted on independently from one another.
- A bit mask can be used with the **`and`**, **`or`**, and **`xor`** instructions to clear, set, or toggle individual bits of the dst. The **`and`** instruction can be used to clear bits. The **`or`** operation can be used to set bits. The **`xor`** operation can be used to toggle bits.
- Setting and clearing bits is such a common operation in an MCU that the MSP430 provides dedicated instructions for it. The **`bis`** and **`bic`** instructions will set and clear bits in the dst based on the location of 1 s in the src. The most common use model for these instructions is to provide a bit mask in the src using immediate addressing.
- The MSP430 provides test instructions that will provide information about the dst, but not change the value of the dst. The **`bit`** instruction will perform a bitwise of the src and dst. The **`cmp`** instruction will subtract the src from the dst. The **`tst`** instruction will subtract 0 from the dst. Information about the dst is indicated by the values of the VNZC bits in the status register.
- The MSP provides four types of rotate instructions. Rotate left/right arithmetically instructions (**`rla`**, **`rca`**) will shift the dst and fill in the vacated bit position with 1 s or 0 s. The rotate left/right through carry instructions (**`rlc`**, **`rrc`**) will shift the dst in a complete loop. All shifts use the C-flag as another bit within the shift.
- Rotates can be used to perform simple multiply-by-2 and divide-by-2 operations on the dst.

Exercise Problems

Section 7.1: Arithmetic Instructions

7.1.1 Is the C-flag asserted after the following instructions are executed?

```
mov.b  #99, R4
add.b  #1, R4
```

7.1.2 Is **the** Z-flag asserted after the following instructions are executed?

```
mov.b  #99, R4
add.b  #1, R4
```

7.1.3 Is **the** N-flag asserted after the following instructions are executed?

```
mov.b  #99, R4
add.b  #1, R4
```

7.1.4 Is **the** C-flag asserted after the following instructions are executed?

```
mov.b  #255, R4
add.b  #1, R4
```

7.1.5 Is the Z-flag asserted after the following instructions are executed?

```
mov.b  #255, R4
add.b  #1, R4
```

7.1.6 Is the N-flag asserted after the following instructions are executed?

```
mov.b  #255, R4
add.b  #1, R4
```

7.1.7 Is the C-flag asserted after the following instructions are executed?

```
mov.w  #255, R4
add.w  #1, R4
```

7.1.8 Is the Z-flag asserted after the following instructions are executed?

```
mov.w  #255, R4
add.w  #1, R4
```

7.1.9 Is the N-flag asserted after the following instructions are executed?

```
mov.w  #255, R4
add.w  #1, R4
```

7.1.10 Is the C-flag asserted after the following instructions are executed?

```
mov.b  #255, R4
sub.b  #1, R4
```

7.1.11 Is the Z-flag asserted after the following instructions are executed?

```
ov.b   #255, R4
sub.b  #1, R4
```

7.1.12 Is the N-**flag** asserted after the following instructions are executed?

```
mov.b  #255, R4
sub.b  #1, R4
```

7.1.13 Is the C-**flag** asserted after the following instructions are executed?

```
mov.b  #1, R4
sub.b  #255, R4
```

7.1.14 Is the Z-flag asserted after the following instructions are executed?

```
mov.b  #1, R4
sub.b  #255, R4
```

7.1.15 Is the N-flag asserted after the following instructions are executed?

```
mov.b  #1, R4
sub.b  #255, R4
```

Section 7.2: Logic Instructions

7.2.1 What bit positions (7 to 0) in R4 will be inverted when the following INV operation is executed?

```
inv.b  R4
```

7.2.2 What bit positions (7 to 0) in R4 will be cleared when the following AND operation is executed?

```
and.b  #01111110b, R4
```

7.2.3 What bit positions (7 to 0) in R4 will be set when the following OR operation is **executed**?

```
or.b  #01111110b, R4
```

7.2.4 What **bit** positions (7 to 0) in R4 will be toggled when the following XOR operation is executed?

```
xor.b  #01111110b, R4
```

7.2.5 What bit positions (7 to 0) in R4 will be cleared when the following AND operation is executed?

```
and.b  #8, R4
```

7.2.6 What bit positions (7 to 0) in R4 will be set when the following OR operation is executed?

```
or.b  #16, R4
```

7.2.7 What bit **positions** (7 to 0) in R4 will be toggled when the following XOR operation is executed?

```
xor.b  #21, R4
```

Section 7.3: Bit Set and Bit Clear Instructions

7.3.1 What bit positions (7 to 0) in R4 will be cleared when the following instruction is executed?

```
bitc.b  #11110000b, R4
```

7.3.2 What bit positions (7 to 0) in R4 will be set when the following instruction is executed?

```
bits.b  #00000011b, R4
```

7.3.3 What bit **positions** (7 to 0) in R4 will be cleared when the following instruction is executed?

```
bitc.b  #1, R4
```

7.3.4 What bit positions (7 to 0) in R4 will be set when the following instruction is executed?

```
bits.b  #2, R4
```

7.3.5 What bit positions (7 to 0) in R4 will be cleared when the following instruction is executed?

```
bitc.w  #256, R4
```

7.3.6 What bit **positions** (7 to 0) in R4 will be set when the following instruction is executed?

```
bits.w  #513, R4
```

Section 7.4: Test Instructions

7.4.1 Is the Z-flag asserted after the following instructions are executed?

```
mov.b  #11110011b, R4
bit.b  #11110011b, R4
```

7.4.2 Is the Z-flag **asserted** after the following instructions are executed?

```
mov.b  #11110011b, R4
bit.b  #00001100b, R4
```

7.4.3 Is the Z-flag asserted after the following instructions are executed?

```
mov.b  #11110011b, R4
bit.b  #4, R4
```

7.4.4 Is the Z-flag asserted after the following instructions are executed?

```
mov.b  #11110011b, R4
bit.b  #16, R4
```

7.4.5 Is the Z-flag asserted after the following instructions are executed?

```
mov.b  #16, R4
cmp.b  #16, R4
```

7.4.6 Is the Z-flag asserted after the following instructions are executed?

```
mov.b  #11110000b, R4
cmp.b  #00001111b, R4
```

7.4.7 Is the Z-flag asserted after the following instructions are executed?

```
mov.b  #0FFh, R4
cmp.b  #255, R4
```

7.4.8 Is the Z-flag asserted after the following instructions are executed?

```
mov.w  #99h, R4
cmp.w  #99, R4
```

7.4.9 Is the Z-flag asserted after the following instructions are executed?

```
mov.b  #10000000b, R4
tst.b  R4
```

7.4.10 Is the N-flag asserted after the following instructions are executed?

```
mov.b  #10000000b, R4
tst.b  R4
```

7.4.11 Is the Z-flag asserted after the following instructions are executed?

```
mov.b  #255, R4
tst.b  R4
```

7.4.12 Is the N-flag asserted after the following instructions are executed?

```
mov.b  #255, R4
tst.b  R4
```

Section 7.5: Rotate Instructions

7.5.1 What is the value in position 0 of R4 after the following instructions have been executed?

```
mov.b  #10000001b, R4
rla.b  R4
```

7.5.2 What is the value in position 7 of R4 after the following instructions have been executed?

```
mov.b  #10000001b, R4
rla.b  R4
```

7.5.3 What is the value of the C-flag after the following instructions have been executed?

```
mov.b  #10000001b, R4
rla.b  R4
```

7.5.4 What is the value in position 0 of R4 after the following instructions have been executed?

```
mov.b  #10000001b, R4
rra.b  R4
```

7.5.5 What is the value in position 7 of R4 after the following instructions have been executed?

```
mov.b  #10000001b, R4
rra.b  R4
```

7.5.6 What is the value of the C-flag after the following instructions have been executed?

```
mov.b  #10000001b, R4
rra.b  R4
```

7.5.7 What is the value in position 0 of R4 after the following instructions have been executed (you can assume C = 0 prior to this code)?

```
mov.b  #10000001b, R4
rlc.b  R4
```

7.5.8 What is the value in position 7 of R4 after the following instructions have been executed (you can assume C = 0 prior to this code)?

```
mov.b  #10000001b, R4
rlc.b  R4
```

7.5.9 What is the value of the C-flag after the following instructions have been executed (you can assume C = 0 prior to this code)?

```
mov.b  #10000001b,
rlc.b  R4
```

7.5.10 What is the value in position 0 of R4 after the following instructions have been executed (you can assume C = 0 prior to this code)?

```
mov.b  #10000001b, R4
rrc.b  R4
```

7.5.11 What is the value in position 7 of R4 after the following instructions have been executed (you can assume C = 0 prior to this code)?

```
mov.b  #10000001b, R4
rrc.b  R4
```

7.5.12 What is the value of the C-flag after the following instructions have been executed (you can assume C = 0 prior to this code)?

```
mov.b  #10000001b, R4
rrc.b  R4
```

Chapter 8: Program Flow Instructions

This chapter introduces the MSP430 program flow instructions [1]. Program flow instructions alter the location of the program counter to enable looping or conditional execution of code. Program flow instructions are what gives a computer program the ability to make decisions about what to do next (i.e., what code to execute) based on input conditions or the results of certain operations. Examples of program flow instructions are provided to help in understanding how they impact the program counter. Examples are also given on how to implement common programming constructs such as loops and decision statements. It is intended that the reader is coding the examples using the MSP430FR2355 LaunchPad™ board as they go through this chapter.

Learning Outcomes—After completing this chapter you will be able to:

8.1 Use unconditional jumps and branches to alter the program counter.
8.2 Use conditional jumps to alter the program counter based on the status register flags.
8.3 Implement common programming constructs (while() loops, for() loops, if/else, and switch/case statements) using conditional and unconditional jumps.
8.4 Use flow charts to describe the functionality of a program.

8.1 Unconditional Jumps and Branches

The program counter holds the address of the next instruction to fetch while executing the current instruction. When executing a sequence of instructions, the PC is simply incremented such that it points to the next address in memory that contains the next instruction. MSP430 instructions are either 1×, 2×, or 3× 16-bit words in size. Upon a fetch, the CPU determines which instruction is being executed and then knows how many times to increment the PC so that it is pointing to the next instruction in memory. A jump, or branch, is an instruction that sets the PC to a different value other than the next subsequent instruction in memory. By altering the PC, the flow of the program is changed. This allows blocks of instructions to be repeated (i.e., a loop) or blocks of code to be selectivity executed (i.e., if/else).

There are two broad categories of program flow instructions: *unconditional* and *conditional*. Unconditional instructions will alter PC when they are executed. Conditional instructions will only alter the PC when certain conditions exist on the VNZC flags in SR (e.g., jump only when N = 1).

Address labels are essential to program flow instructions in assembly. As the size of a program grows, it becomes nearly impossible for a programmer to track the exact address locations to put into the PC during a jump or branch. As such, programmers leave it to the assembler to track the exact addresses to use during a jump/branch by using address labels.

The branch (**br**) instruction is an unconditional program flow instruction that simply moves the value of the src operand into PC (`mov.w #dst, PC`). This instruction can point PC to any location within the 16-bit address space of the MSP430FR2355 memory system. A branch almost always uses immediate mode addressing to specify the address of the label. The branch instruction takes three words of program memory.

The jump (**jmp**) instruction is another unconditional program flow instruction that always alters the PC, but does not use the src operand as an address directly. Instead, the src operand is interpreted as a signed offset to apply to the PC. During assembly, the offset is calculated by subtracting the current value of the PC with the address of the label and then forming a 10-bit offset to be used as part of the operand. This limits the jump to only the −511 to +512 addresses around the current value of the PC; however, the **jmp** instruction executes faster because it only takes one word of program memory compared to the

B. J. LaMeres, *Embedded Systems Design using the MSP430FR2355 LaunchPad™*,
https://doi.org/10.1007/978-3-031-20888-1_8

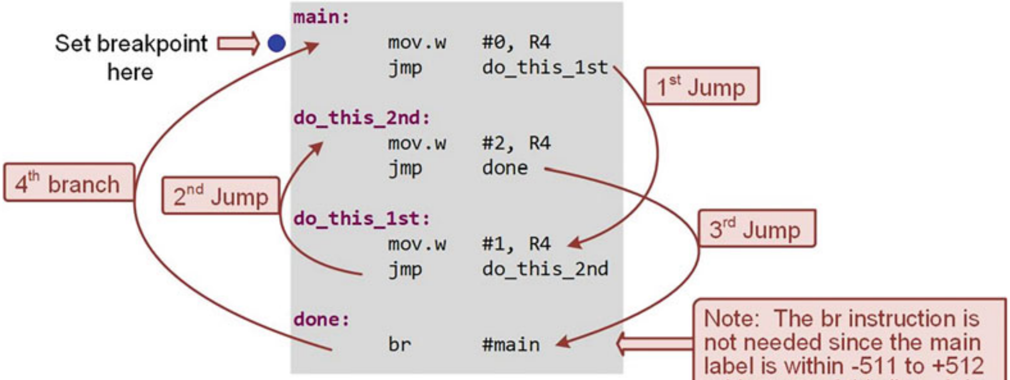
three words that **br** takes. As such, the **jmp** instruction is used more often than **br**. If the jump offset calculated during assembly happens to be outside of the −511 to +512 range, the assembler will give a "jump out of range" error. If this occurs, then a branch must be used. A jump almost always uses symbolic mode addressing with an address label of where to jump. Follow Example 8.1 to see how these unconditional program flow instructions work.

Example: Using Unconditional Jump and Branch Instructions

Unconditional program flow instructions always alter the value of PC when executed. The jump (jmp) instruction applies a 10-bit offset to the current value of PC. This instruction executes in minimal time, but limits the jumps to only the -511 to +512 words around the current PC value. The branch (br) instruction moves the 16-bit src operand into PC, allowing branches to the entire memory range of the MSP430FR2355. Let's see how these instructions work.

1) Create a new Empty Assembly-only CCS project titled: **Asm_Flow_Jump_n_Branch**.

2) Type in the following code into the main.asm file where the comments say "Main loop here".



```
main:
        mov.w   #0, R4
        jmp     do_this_1st

do_this_2nd:
        mov.w   #2, R4
        jmp     done

do_this_1st:
        mov.w   #1, R4
        jmp     do_this_2nd

done:
        br      #main
```

Note: The br instruction is not needed since the main label is within -511 to +512 addresses of this line. It is just used to show the branch syntax.

3) Debug your program. Set a breakpoint before the first instruction (**mov.w #0, R4**).

4) Open the Register Viewer and expand the Core Registers item to see the Program Counter.

5) Run your program to the breakpoint. Step your program to observe its operation.

? Did it work? Do you see the program counter jump to different spots in the code?

Example 8.1
Using unconditional jump and branch instructions

CONCEPT CHECK

CC8.1 Can we implement the equivalent functionality of a branch instruction with a `mov`?

A) Yes. The branch simply puts the value of the address label into PC. This is the same as **`mov dst, PC`**. In fact, the branch is an emulated instruction and is replaced during assembly with a move instruction.

B) No. While a move might produce the same functionality, we need to use mnemonics that make more sense to the assembler. The branch mnemonic is more descriptive, so we need to use it instead of a move.

8.2 Conditional Jumps

Conditional jumps alter the program counter when certain conditions exist in the status flags within the status register. A conditional jump instruction will alter the program counter if the condition is true, which jumps the program counter to a new location in the program. If the condition is false, the program counter will simply move on to the next instruction residing in memory.

8.2.1 Carry-Based Jumps

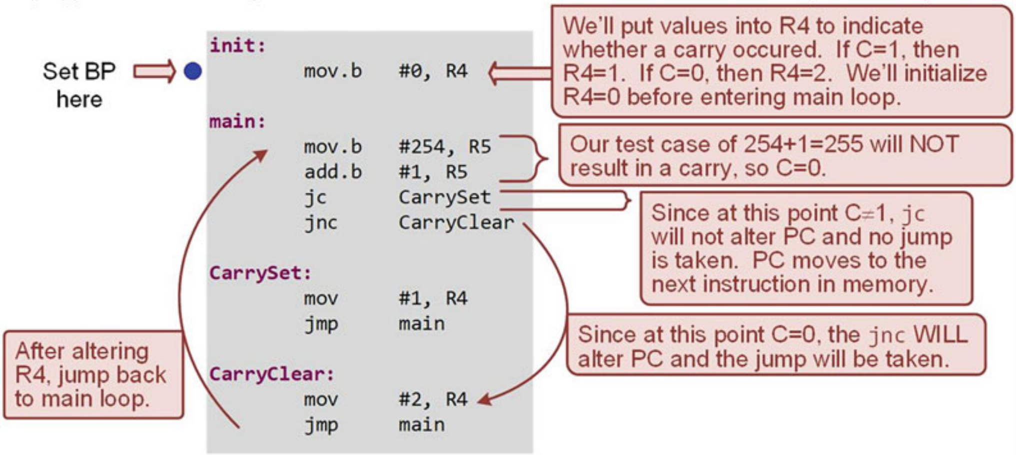
The *jump if carry* (**`jc`**) instruction will alter the program counter if C = 1; otherwise it will simply move on to the next instruction in memory. The *jump if no carry* (**`jn`**`c`) instruction will alter the program counter if C = 0; otherwise it will simply move on to the next instruction in memory. Follow Example 8.2 to see how these carry-based conditional jump instructions work.

Example: Using Jumps Based on the Carry Flag (JC, JNC)

The *jump if carry* (jc) instruction will alter PC if C=1, otherwise it will move to the next instruction in memory. The *jump if no carry* (jnc) instruction will alter PC if C=0, otherwise it will move to the next instruction in memory. Let's see how these work.

1) Create a new Empty Assembly-only CCS project titled: **Asm_Flow_Carry_Jumps**.

2) Type in the following code into the main.asm file where the comments say "Main loop here".



```
init:
        mov.b   #0, R4

main:
        mov.b   #254, R5
        add.b   #1, R5
        jc      CarrySet
        jnc     CarryClear

CarrySet:
        mov     #1, R4
        jmp     main

CarryClear:
        mov     #2, R4
        jmp     main
```

We'll put values into R4 to indicate whether a carry occured. If C=1, then R4=1. If C=0, then R4=2. We'll initialize R4=0 before entering main loop.

Our test case of 254+1=255 will NOT result in a carry, so C=0.

Since at this point C≠1, jc will not alter PC and no jump is taken. PC moves to the next instruction in memory.

Since at this point C=0, the jnc WILL alter PC and the jump will be taken.

After altering R4, jump back to main loop.

3) Debug your program. Set a breakpoint before the first instruction (**mov.b #0, R4**).

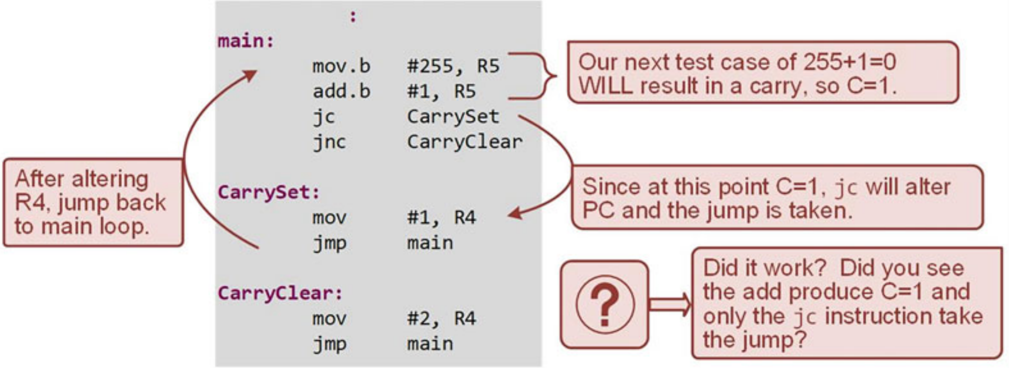
4) Open the Register Viewer and expand the Core Registers and the Status Register so you can see the Carry flag, PC, R4, and R5. Change the format of R4 and R5 to decimal.

5) Run your program to the breakpoint. Step your program to observe its operation.

? Did it work? Did you see the add produce C=0 and only the jnc instruction take the jump?

6) Now change the src of the second mov instruction from 254 to 255 (**mov.b #255, R5**).

7) Debug your program and observe the new behavior of your program.



```
            :
main:
        mov.b   #255, R5
        add.b   #1, R5
        jc      CarrySet
        jnc     CarryClear

CarrySet:
        mov     #1, R4
        jmp     main

CarryClear:
        mov     #2, R4
        jmp     main
```

Our next test case of 255+1=0 WILL result in a carry, so C=1.

Since at this point C=1, jc will alter PC and the jump is taken.

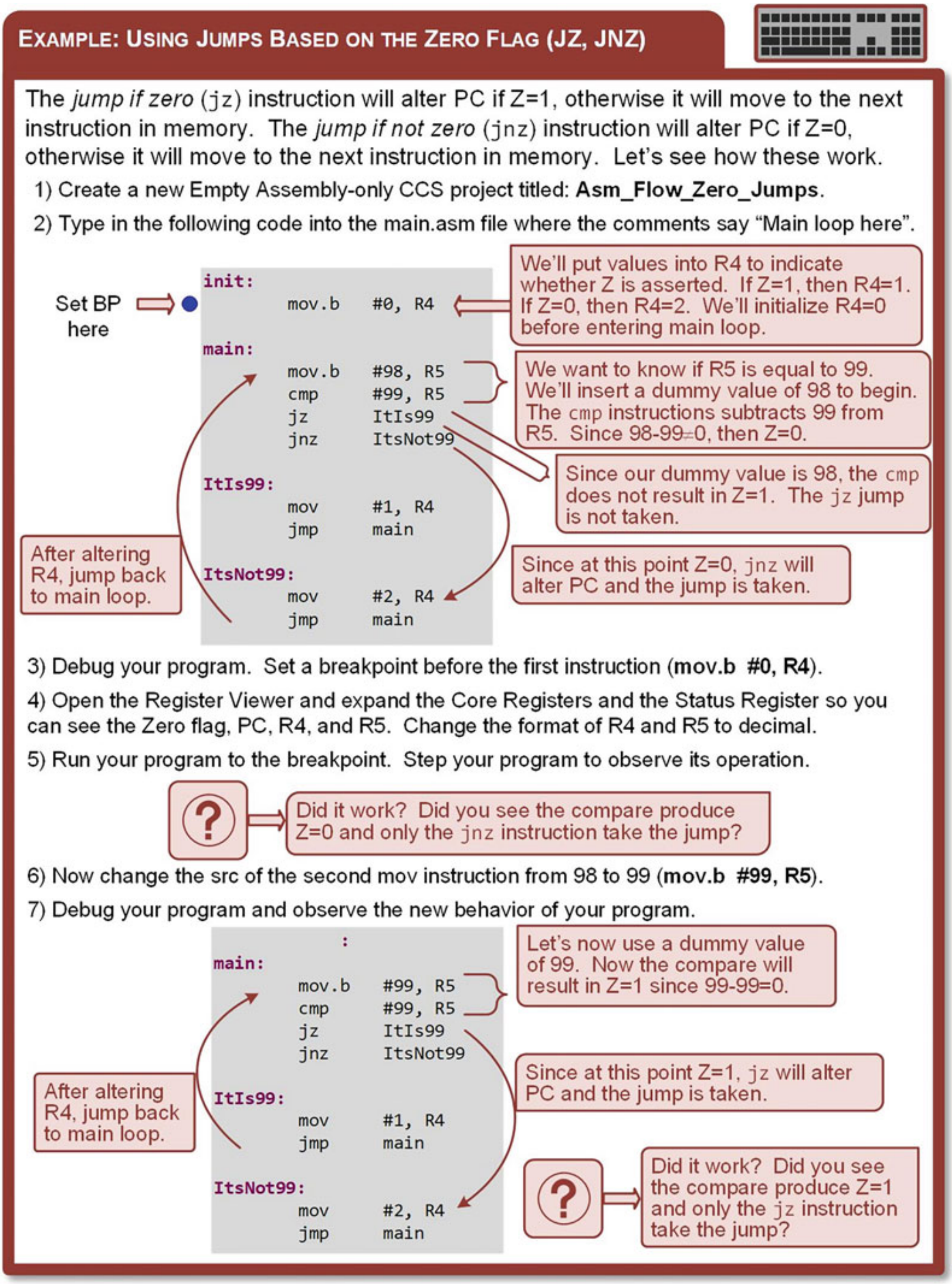
After altering R4, jump back to main loop.

? Did it work? Did you see the add produce C=1 and only the jc instruction take the jump?

Example 8.2
Using jumps based on the carry flag (JC, JNC)

8.2.2 Zero-Based Jumps

The *jump if zero* (**jz**) instruction will alter the program counter if Z = 1; otherwise it will simply move on to the next instruction in memory. The *jump if not zero* (**jnz**) instruction will alter the program counter if Z = 0; otherwise it will simply move on to the next instruction in memory. Follow Example 8.3 to see how these zero-based conditional jump instructions work.



EXAMPLE: USING JUMPS BASED ON THE ZERO FLAG (JZ, JNZ)

The *jump if zero* (jz) instruction will alter PC if Z=1, otherwise it will move to the next instruction in memory. The *jump if not zero* (jnz) instruction will alter PC if Z=0, otherwise it will move to the next instruction in memory. Let's see how these work.

1) Create a new Empty Assembly-only CCS project titled: **Asm_Flow_Zero_Jumps**.

2) Type in the following code into the main.asm file where the comments say "Main loop here".

Set BP here

```
init:
        mov.b   #0, R4

main:
        mov.b   #98, R5
        cmp     #99, R5
        jz      ItIs99
        jnz     ItsNot99

ItIs99:
        mov     #1, R4
        jmp     main

ItsNot99:
        mov     #2, R4
        jmp     main
```

We'll put values into R4 to indicate whether Z is asserted. If Z=1, then R4=1. If Z=0, then R4=2. We'll initialize R4=0 before entering main loop.

We want to know if R5 is equal to 99. We'll insert a dummy value of 98 to begin. The cmp instructions subtracts 99 from R5. Since 98-99≠0, then Z=0.

Since our dummy value is 98, the cmp does not result in Z=1. The jz jump is not taken.

Since at this point Z=0, jnz will alter PC and the jump is taken.

After altering R4, jump back to main loop.

3) Debug your program. Set a breakpoint before the first instruction (**mov.b #0, R4**).

4) Open the Register Viewer and expand the Core Registers and the Status Register so you can see the Zero flag, PC, R4, and R5. Change the format of R4 and R5 to decimal.

5) Run your program to the breakpoint. Step your program to observe its operation.

? Did it work? Did you see the compare produce Z=0 and only the jnz instruction take the jump?

6) Now change the src of the second mov instruction from 98 to 99 (**mov.b #99, R5**).

7) Debug your program and observe the new behavior of your program.

```
              :
main:
        mov.b   #99, R5
        cmp     #99, R5
        jz      ItIs99
        jnz     ItsNot99

ItIs99:
        mov     #1, R4
        jmp     main

ItsNot99:
        mov     #2, R4
        jmp     main
```

Let's now use a dummy value of 99. Now the compare will result in Z=1 since 99-99=0.

Since at this point Z=1, jz will alter PC and the jump is taken.

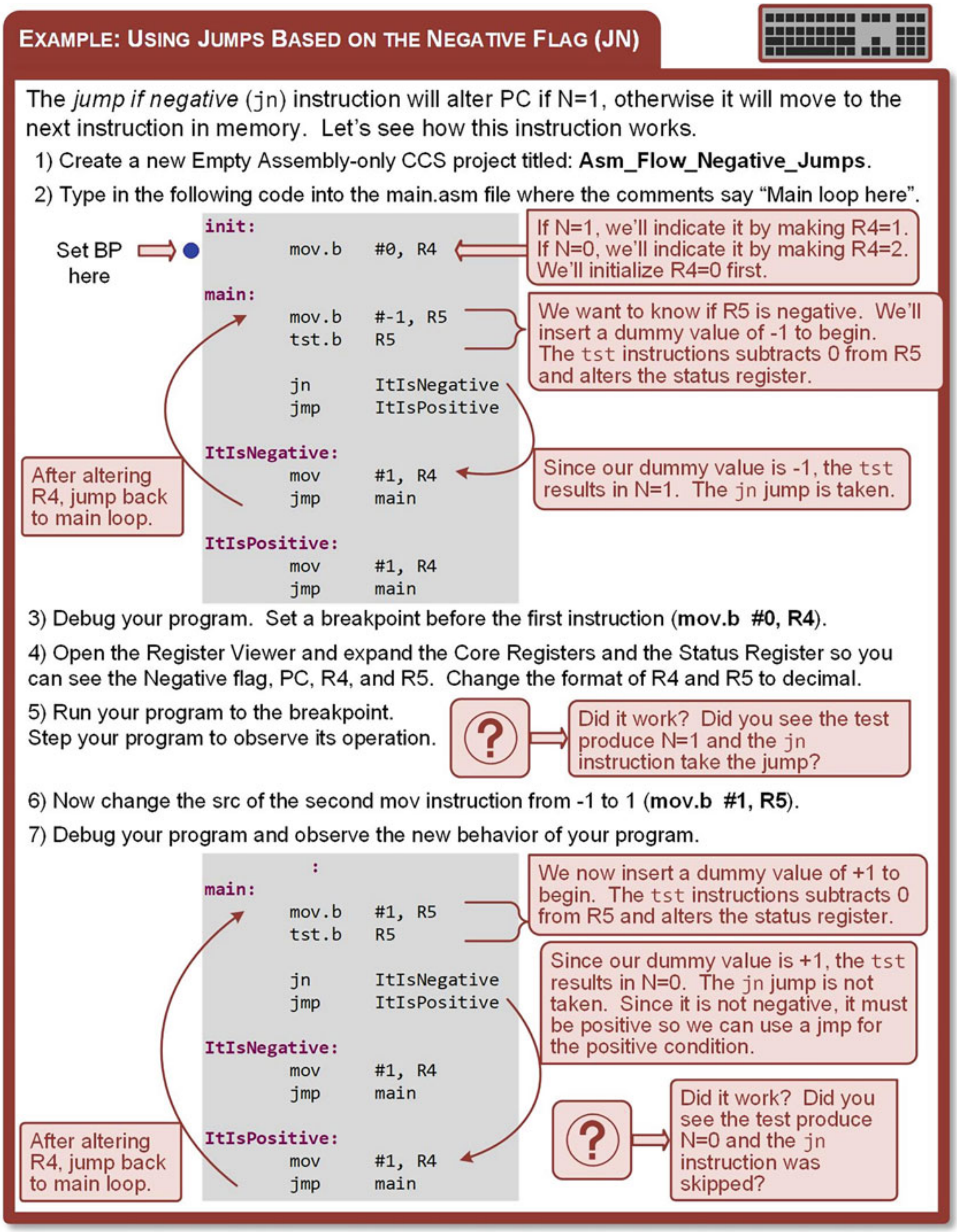
After altering R4, jump back to main loop.

? Did it work? Did you see the compare produce Z=1 and only the jz instruction take the jump?

Example 8.3
Using jumps based on the zero flag (JZ, JNZ)

8.2.3 Negative-Based Jumps

The *jump if negative* (`jn`) instruction will alter the program counter if N = 1; otherwise it will simply move on to the next instruction in memory. There is no *jump if not negative* instruction in the MSP430 instruction set; however, this condition can be created using the logic that if the result is not negative, it must be positive. Follow Example 8.4 to see how this negative-based conditional jump works.



EXAMPLE: USING JUMPS BASED ON THE NEGATIVE FLAG (JN)

The *jump if negative* (`jn`) instruction will alter PC if N=1, otherwise it will move to the next instruction in memory. Let's see how this instruction works.

1) Create a new Empty Assembly-only CCS project titled: **Asm_Flow_Negative_Jumps**.

2) Type in the following code into the main.asm file where the comments say "Main loop here".

```
init:
        mov.b   #0, R4

main:
        mov.b   #-1, R5
        tst.b   R5

        jn      ItIsNegative
        jmp     ItIsPositive

ItIsNegative:
        mov     #1, R4
        jmp     main

ItIsPositive:
        mov     #1, R4
        jmp     main
```

Set BP here

If N=1, we'll indicate it by making R4=1. If N=0, we'll indicate it by making R4=2. We'll initialize R4=0 first.

We want to know if R5 is negative. We'll insert a dummy value of -1 to begin. The `tst` instructions subtracts 0 from R5 and alters the status register.

Since our dummy value is -1, the `tst` results in N=1. The `jn` jump is taken.

After altering R4, jump back to main loop.

3) Debug your program. Set a breakpoint before the first instruction (**mov.b #0, R4**).

4) Open the Register Viewer and expand the Core Registers and the Status Register so you can see the Negative flag, PC, R4, and R5. Change the format of R4 and R5 to decimal.

5) Run your program to the breakpoint. Step your program to observe its operation.

? Did it work? Did you see the test produce N=1 and the `jn` instruction take the jump?

6) Now change the src of the second mov instruction from -1 to 1 (**mov.b #1, R5**).

7) Debug your program and observe the new behavior of your program.

```
          :
main:
        mov.b   #1, R5
        tst.b   R5

        jn      ItIsNegative
        jmp     ItIsPositive

ItIsNegative:
        mov     #1, R4
        jmp     main

ItIsPositive:
        mov     #1, R4
        jmp     main
```

We now insert a dummy value of +1 to begin. The `tst` instructions subtracts 0 from R5 and alters the status register.

Since our dummy value is +1, the `tst` results in N=0. The `jn` jump is not taken. Since it is not negative, it must be positive so we can use a jmp for the positive condition.

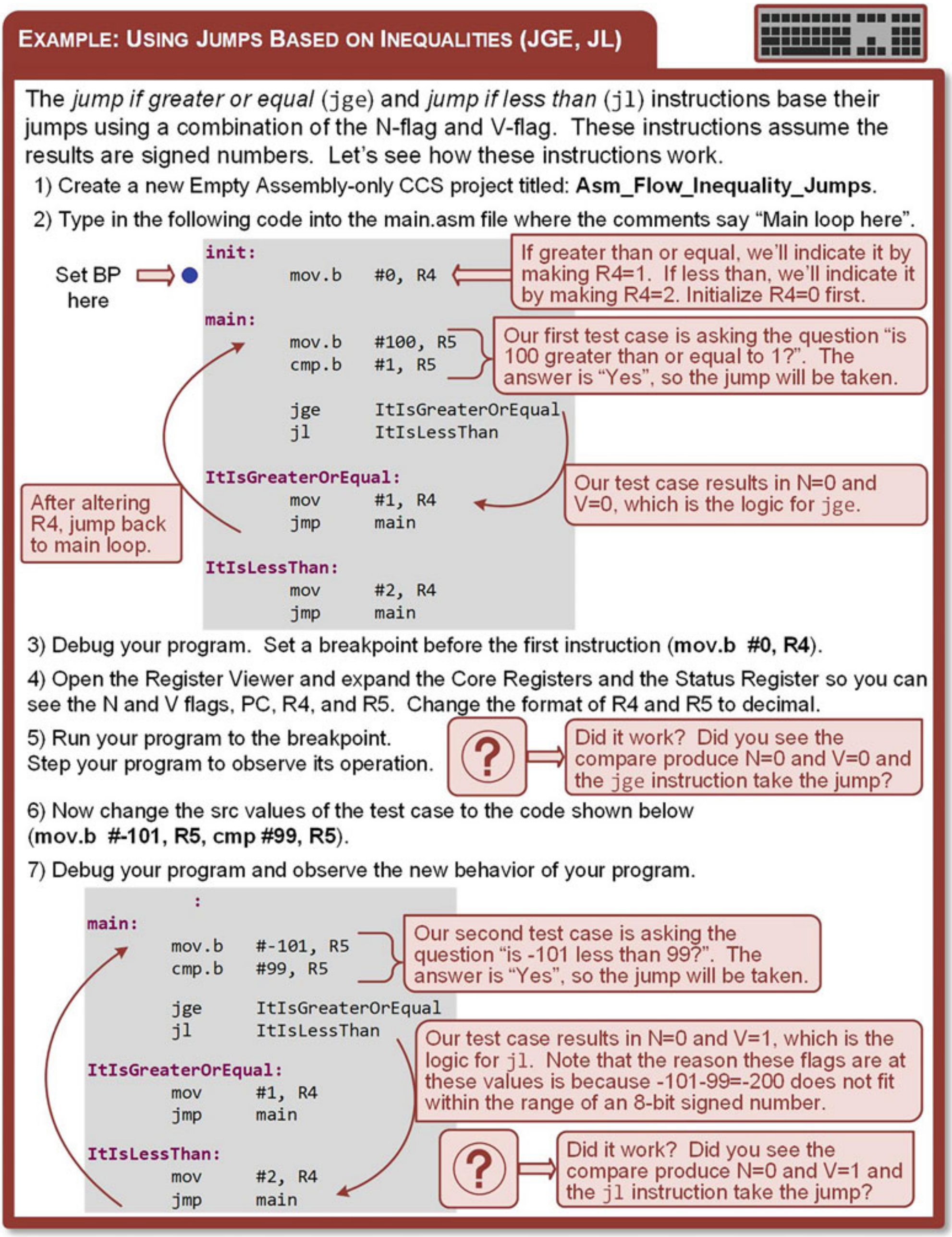
After altering R4, jump back to main loop.

? Did it work? Did you see the test produce N=0 and the `jn` instruction was skipped?

Example 8.4
Using jumps based on the negative flag (JN)

8.2.4 Overflow-Based Jumps

The *jump if greater than or equal* (**jge**) and *jump if less than* (**jl**) instructions provide the ability to jump based on inequalities and also to consider two's complement overflow. These jumps use both the N-flag and V-flag and assume the results are signed numbers. The **jge** instruction will jump when ($N \oplus V = 0$). The **jl** instruction will jump when ($N \oplus V = 1$); however, both instructions are easier to understand by simply using their mnemonic description (i.e., $\geq$ and <). Follow Example 8.5 to see how these inequality-based conditional jumps work.



EXAMPLE: USING JUMPS BASED ON INEQUALITIES (JGE, JL)

The *jump if greater or equal* (jge) and *jump if less than* (jl) instructions base their jumps using a combination of the N-flag and V-flag. These instructions assume the results are signed numbers. Let's see how these instructions work.

1) Create a new Empty Assembly-only CCS project titled: **Asm_Flow_Inequality_Jumps**.

2) Type in the following code into the main.asm file where the comments say "Main loop here".

Set BP here

```
init:
        mov.b   #0, R4

main:
        mov.b   #100, R5
        cmp.b   #1, R5

        jge     ItIsGreaterOrEqual
        jl      ItIsLessThan

ItIsGreaterOrEqual:
        mov     #1, R4
        jmp     main

ItIsLessThan:
        mov     #2, R4
        jmp     main
```

If greater than or equal, we'll indicate it by making R4=1. If less than, we'll indicate it by making R4=2. Initialize R4=0 first.

Our first test case is asking the question "is 100 greater than or equal to 1?". The answer is "Yes", so the jump will be taken.

Our test case results in N=0 and V=0, which is the logic for jge.

After altering R4, jump back to main loop.

3) Debug your program. Set a breakpoint before the first instruction (**mov.b #0, R4**).

4) Open the Register Viewer and expand the Core Registers and the Status Register so you can see the N and V flags, PC, R4, and R5. Change the format of R4 and R5 to decimal.

5) Run your program to the breakpoint.
Step your program to observe its operation.

?

Did it work? Did you see the compare produce N=0 and V=0 and the jge instruction take the jump?

6) Now change the src values of the test case to the code shown below (**mov.b #-101, R5, cmp #99, R5**).

7) Debug your program and observe the new behavior of your program.

```
            :
main:
        mov.b   #-101, R5
        cmp.b   #99, R5

        jge     ItIsGreaterOrEqual
        jl      ItIsLessThan

ItIsGreaterOrEqual:
        mov     #1, R4
        jmp     main

ItIsLessThan:
        mov     #2, R4
        jmp     main
```

Our second test case is asking the question "is -101 less than 99?". The answer is "Yes", so the jump will be taken.

Our test case results in N=0 and V=1, which is the logic for jl. Note that the reason these flags are at these values is because -101-99=-200 does not fit within the range of an 8-bit signed number.

?

Did it work? Did you see the compare produce N=0 and V=1 and the jl instruction take the jump?

Example 8.5
Using jumps based on inequalities (JGE, JL)

Concept Check

CC8.2 Is it possible to create the functionality of an unconditional jump using a conditional jump?

A) Yes. You can simply set or clear one of the flags in the SR and then use a conditional jump that is guaranteed to always take. For example:

```
 bic.b  #00000001b, SR   ; clear Carry Flag
jnc                      ; jump if No Carry
```

B) No. The conditional jumps are too complicated. They need to be used exactly like the data sheet says. No messing around.

8.3 Implementing Common Programming Constructs in Assembly

Now that we have covered the majority of the instructions within the MSP430 instruction set, we have the ability to implement any programming behavior that is possible on a single CPU computer. This section provides some examples of how common higher-level programming constructs such as while() loops, for() loops, if/else, and switch/case statements are implemented in assembly.

8.3.1 Implementing While() Loop Functionality

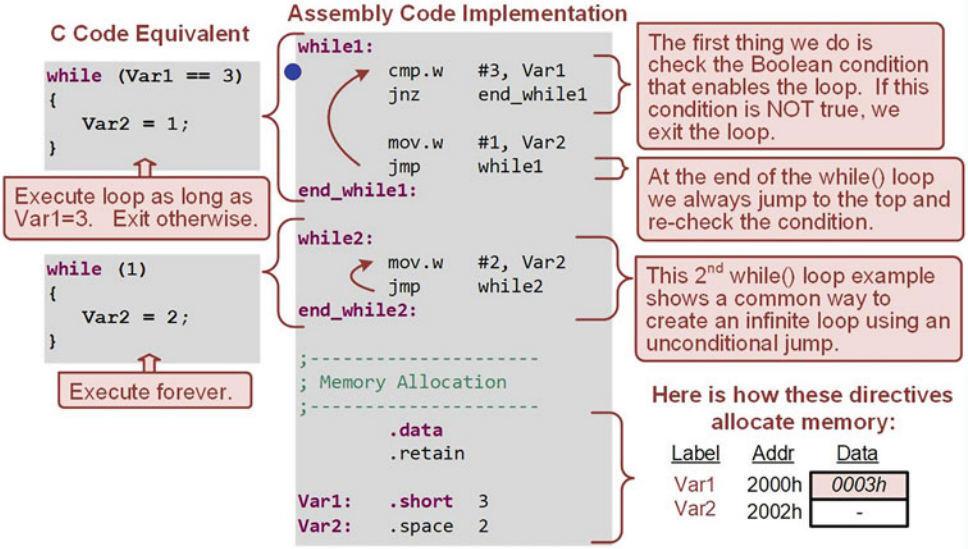
A while() loop is a sequence of statements that will continually execute as long as a Boolean condition at the beginning of the loop is satisfied. In the C programming language, the Boolean condition is inserted within the parenthesis of the while() keyword and the statements to be executed are listed within curly brackets ({}). In assembly, this behavior is implemented with a combination of test, compare, and conditional jump instructions. Follow Example 8.6 to see how while() loop functionality is implemented in assembly.

Example: Implementing While() Loops in Assembly

A while() loop in C will execute the statements within its brackets as long as a Boolean condition is true. A while() loop is implemented in assembly using a combination of test, compare, and conditional jump instructions. Let's see how some while() loops work in assembly.

1) Create a new Empty Assembly-only CCS project titled: **Asm_Flow_While_Loops**.

2) Type in the following assembly code into the main.asm file where it says "Main loop here".



C Code Equivalent

```
while (Var1 == 3)
{
   Var2 = 1;
}
```

Execute loop as long as Var1=3. Exit otherwise.

```
while (1)
{
   Var2 = 2;
}
```

Execute forever.

Assembly Code Implementation

```
while1:
        cmp.w   #3, Var1
        jnz     end_while1

        mov.w   #1, Var2
        jmp     while1
end_while1:

while2:
        mov.w   #2, Var2
        jmp     while2
end_while2:

;--------------------
; Memory Allocation
;--------------------
        .data
        .retain

Var1:   .short  3
Var2:   .space  2
```

The first thing we do is check the Boolean condition that enables the loop. If this condition is NOT true, we exit the loop.

At the end of the while() loop we always jump to the top and re-check the condition.

This 2nd while() loop example shows a common way to create an infinite loop using an unconditional jump.

Here is how these directives allocate memory:

Label	Addr	Data
Var1	2000h	*0003h*
Var2	2002h	-

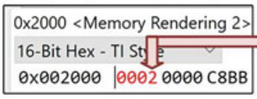
3) Debug your program. Set a breakpoint before the first instruction (**cmp.w #3, Var1**).

4) Open the Memory Browser and go to address 0x2000. You should see a word at 0x2000 initialized to 3 and a word at 0x2002 reserved with no values.

5) Run your program to the breakpoint. Step your program to observe the behavior of the first while() loop.

? Did it work? Your program should stay in the first while() loop because Var1 was initialized to 3.

6) Now you are going to manually change the value of Var1 from 3 to 2 in the memory browser.



```
0x2000 <Memory Rendering 2>
16-Bit Hex - TI Style
0x002000  0002 0000 C8BB
```

Click in the value at address 0x2000, delete the 3, type in 2, press enter. The new value will take effect once you step the program again.

7) Continue stepping your program with the new value of Var1=2.

? Did it work? Your program should have exited the first while() loop and entered the second while() loop where it stayed forever.

Example 8.6
Implementing while() loops in assembly

8.3.2 Implementing For() Loop Functionality

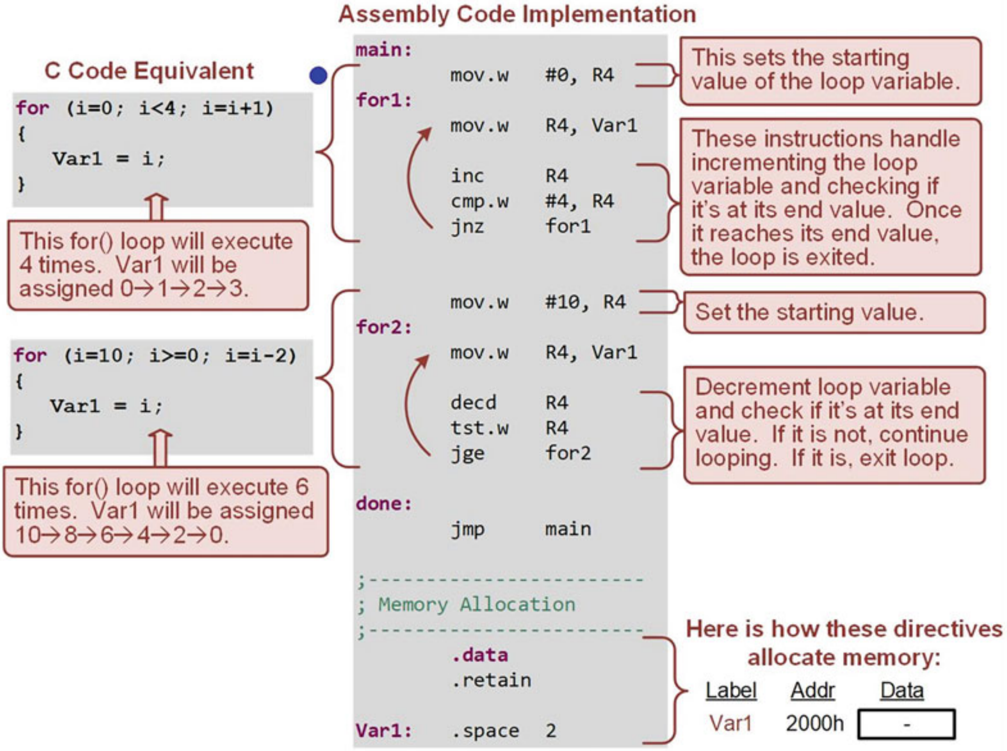
A for() loop is a sequence of statements that will execute a fixed number of times. At the beginning of the for() loop, the number of times to iterate is specified by stating a loop variable, the starting value of the variable, the final value of the variable, and the method that the variable will be incremented/decremented (i.e., increment by 1, decrement by 2, etc.). For() loops are powerful because the loop variable can be used as an offset when accessing blocks of storage. In assembly, for() loop functionality is accomplished using increment/decrement, test, compare, and conditional jump instructions. Follow Example 8.7 to see how for() loop functionality is implemented in assembly.

Example: Implementing For() Loops in Assembly

A for() loop in C will execute the statements within its brackets a specified number of times. A loop variable is typically used to control how many times the loop is executed. A starting value, ending value, and increment/decrement behavior of the loop variable is specified to control the for() loop. A for() loop is implemented in assembly using a combination of inc/dec, test, compare, and conditional jump instructions. Let's see how some for() loops work in assembly.

1) Create a new Empty Assembly-only CCS project titled: **Asm_Flow_For_Loops**.

2) Type in the following assembly code into the main.asm file where it says "Main loop here".



Assembly Code Implementation

```
main:
        mov.w   #0, R4
for1:
        mov.w   R4, Var1

        inc     R4
        cmp.w   #4, R4
        jnz     for1

        mov.w   #10, R4
for2:
        mov.w   R4, Var1

        decd    R4
        tst.w   R4
        jge     for2

done:
        jmp     main

;-----------------------
; Memory Allocation
;-----------------------
        .data
        .retain

Var1:   .space  2
```

This sets the starting value of the loop variable.

These instructions handle incrementing the loop variable and checking if it's at its end value. Once it reaches its end value, the loop is exited.

Set the starting value.

Decrement loop variable and check if it's at its end value. If it is not, continue looping. If it is, exit loop.

C Code Equivalent

```
for (i=0; i<4; i=i+1)
{
   Var1 = i;
}
```

This for() loop will execute 4 times. Var1 will be assigned 0→1→2→3.

```
for (i=10; i>=0; i=i-2)
{
   Var1 = i;
}
```

This for() loop will execute 6 times. Var1 will be assigned 10→8→6→4→2→0.

Here is how these directives allocate memory:

Label	Addr	Data
Var1	2000h	-

3) Debug your program. Set a breakpoint before the first instruction (**mov.w #0, R4**).

4) Open the Memory Browser and go to address 0x2000. You should see a word at 0x2000 reserved with no values.

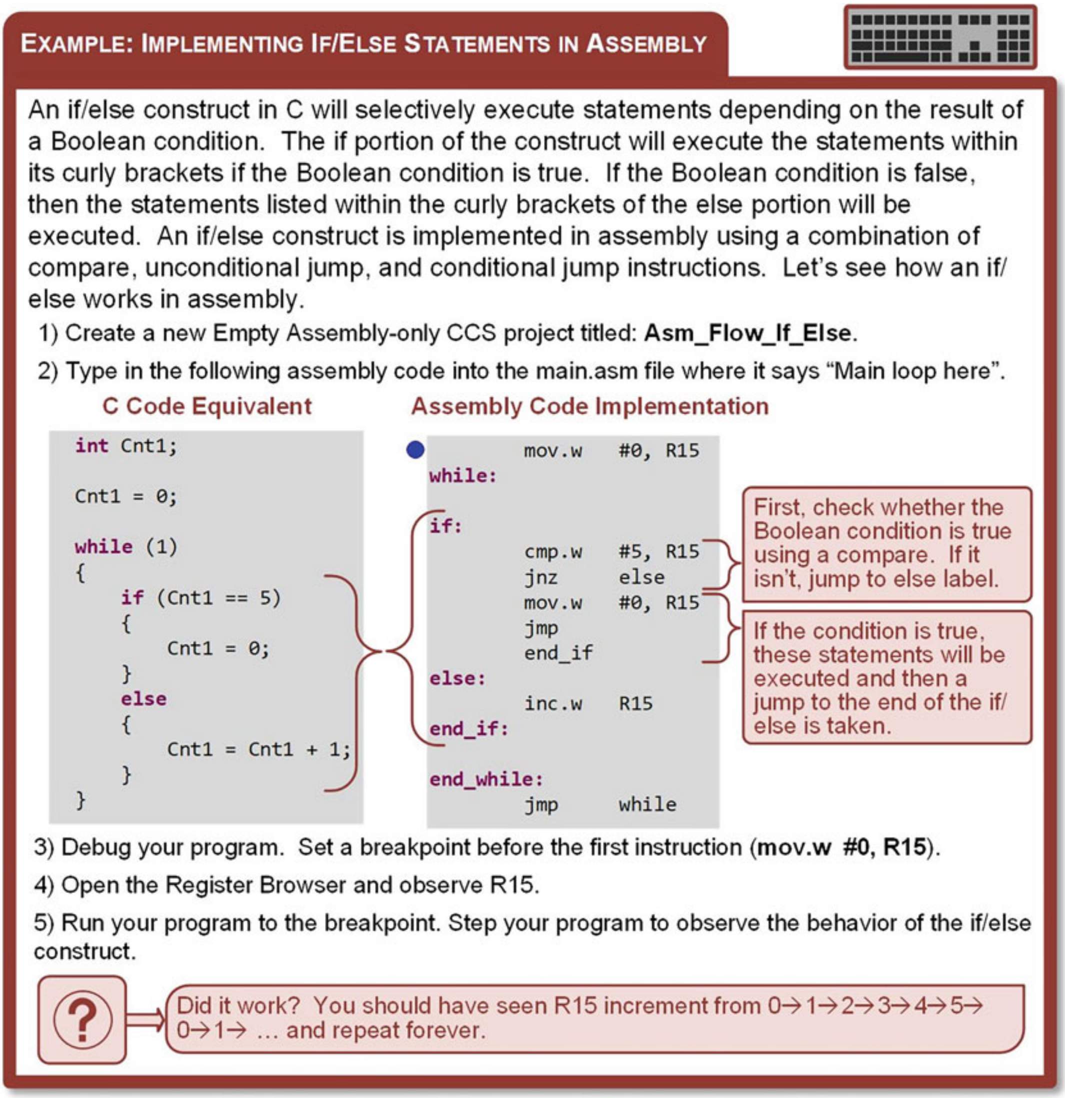
5) Run your program to the breakpoint. Step your program to observe the behavior of the for() loops.

Did it work? Did you see each for() loop execute the expected number of times? Was Var1 updated as expected in each loop?

Example 8.7
Implementing for() loops in assembly

8.3.3 Implementing If/Else Functionality

An if/else is a decision construct that allows statements to be selectively executed based on the result of a Boolean condition. In its simplest form, a Boolean condition is entered after the if portion of the construct. If this condition is true, the statements listed within the subsequent curly brackets will be executed. If the condition is not true, then the first set of statements are skipped, and the statements listed within the curly brackets after the else portion of the construct are executed. If/else statements can be nested to provide the ability to check multiple Boolean conditions. In assembly, if/else functionality is accomplished using compare, unconditional jump, and conditional jump instructions. Follow Example 8.8 to see how if/else functionality is implemented in assembly.



EXAMPLE: IMPLEMENTING IF/ELSE STATEMENTS IN ASSEMBLY

An if/else construct in C will selectively execute statements depending on the result of a Boolean condition. The if portion of the construct will execute the statements within its curly brackets if the Boolean condition is true. If the Boolean condition is false, then the statements listed within the curly brackets of the else portion will be executed. An if/else construct is implemented in assembly using a combination of compare, unconditional jump, and conditional jump instructions. Let's see how an if/else works in assembly.

1) Create a new Empty Assembly-only CCS project titled: **Asm_Flow_If_Else**.

2) Type in the following assembly code into the main.asm file where it says "Main loop here".

C Code Equivalent

```
int Cnt1;

Cnt1 = 0;

while (1)
{
    if (Cnt1 == 5)
    {
        Cnt1 = 0;
    }
    else
    {
        Cnt1 = Cnt1 + 1;
    }
}
```

Assembly Code Implementation

```
            mov.w   #0, R15
while:

if:
            cmp.w   #5, R15
            jnz     else
            mov.w   #0, R15
            jmp
            end_if
else:
            inc.w   R15
end_if:

end_while:
            jmp     while
```

First, check whether the Boolean condition is true using a compare. If it isn't, jump to else label.

If the condition is true, these statements will be executed and then a jump to the end of the if/else is taken.

3) Debug your program. Set a breakpoint before the first instruction (**mov.w #0, R15**).

4) Open the Register Browser and observe R15.

5) Run your program to the breakpoint. Step your program to observe the behavior of the if/else construct.

?

Did it work? You should have seen R15 increment from 0→1→2→3→4→5→0→1→ … and repeat forever.

Example 8.8
Implementing if/else statements in assembly

8.3.4 Implementing Switch/Case Functionality in Assembly

A switch/case statement (also called either just a switchor just a case statement) allows a variable to be tested against a list of values. The first value in the list that matches the variable will result in the execution of statements associated with that value. The functionality is similar to nested if/else statements; however, the syntax is more amenable to large lists of comparisons. In assembly, a switch/case statement is implemented with a sequence of compare instructions, each with an associated conditional jump to a corresponding series of instructions to be executed. Follow Example 8.9 to see how switch/case statements are implemented in assembly.

EXAMPLE: IMPLEMENTING SWITCH/CASE STATEMENTS IN ASSEMBLY

A switch/case statement in C will compare a variable against a list of values. The first value in the list that matches will cause the statement(s) associated with that value to execute. A switch/case statement is implemented in assembly using a sequence of compare statements, each with an associated conditional jump to a location in the program containing the corresponding instructions to be executed for the matched value. Let's see how a switch/case works in assembly.

1) Create a new Empty Assembly-only CCS project titled: **Asm_Flow_Switch_Case**.

2) Type in the following assembly code into the main.asm file where it says "Main loop here".

C Code Equivalent

```
int VarIn = 0;
int VarOut = 0;

while (1)
{
  switch(VarIn) {
    case 0  : VarOut = 0x01;
              break;
    case 1  : VarOut = 0x02;
              break;
    case 2  : VarOut = 0x04;
              break;
    case 3  : VarOut = 0x08;
              break;
    default : VarOut = 0x00;
   }
}
```

Assembly Code Implementation

```
        mov.w   #0, R14   ; VarIn
        mov.w   #0, R15   ; VarOut
while:

switch:
        cmp.w   #0, R14
        jz      case0
        cmp.w   #1, R14
        jz      case1
        cmp.w   #2, R14
        jz      case2
        cmp.w   #3, R14
        jz      case3
        jmp     default
case0:
        mov.w   #0001h, R15
        jmp     end_switch
case1:
        mov.w   #0002h, R15
        jmp     end_switch
case2:
        mov.w   #0004h, R15
        jmp     end_switch
case3:
        mov.w   #0008h, R15
        jmp     end_switch
default:
        mov.w   #0000h, R15

end_switch:

end_while:
        jmp     while
```

These instructions compare the input to a list of values and then jump to the respective case label.

Each case executes the desired instruction(s) and then jumps to the end of the switch statement.

3) Debug your program. Set a breakpoint before the first instruction (**mov.w #0, R14**).

4) Open the Register Browser and observe R14 and R15.

5) Run your program to the breakpoint. Step your program to observe the behavior of the switch/case statement.

6) Manually alter the value of R14 by clicking in the register browser and entering values between 0 to 3. Remember to press enter and step again for these values to take place.

Did it work? As you enter different values for R14, you should see corresponding jumps to the appropriate case label and R15 should be updated accordingly.

Example 8.9
Implementing switch/case statements in assembly

Concept Check

CC8.3 Does a C compiler create assembly code to implement its desired functionality?

A) Yes. All of the functionality in your C program will be converted into a series of assembly code instructions to implement the desired program behavior.

B) No. Compilers don’t understand assembly. They just deal with C.

8.4 Flow Charts

A flow chart is a graphical depiction of the behavior of a program. Flow charts are useful in the design stage of a program as they allow the algorithm to be thought through prior to implementation. There are five basic elements to a flow chart (shown in Fig. 8.1). An oval represents the start and end to a program. A rectangle represents a process, which can be a single instruction or a sequence of instructions that accomplishes a specific task. A diamond represents a decision where the corners of the shape represent different paths the program can take based on the decision. Finally, a subprocess is a rectangle with double sides that represents a sequence of instructions that occurs separate from the main program flow. A subprocess can be a subroutine or a service routine.

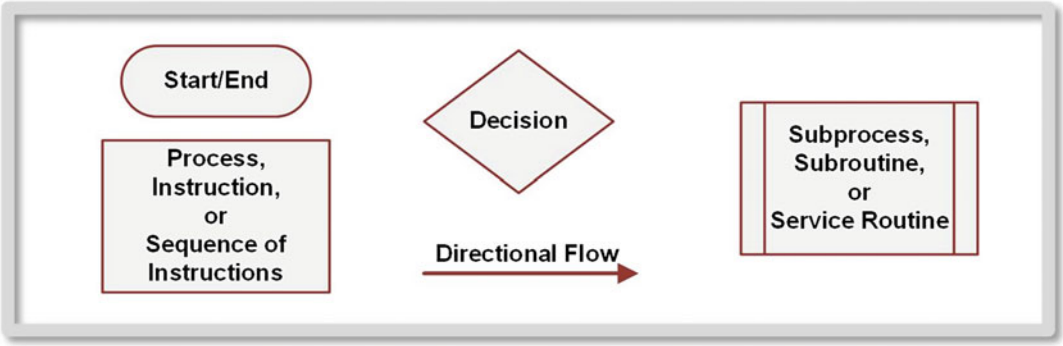
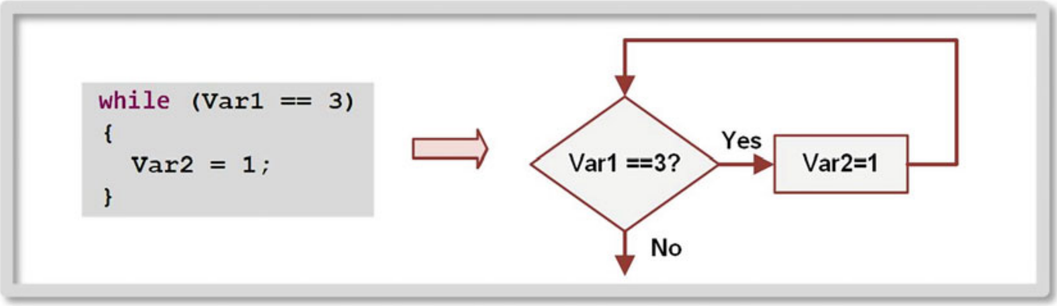

Start/End
Process, Instruction, or Sequence of Instructions
Decision
Directional Flow
Subprocess, Subroutine, or Service Routine

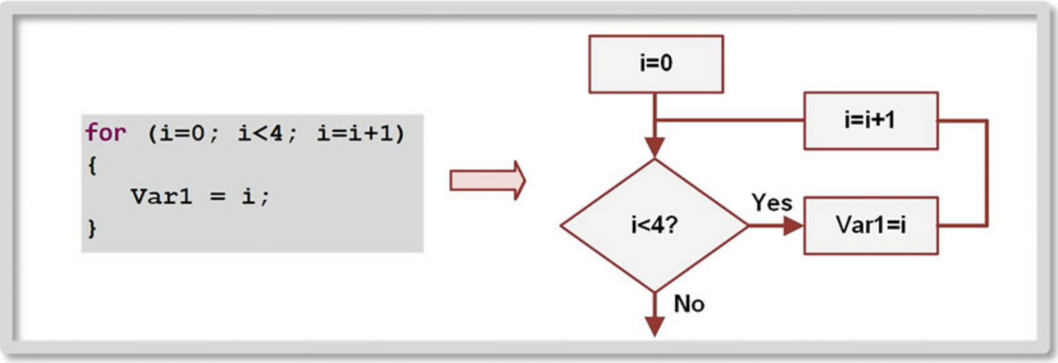
Fig. 8.1
Main elements of a flow chart

Let’s look at some flow charts for common programming constructs. Figures 8.2, 8.3, 8.4, and 8.5 show flow charts for a while() loop, for() loop, if/else, and switch/case, respectively.



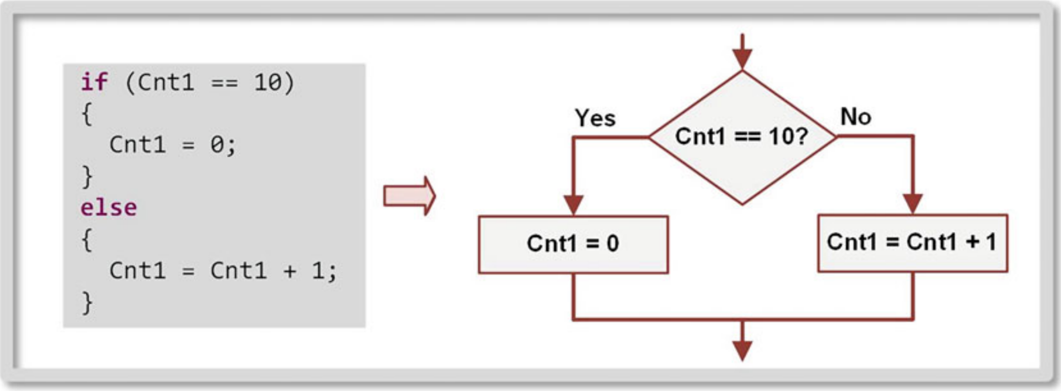
```
while (Var1 == 3)
{
  Var2 = 1;
}
```

Fig. 8.2
Flow chart for a while() loop



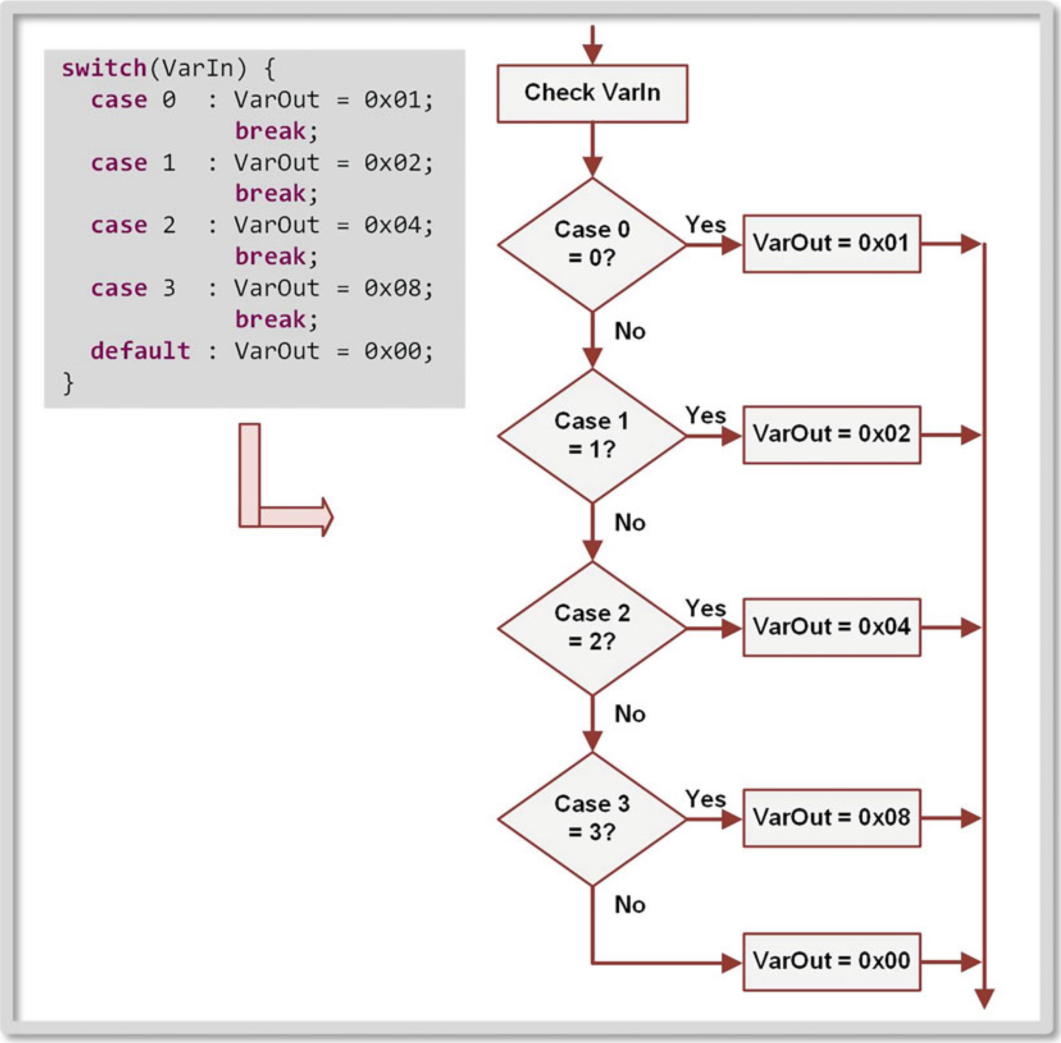
```
for (i=0; i<4; i=i+1)
{
   Var1 = i;
}
```

Fig. 8.3
Flow chart for a for() loop



```
if (Cnt1 == 10)
{
  Cnt1 = 0;
}
else
{
  Cnt1 = Cnt1 + 1;
}
```

Fig. 8.4
Flow chart for an if/else statement



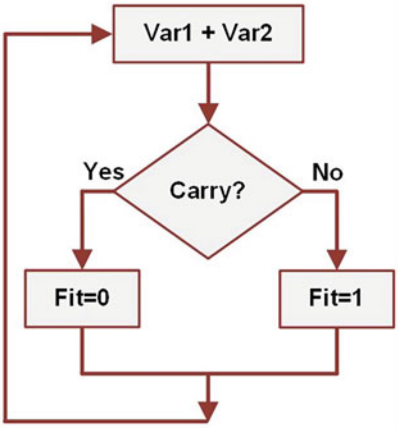
```
switch(VarIn) {
   case 0   : VarOut = 0x01;
              break;
   case 1   : VarOut = 0x02;
              break;
   case 2   : VarOut = 0x04;
              break;
   case 3   : VarOut = 0x08;
              break;
   default  : VarOut = 0x00;
}
```

Fig. 8.5
Flow chart for a switch/case statement

Let's look at an example of creating an assembly program from a flow chart. Follow Example 8.10 to gain experience with this process.

Example: Implementing Assembly Code From a Flow Chart

Create an assembly program that will implement the functionality of the following flow chart. Var1, Var2, and Fit will be 16-bit variables in data memory that we need to reserve. Var1 and Var2 are updated by another process, but our program will need to update Fit per the flow chart.



1) Create a new Empty Assembly-only CCS project titled: **Asm_Flow_DesignFromFlowChart**.

2) Let's think about the functionality of this flow chart. First, we'll need to reserve 3x words of data memory called Var1, Var2, and Fit. Then in the main program, we'll add Var1 and Var2. We then need to check if the C-flag was asserted. We can use the jc and jnc conditional jumps for that. Those jumps will go to labels that will set the value of Fit. The final part of this flow chart is to repeat the loop forever. We can accomplish that by doing an unconditional jump to the start of the program, which we'll call "main".

3) Type in the following assembly code into the main.asm file where it says "Main loop here".

4) Debug your program. Set a breakpoint before the first instruction.

5) Open the Memory Browser and observe Var1, Var2, and Fit starting at address 0x2000.

6) Run your program to the breakpoint. Step your program to observe its behavior.

7) Manually change the values of Var1 and Var2 to generate a carry. Try setting Var1=FFFFh and Var2=1 to generate a carry.

Did it work? Are you able to see the variable Fit set to the correct value when a carry is generated? Does it make sense how the assembly code was created based on the flow chart?

```
main:
        mov.w   Var1, R4
        add.w   Var2, R4
        jc      Carry
        jnc     NoCarry
Carry:
        mov.w   #0, Fit
        jmp     done
NoCarry:
        mov.w   #1, Fit
        jmp     done
done:
        jmp     main

;-----------------------
; Memory Allocation
;-----------------------
        .data
        .retain

Var1:   .space  2
Var2:   .space  2
Fit:    .space  2
```

Example 8.10
Implementing assembly code from a flow chart

Concept Check

CC8.4 Creating flow charts seems like extra work. Why would I take the time to do one?

A) It really is. You should just code without thinking and hope it works.

B) Good coders don't need to think. They just type fast.

C) Most coders aren't good at drawing, so flow charts aren't helpful anyway.

D) A flow chart allows the developer to think through the algorithm needed to solve the program before getting into the implementation details of the code.

Summary

- Branches and jumps are instructions that alter the program counter. This allows a program to run loops and make decisions about which instructions to execute and which to skip.
- An unconditional branch (`br`) and jump (`jmp`) will always alter the program counter.
- Branches can point PC to the entire memory range of the MSP430FR2355.
- Jumps apply a 10-bit offset to the program counter that is calculated by determining the difference between the current address of PC and the address of the label to jump to. Using a 10-bit offset limits jumps to only the −511 to +512 addresses around the current value of the program counter.
- Conditional jumps alter the PC only when certain conditions are present on the status flags in the status register.
- There are conditional jumps based on the carry flag (`jc`, `jnc`), the zero flag (`jz`, `jnz`), and the negative flag (`jn`).
- There are also conditional jumps that execute based on equalities that use both the N- and Z-flags. There is a branch if greater or equal jump (`jge`), which executes when $N \oplus V = 1$. There is also a branch if less than (`jl`), which executes when $N \oplus V = 0$. The `jge` and `jl` branches assume the operands are treated as signed numbers.
- Traditional programming constructs such as while() loops, for() loops, if/else, and switch/case statements can be implemented using compare, test, and conditional jump instructions.
- Flow charts provide a graphical method to describe the flow of a program. Flow charts are used during the program design phase to decide how an algorithm will be implemented. Once the flow chart is created, the assembly code can be directly implemented from the chart.

Exercise Problems

Section 8.1: Unconditional Jumps and Branches

8.1.1 What is the main difference between an unconditional branch and an unconditional jump?

8.1.2 When would you encounter an assembly error titled "jump out of range"?

8.1.3 If you encounter a "jump out of range" error, what is the resolution?

8.1.4 Why are labels so important to branch and jump instructions?

8.1.5 What is the range of addresses that a jump can reach from the current PC value?

Section 8.2: Conditional Jumps

8.2.1 What instruction will perform a conditional jump when $C = 1$?

8.2.2 What instruction will perform a conditional jump when $Z = 1$?

8.2.3 What instruction will perform a conditional jump when $N = 1$?

8.2.4 What instruction will perform a conditional jump when $C = 0$?

8.2.5 What instruction will perform a conditional jump when $Z = 0$?

8.2.6 What instruction will perform a conditional jump when $N \oplus V = 0$?

8.2.7 What instruction will perform a conditional jump when $N \oplus V = 1$?

Section 8.3: Implementing Common Programming Constructs

8.3.1 How does a while() loop function?

8.3.2 How does a for() loop function?

8.3.3 How does an if/else statement function?

8.3.4 How does a switch/case statement function?

8.3.5 In a while() loop, what are typically the first two instructions used to check the Boolean condition for the loop?

8.3.6 In a for() loop, what do the last sequence of instructions do that control the number of iterations of the loop?

8.3.7 In an if/else statement, why is it critical to have an unconditional jump instruction?

8.3.8 In a switch/case statement, how does the number of compare statements relate to the number of cases in the decision statement?

Section 8.4: Flow Charts

8.4.1 Which high-level programming construct does the flow chart in Fig. 8.6 represent?

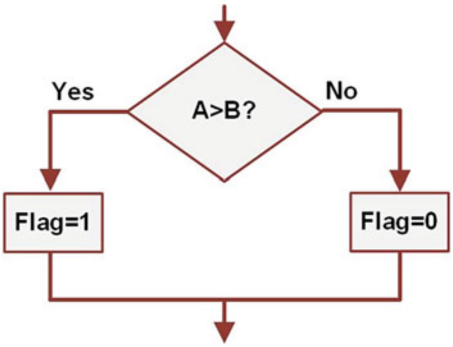


Fig. 8.6
Flow chart 1

8.4.2 Which high-level programming construct does the flow chart in Fig. 8.7 represent?

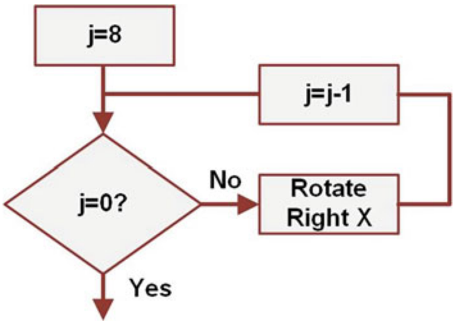


Fig. 8.7
Flow chart 2

8.4.3 Which high-level programming construct does the flow chart in Fig. 8.8 represent?

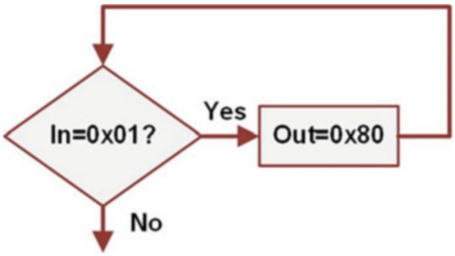


Fig. 8.8
Flow chart 3

8.4.4 Which high-level programming construct does the flow chart in Fig. 8.9 represent?

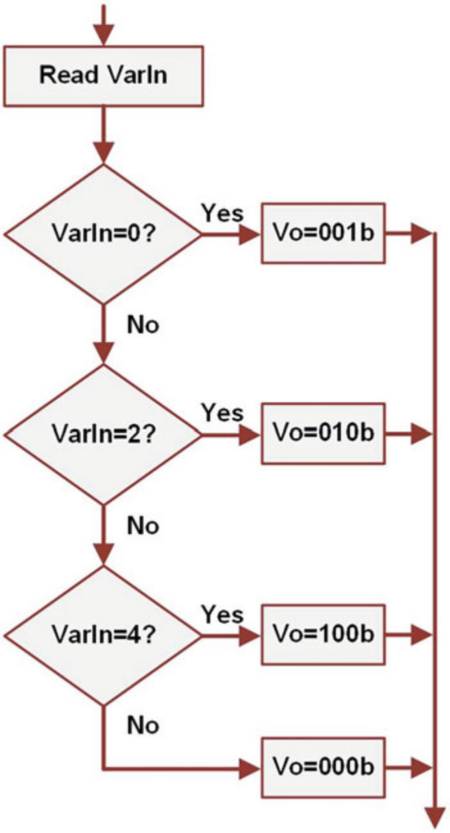


Fig. 8.9
Flow chart 4

Chapter 9: Digital I/O

This chapter introduces the digital I/O system for the MSP430 [1–3]. Digital I/O provides a way for an MCU to directly read or write logic levels to pins on the device package. The digital I/O system is highly versatile as it can be used to implement nearly any functionality a programmer wishes. We will begin by looking at the general characteristics of digital I/O on the MSP430 and how they are configured. Then, we will look at specific examples of writing to and reading from pins on the MCU that are connected to LEDs and buttons on the MSP430FR2355 LaunchPad™.

Learning Outcomes—After completing this chapter you will be able to:

9.1 Describe the procedure to set up a digital I/O pin on the MSP430.
9.2 Design a program to write logic levels to an output pin on the MCU.
9.3 Design a program to read logic levels from an input pin on the MCU.

9.1 The MSP430 Digital I/O System

The full MSP430 architecture supports up to 12-, 8-bit I/O ports; however, very few MCU devices have these many I/O ports implemented. We will focus specifically on the MSP430FR2355 MCU so that so we can look at specific digital I/O coding examples. The MSP430FR2355 contains six I/O ports labeled P1, P2, P3, P4, P5, and P6. P1-P4 are 8-bit ports. P5 is 5 bits. P6 is 7nbits. This gives 44 total digital I/O available on the MCU package. Figure 9.1 shows the MSP430FR2355TPT package pinout

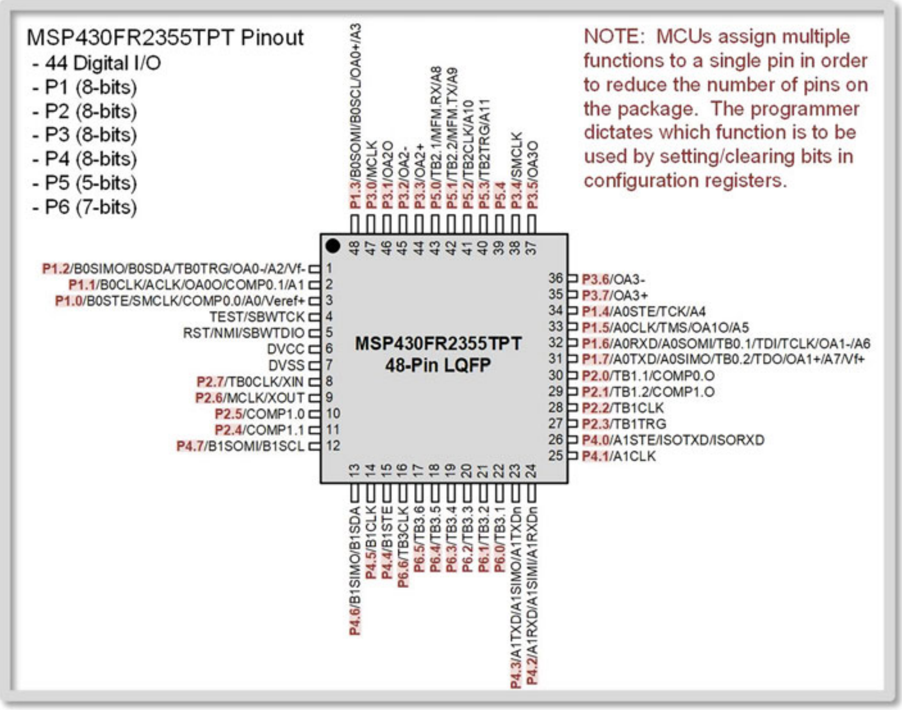
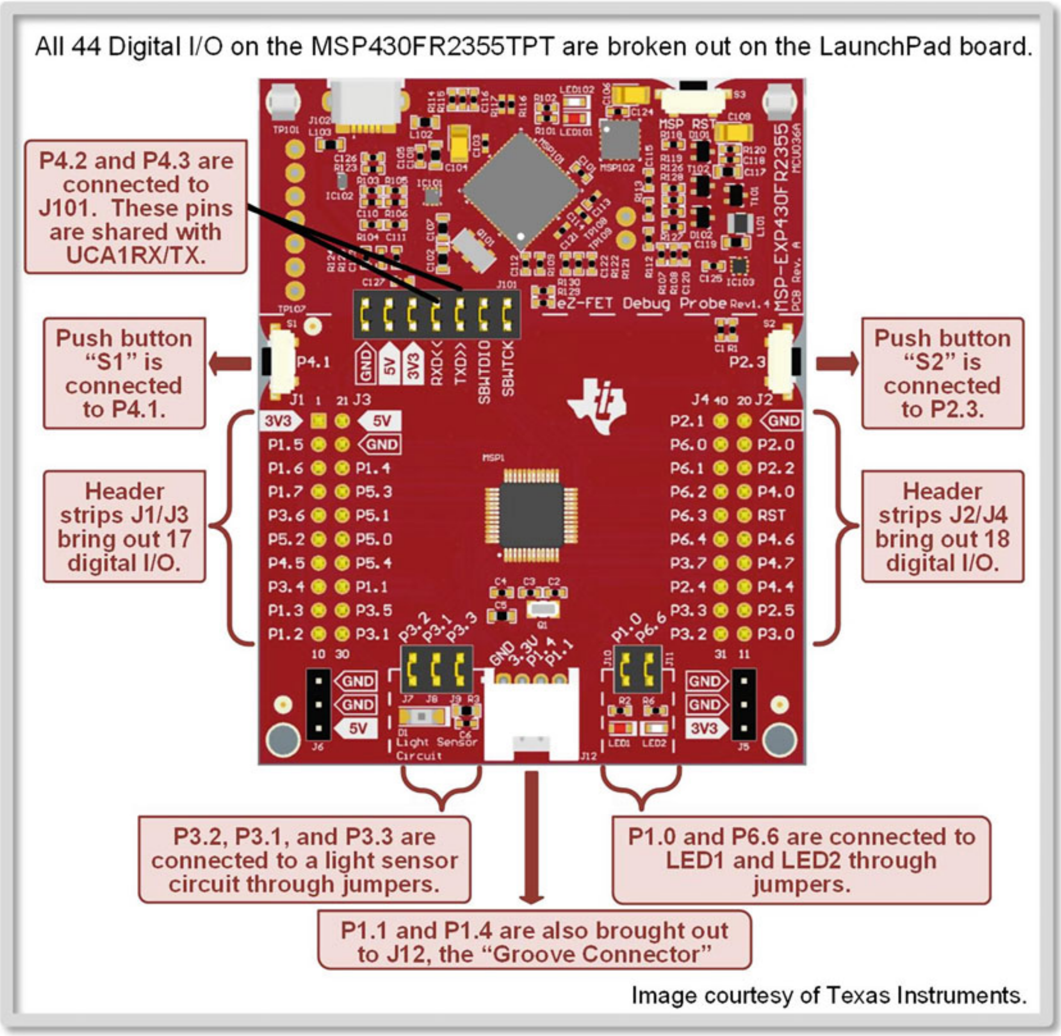


Fig. 9.1
Digital I/O breakout on the MSP430FR2355TPT package

B. J. LaMeres, *Embedded Systems Design using the MSP430FR2355 LaunchPad™*,
https://doi.org/10.1007/978-3-031-20888-1_9

highlighting the digital I/O assignments. Recall that MCUs share functionality on pins in order to reduce the size and cost of the device.

Each of the digital I/O from the MSP430FR2355TPT MCU is broken out to a variety of external circuits on the MSP430FR2355 LaunchPad™ board. Figure 9.2 shows the details of the MCU digital I/O breakout on the LaunchPad™.



All 44 Digital I/O on the MSP430FR2355TPT are broken out on the LaunchPad board.

P4.2 and P4.3 are connected to J101. These pins are shared with UCA1RX/TX.

Push button "S1" is connected to P4.1.

Header strips J1/J3 bring out 17 digital I/O.

Push button "S2" is connected to P2.3.

Header strips J2/J4 bring out 18 digital I/O.

P3.2, P3.1, and P3.3 are connected to a light sensor circuit through jumpers.

P1.0 and P6.6 are connected to LED1 and LED2 through jumpers.

P1.1 and P1.4 are also brought out to J12, the "Groove Connector"

Image courtesy of Texas Instruments.

Fig. 9.2
Digital I/O breakout on the MSP430FR2355 LaunchPad™ Development Kit

The MSP430 provides the ability to access ports using 16-bit operations using the labels PA, PB, and PC. PA represents the combination of P1:P2. PB represents the combination of P3:P4. PC represents the combination of P4:P5. In this text, we will always access the ports using 8-bit operations (.b) as it is much simpler to understand their operation and write programs for.

Each bit within every port is independently configurable with a handful of useful settings. First, each bit can be configured to either be an input or an output. Additionally, bits that are programmed to be an input can have an optional pull-up or pull-down resistor. Ports 1–4 can also contain edge-triggered interrupts with selectable edge sensitivity (LOW-to-HIGH or HIGH-to-LOW). Note that interrupts will be covered in Chap. 11, so we will not go into those details now. The digital I/O system contains a set of registers for each port that facilitate configuration, reading, and writing. The terminology used within the data sheets for the MSP430 attempts to generalize the register names so that they apply to all of the ports. A lower-case "x" is used to denote a number that can take on any value between 1 and 5 and represents a port number. For example, if the documentation stated "Px can be configured to be both an input and output," this means all ports from P1 to P5 can be configured to be both an input and output. The rationale for this terminology will become clearer as the configuration registers are explored. There are six key configuration registers for each port within the MSP430FR2355 MCU: PxDIR, PxIN, PxOUT, PxREN, PxSEL0, and PxSEL1.

9.1.1 Port Direction Registers (PxDIR)

The port direction registers dictate whether the port bits are configured as inputs or outputs. Each bit can be configured independently. The logic for PxDIR is as follows:

- Bit = 0: Pin is an input (default).
- Bit = 1: Pin is an output.

9.1.2 Port Input Registers (PxIN)

Each bit within the PxIN registers represents the logic levels at the input signal pins. The bits within this register use positive logic, meaning that a 0 = low and 1 = high. These registers are read-only. In order to *read* from an input port bit, a program can either move the information into a CPU register or do bit compares on the PxIN memory location.

9.1.3 Port Output Registers (PxOUT)

Each bit within the PxOUT registers is the value to be output to the port's signal pin when the bit is configured to be an output. In order to *write* to an output port bit, a program can move information into the PxOUT register or perform bit set/clear operations on the PxOUT address location.

When a port bit is configured as an input, the PxOUT register has a secondary role, which is to dictate the polarity of an optional pull-up/down resistor (see next section).

9.1.4 Port Pull-up or Pull-down Resistor Enable Registers (PxREN)

When a port bit is configured as an input, an optional pull-up or pull-down resistor can be attached to the input pin. This resistor resides within the MCU, so no external components are needed. Pull-up/pull-down resistors are useful when the external circuit connected to the MCU input has states that are not driven to a known logic level. A pull-up/pull-down resistor can produce a known value for these cases.

One common use of pull-up/pull-down resistors is with *single-pole, single-throw* (SPST) *switches*. SPST switches are the simplest form of a switch and are very common in embedded systems. An SPST switch has one input and one output. When the switch is open, the input and output are *not* connected. When the switch is closed, the input and output *are* connected. When using an SPST switch, the input is typically connected to ground to provide a logic low when the switch is closed; however, when the switch is open the output is not connected to anything so the MCU sees an indeterminant logic level. Adding a pull-up resistor connected to the power supply can provide a logic high when the switch is open. This provides two determinate states coming from the switch corresponding to it being open or closed. Pull-up and pull-down resistors are typically very large (10kΩ → 1 MΩ) so that when the switch is closed, the current that flows from the power supply to the ground through the resistor and switch is minimal. Figure 9.3 shows some configurations of the SPST switch and the use of a pull-up resistor to avoid indeterminant states.

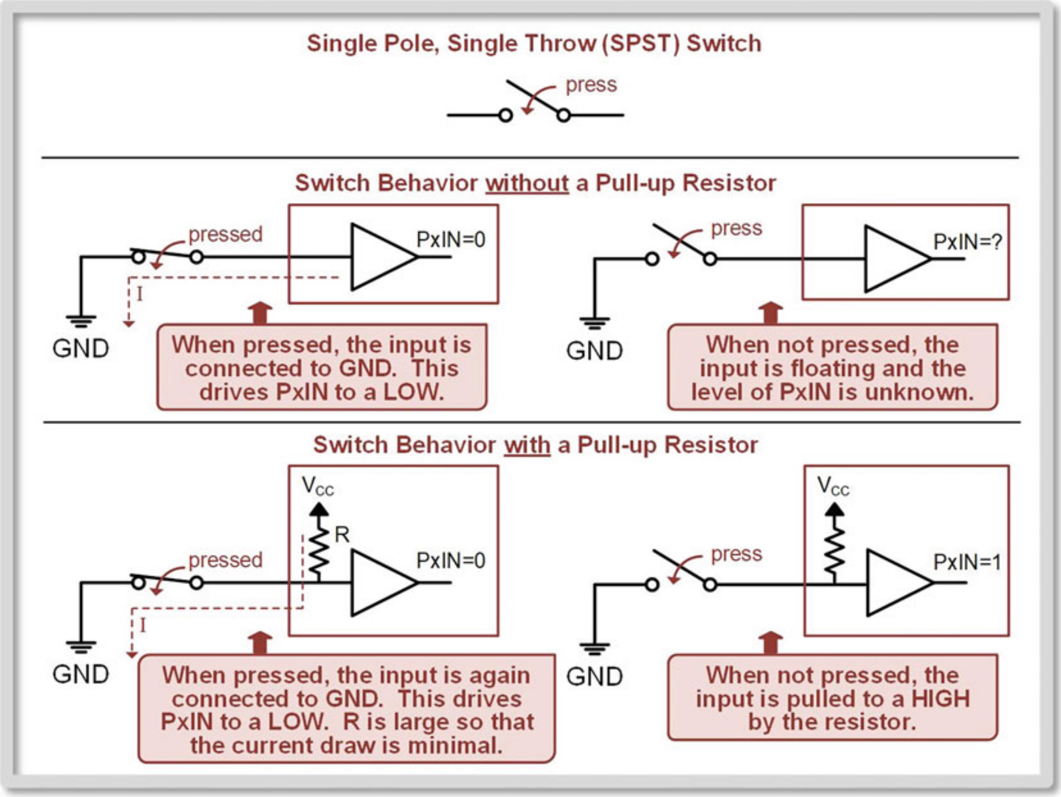

Single Pole, Single Throw (SPST) Switch
press
Switch Behavior without a Pull-up Resistor
pressed
PxIN=0
I
GND
When pressed, the input is connected to GND. This drives PxIN to a LOW.
press
PxIN=?
GND
When not pressed, the input is floating and the level of PxIN is unknown.
Switch Behavior with a Pull-up Resistor
Vcc
R
pressed
PxIN=0
I
GND
When pressed, the input is again connected to GND. This drives PxIN to a LOW. R is large so that the current draw is minimal.
Vcc
press
PxIN=1
GND
When not pressed, the input is pulled to a HIGH by the resistor.

Fig. 9.3
The use of pull-up resistors with SPST switches

Whether or not a port input has a pull-up/pull-down resistor is dictated by the bits within the PxREN register. The bits within PxREN control the corresponding bit location within PxIN (i.e., bit 0 of PxREN controls whether bit 0 of PxIN has a pull-up/pull-down resistor). The logic for PxREN is as follows:

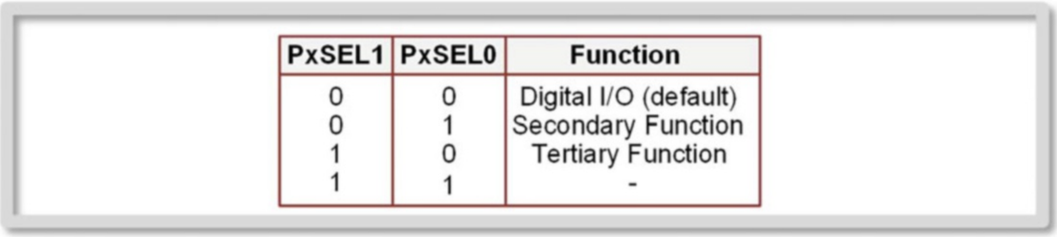
- Bit = 0: Pull-up/pull-down resistor disabled (default).
- Bit = 1: Pull-up/pull-down resistor enabled.

When PxREN enables a pull-up/pull-down resistor, the PxOUT register is used to dictate whether the resistor is a pull-up or pull-down. If PxDIR = 0, making the port bit an input, and PxREN = 1, enabling the pull-up/pull-down resistor, then the PxOUT register has the following logic:

- PxOUT = 0: Insert a **pull-down** resistor.
- PxOUT = 1: Insert a **pull-up** resistor.

9.1.5 Port Function Select Registers (PxSEL1 and PxSEL0)

All of the above settings are used for all of the functions that may use a particular pin on the MCU package as shown in Fig. 9.1. We use the Port Function Select (PxSEL) registers to tell the MCU which function to use, including whether to make the signal pin a digital input/output. The MSP430FR2355 has more than two functions assigned to most of its pins, so it requires two bits to control the function selection. There are two registers, PxSEL1 and PxSEL0, that hold the two selection bits. The logic for these registers is shown in Table 9.1. Note that each signal pin has digital I/O as its default selection (i.e., PxSEL1:PxSEL0 = 00). To see what the secondary and tertiary functions are for each pin, you must look in the device-specific data sheet for the MCU.



PxSEL1	PxSEL0	Function
0	0	Digital I/O (default)
0	1	Secondary Function
1	0	Tertiary Function
1	1	-

Table 9.1
Port register select logic (PxSEL1:PxSEL0)

9.1.6 Digital I/O Enabling After Reset

When an MCU is first powered up, or reset by the user, all digital I/O are put into a high-impedance mode with Schmitt triggers in order to reduce power consumption and avoid any errant current flow. After configuring the digital I/O using the configuration registers (PxDIR, PxREN, PxOUT if applicable, and PxSEL1:PxSEL0), the digital I/O must be taken out of low-power inhibit mode. This is accomplished by clearing the LOCKLPM5 bit in the PM5CTL0 register. This bit is part of the power management module. At this point in this textbook we do not need to understand the exact details of this system; all we need to know is that the digital I/O doesn't work unless this bit is cleared and the system is taken out of low-power inhibit. The steps to fully set up a digital I/O for use can be summarized in the following steps:

- Initialize configuration registers: PxDIR, PxREN, PxOUT if applicable, and PxSEL1:PxSEL0.
- Clear the LOCKLPM5 bit in the PM5CTL0 register.
- Your program may now start using the PxIN or PxOUT registers.

Programmers will often take advantage of the reset conditions of the configuration registers when setting up digital I/O. As a case in point, consider setting up a signal pin as a digital input without a pull-up/pull-down resistor. Upon reset, the default value of PxDIR = 0, which defaults the pin to an input. Upon reset, the default value of PxREN = 0, which defaults the pin to *not* have a pull-up/pull-down resistor. Finally, upon reset the values of PxSEL1/PxSEL0 = 00, which defaults the function select for the pin to be digital I/O. So, all that a programmer actually needs to do is clear the LOCKLPM5 bit in the PM5CTL0 register to take the digital I/O system out of low-power mode and all pins are configured as inputs without pull-up/pull-down resistors.

The level of explicit configuration of digital I/O can vary upon coding expectations, typically set by the programmer's organization. Some organizations create coding standards that require each configuration bit to be explicitly set or cleared upon startup. This is done regardless of whether the explicit altering of the configuration bit matches its reset value (i.e., the reset value is 0, but we still do a bit clear to ensure it is a 0). This can be advantageous for two reasons: first, there is no risk that an oversight in understanding the default values in the data sheet occurs; second, it makes the program highly readable by subsequent programmers. This approach can be simultaneously disadvantageous. The explicit instructions that make sure every bit is configured as desired, regardless of whether the desired value matches the reset value, can lead to excessive startup times. Additionally, the program code can become excessively large and use up the program memory.

In this text, an approach is taken that balances the two initialization extremes while emphasizing the core concepts of the digital I/O system. We will always configure the PxDIR register to explicitly show that we are setting the direction of a bit to be used by the program. We will only configure PxREN if the input uses a pull-up/pull-down resistor. That way if a resistor is used it is obvious, and if it doesn't, there is no mention of PxREN in the program. We will not explicitly configure PxSEL1:PxSEL0 to select digital I/O as the function for the pin as it is a common assumption that upon reset, all MCU pins default to the digital I/O function.

9.1.7 Using Literal Definitions from the MSP430.H Header File

Each of the configuration registers just described has a unique memory address within the MSP430 address map. Looking up and keeping track of these exact address values can be time-consuming and overwhelming. Even something as simple as setting up a single pin to be a digital input with a pull-up resistor requires the programmer to look up multiple absolute addresses including PxDIR, PxREN, and PxIN. Not only is this an excessive amount of work, but using absolute addressing makes the program unusable on other MCUs since other MSP430 variants may have slightly different memory maps.

To eliminate the need for the programmer to look up the absolute address of each register within the memory map, Code Composer Studio automatically includes a header file with literal names defined for the memory addresses. This header file also contains useful literals for specific bit masks (i.e., BIT0 = 01 h and Z = 02 h). When creating a new project in CCS for the MSP430FR2355, a header file named **msp430.h** is included. This generic header file points to a device-specific header file dictated by the MCU settings entered when creating the CCS project. In the case of the MSP430FR2355 MCU we are discussing in this book, the device header file that will ultimately be used is named "msp430fr2355.h." In CCS, this file can be found by expanding the "Includes" folder within the CCS project. It is important to know where this file is because it contains the exact spelling of the register names and abbreviations for the memory map of the MSP430FR2355 MCU we are using. In subsequent sections, names from this header file will be used in the programming examples.

Concept Check

CC9.1 Why is it a good idea for the digital I/O system to be disabled and in low-power mode upon startup?

A) It isn't. All it does is cause the programmer to spend time figuring out how to turn it on. I would prefer if by default all the pins were configured as outputs and just worked.

B) Since the MCU can be used in any type of application, we never know what is going to be physically connected to the pins of the package. Disabling the digital I/O system upon startup avoids accidentally damaging external circuitry by inadvertently driving a logic level into it.

C) It makes it difficult to program so the developers are forced to take an embedded systems class.

D) The entire MCU is disabled upon startup, which includes the digital I/O.

9.2 Digital Output Programming

Let's now look at writing a program that will configure a pin as a digital output on the MSP430FR2355. There are two user LEDs provided on the LaunchPad™ board labeled LED1 and LED2. LED1 is connected to bit 0 of port 1 and LED2 is connected to bit 6 of port 6. The shorthand for these I/O locations is P1.0 and P6.6. Let's look at how to drive LED1. Following the steps outlined in the prior section, configuring this port as an output requires the following steps:

- Initialize configuration registers:
  -P1DIR bit 0 = 1; Configure P1.0 as an OUTPUT
  Note: For an output, this is all we need to do (relying on PxSEL1:PxSEL0 = 00).
- Clear LOCKLPM5 in PM5CTL0 register.

If we open the msp430fr2355.h header file, we can see that there are literals defined for the P1DIR and PM5CTL0 register addresses. We can also see that there is a bit mask defined for BIT0 (0x0001) and LOCKLPM5 (0x0001). Using the names defined in the header file, the assembly code to accomplish this configuration is:

```
bis.b   #BIT0, P1DIR;       Set P1.0 as an output. P1.0 = LED1
bic.b   #LOCKLPM5, PM5CTL0; Disable Digital I/O low-power default
```

After these two lines of code, the output is now ready to be used. From the msp430fr2355.h header file, we can see that the register name for the port outputs is P1OUT. This name can be used when writing to the port. Since literal names from the header file will be directly substituted into the main.asm as their numeric values, we need to use an & in front of the names to treat the numbers as absolute addresses. Follow Example 9.1 to see the actual code to turn on and off LED1 on the LaunchPad™ board. Note that upon reset, P1OUT is indeterminant. That means that the first time through the main loop, LED1 may be in either the ON or OFF states.

Example: Using a Digital Output to Drive LED1

LED1 on the LaunchPad™ board is connected to P1 of the MSP430FR2355 MCU. To turn the LED on and off, we setup P1.0 as a digital output and then simply write low and high logic levels to the pin. Let's see how this works.

1) Create a new Empty Assembly-only CCS project titled: **Asm_Dig_IO_Outputs_n_LEDs**.

2) Type in the following code into the main.asm file where the comments say "Main loop here".

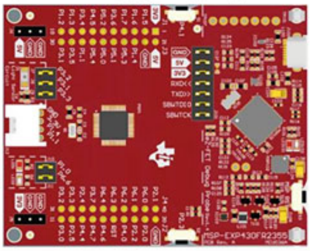
```
init:
        bis.b   #BIT0, &P1DIR       ; Set P1.0 as an output.  P1.0 = LED1
        bic.b   #LOCKLPM5, &PM5CTL0 ; Disable Digital I/O low-power default

main:
        bis.b   #BIT0, &P1OUT       ; Turn LED1 ON
        bic.b   #BIT0, &P1OUT       ; Turn LED1 OFF

        jmp     main
```

The & is needed when using the literal names from the msp430.h header file as the numeric values are directly substituted into main.asm.

3) Debug your program. Set a breakpoint before the first instruction (**bis.b #BIT0, P1DIR**).

4) Run your program to the breakpoint. Step your program to observe its operation.

Remember LED1 is located here on the LaunchPad™ board.



These LEDs are connected to the debugger chip indicating the status of download and debug.

Did it work? You should see LED1 continually turn on and off as you step your program through the main loop.

Example 9.1
Using a digital output to drive LED1 on the LaunchPad™ board

The CCS debugger allows the port registers to be viewed during debug. This is a useful tool when configuring a digital I/O pin for the first time. Often when a digital I/O is not configured correctly, the result is that the I/O pin simply does not work. When this occurs, the only way to figure out what is going on is to go into the Register Viewer and make sure the configuration registers are set up as desired. Follow Example 9.2 to experiment with the configuration registers in the Register Viewer.

Example: Viewing Port Registers in the Debugger

All of the configuration registers for the MSP430FR2355 ports can be observed in the register viewer of CCS during debug. Let's see how these look.

1) Open the project from the last example titled **Asm_Dig_IO_Outputs_n_LEDs**. If you didn't do the last example, create a new Empty Assembly-only CCS project of this name.

2) If necessary, type in the following code into the main.asm file where the comments say "Main loop here".

```
init:
        bis.b   #BIT0, &P1DIR       ; Set P1.0 as an output.  P1.0 = LED1
        bic.b   #LOCKLPM5, &PM5CTL0 ; Disable Digital I/O low-power default

main:
        bis.b   #BIT0, &P1OUT       ; Turn LED1 ON
        bic.b   #BIT0, &P1OUT       ; Turn LED1 OFF

        jmp     main
```

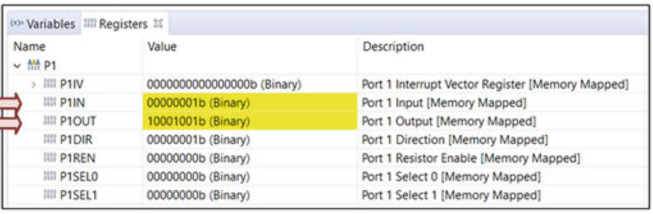
3) Debug your program. Set a breakpoint before the first instruction (**bis.b #BIT0, P1DIR**). Run to the breakpoint.

4) In the Register Viewer and scroll down. Expand the P1 register. You will see all of the configuration registers for this port and their current settings. Change the format to binary. Note that there are additional configuration registers for P1 that we haven't covered yet.

5) Step your program through the main loop and observe how P1.0 is set and cleared and how this corresponds to LED1 turning on and off.

Each time P1.0 changes values, it is observed in P1OUT.

Note that bit 0 of P1IN mirrors bit 0 of P1OUT.



Name	Value	Description
P1		
P1IV	0000000000000000b (Binary)	Port 1 Interrupt Vector Register [Memory Mapped]
P1IN	00000001b (Binary)	Port 1 Input [Memory Mapped]
P1OUT	10001001b (Binary)	Port 1 Output [Memory Mapped]
P1DIR	00000001b (Binary)	Port 1 Direction [Memory Mapped]
P1REN	00000000b (Binary)	Port 1 Resistor Enable [Memory Mapped]
P1SEL0	00000000b (Binary)	Port 1 Select 0 [Memory Mapped]
P1SEL1	00000000b (Binary)	Port 1 Select 1 [Memory Mapped]

6) Now manually enter 1's and 0's for P1.0 in the P1OUT register. After each entry, hit return. Notice that LED1 turns on and off based on your entry. This is a useful technique to ensure that the configuration registers are correct and that the LaunchPad^{TM} is working as expected.

? Did it work? Were you able to see bit0 of P1OUT change values as you stepped through your main program? Were you able to manually turn on and off LED1 by typing in values for bit 0 of P1OUT?

Example 9.2
Viewing port registers in the CCS Register Viewer

Concept Check

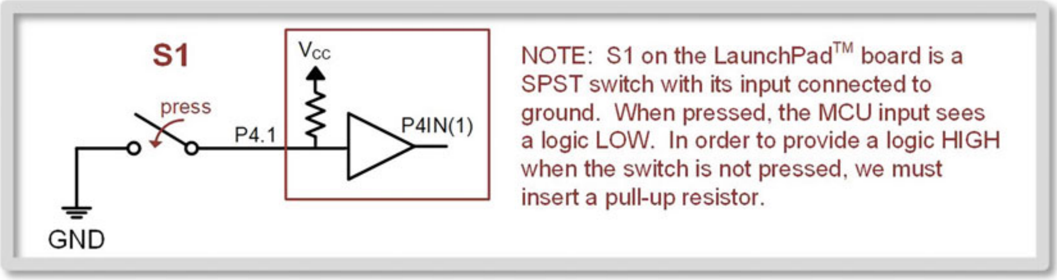
CC9.2 Is it possible to experiment with different port configuration settings by manually typing in values into the Register Viewer?

A) Yes. Any register that shows up in the Register Viewer in the CCS debugger can be manually altered by typing in values directly in their fields. You just need to remember to hit return after you enter the value for it to take effect.

B) No. That is just unnatural. Registers should only be configured by the program, not the debugger.

9.3 Digital Input Programming

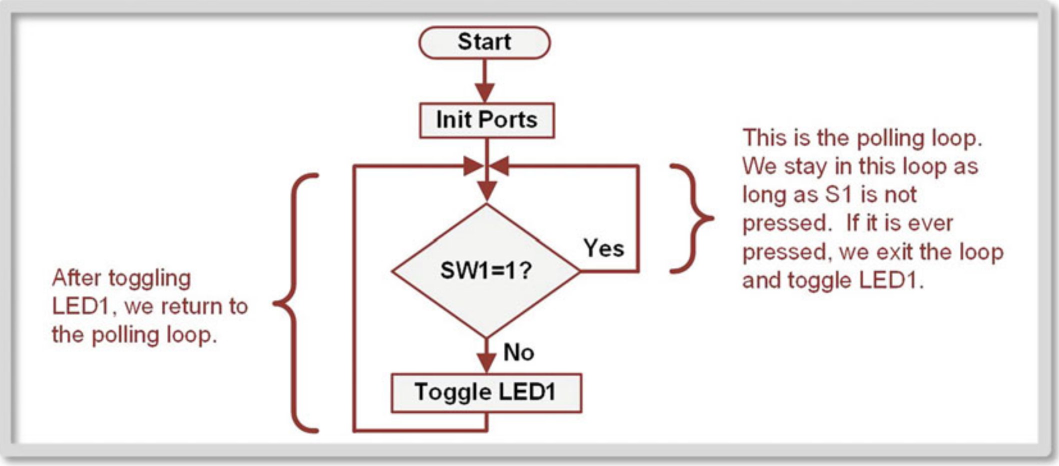
Let's look at writing a program that will configure a pin as a digital input on the MSP430FR2355. There are two push-button switches provided on the LaunchPad™ board labeled S1 and S2. S1 is connected to P4.1 and S2 is connected to P2.3. Both push-button switches are SPST with their inputs connected to ground. This provides a logic LOW to the MCU when not pressed. This means if we want to use these switches, we need to include pull-up resistors on the MCU in order to provide a known state when not pressed. Figure 9.4 shows the SPST push-button switch circuitry for S1 on the LaunchPad™ board indicating how the pull-up resistor should be configured.



NOTE: S1 on the LaunchPad™ board is a SPST switch with its input connected to ground. When pressed, the MCU input sees a logic LOW. In order to provide a logic HIGH when the switch is not pressed, we must insert a pull-up resistor.

Fig. 9.4
Push-button switch circuitry on the LaunchPad™ board

Let's specifically look at reading a logic level from S1. One of the simplest ways to read from an input pin is through a process called *polling*. Polling consists of creating a program loop that will continually check the value of the input and only exit the loop if the value changes. Applying the concept of polling to reading S1, we can create a program loop that will continually check the switch to see if it has been pressed. Based on the circuit diagrams in Fig. 9.4, we should stay in the loop as long as the value of S1 = 1, meaning that the switch has not been pressed. If S1 ever equals 0, then that indicates that the switch has been pressed and we can exit the loop and perform some actions using other instructions. After the action is taken, the program reenters the polling loop to wait for the next press. Let's create a program that will poll S1 and if it is pressed, toggle LED1. The flow chart for this program is shown in Fig. 9.5.

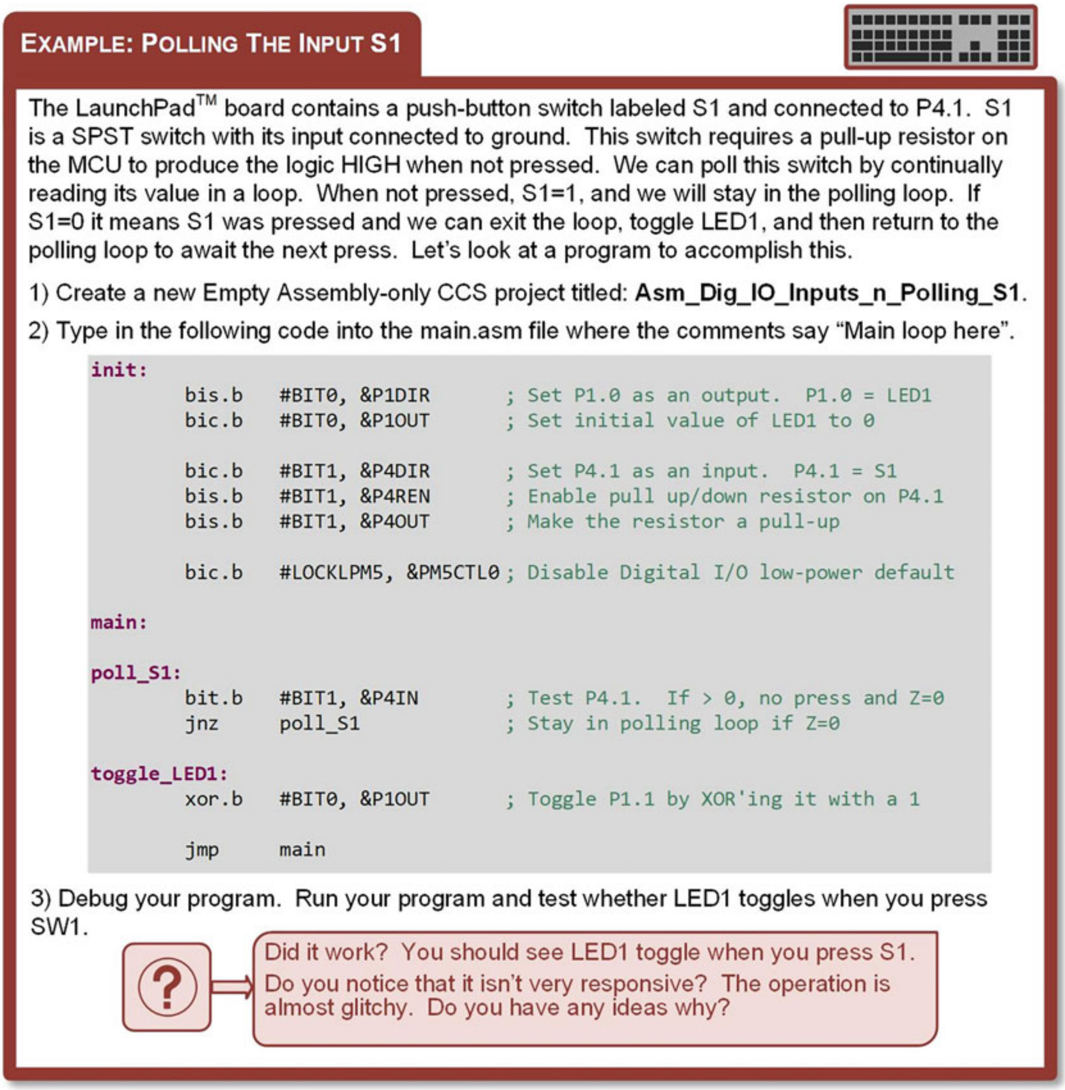


After toggling LED1, we return to the polling loop.

This is the polling loop. We stay in this loop as long as S1 is not pressed. If it is ever pressed, we exit the loop and toggle LED1.

Fig. 9.5
Flow chart for polling SW1

Follow Example 9.3 to create a program to implement the S1 polling logic.



Example: Polling The Input S1

The LaunchPad™ board contains a push-button switch labeled S1 and connected to P4.1. S1 is a SPST switch with its input connected to ground. This switch requires a pull-up resistor on the MCU to produce the logic HIGH when not pressed. We can poll this switch by continually reading its value in a loop. When not pressed, S1=1, and we will stay in the polling loop. If S1=0 it means S1 was pressed and we can exit the loop, toggle LED1, and then return to the polling loop to await the next press. Let's look at a program to accomplish this.

1) Create a new Empty Assembly-only CCS project titled: **Asm_Dig_IO_Inputs_n_Polling_S1**.

2) Type in the following code into the main.asm file where the comments say "Main loop here".

```
init:
        bis.b   #BIT0, &P1DIR       ; Set P1.0 as an output.  P1.0 = LED1
        bic.b   #BIT0, &P1OUT       ; Set initial value of LED1 to 0

        bic.b   #BIT1, &P4DIR       ; Set P4.1 as an input.  P4.1 = S1
        bis.b   #BIT1, &P4REN       ; Enable pull up/down resistor on P4.1
        bis.b   #BIT1, &P4OUT       ; Make the resistor a pull-up

        bic.b   #LOCKLPM5, &PM5CTL0 ; Disable Digital I/O low-power default

main:

poll_S1:
        bit.b   #BIT1, &P4IN        ; Test P4.1.  If > 0, no press and Z=0
        jnz     poll_S1             ; Stay in polling loop if Z=0

toggle_LED1:
        xor.b   #BIT0, &P1OUT       ; Toggle P1.1 by XOR'ing it with a 1

        jmp     main
```

3) Debug your program. Run your program and test whether LED1 toggles when you press SW1.

Did it work? You should see LED1 toggle when you press S1. Do you notice that it isn't very responsive? The operation is almost glitchy. Do you have any ideas why?

Example 9.3
Polling the input S1

When running the program from Example 9.3 you probably noticed that pressing S1 did not always result in LED1 toggling. You may have also noticed that when you held down S1, LED1 appeared dimmer than when it was on and S1 was not pressed. These issues have to do with the timing of the polling loop. The polling loop is testing the S1 bit many millions of times per second. This means no matter how fast you press and release S1, the program will exit the polling loop and toggle LED1 thousands of times. The reason that LED1 is not always toggled when you press and release S1 is because you can never tell what value LED1 is at when you release the button. The reason that LED1 appears dimmer when you hold down S1 is because it is being continually turning LED1 on and off millions of times each second while the program checks S1, exits the polling loop, and performs the XOR toggle operation. One common technique to address the issues encountered when applying polling to the relatively slow human interaction with the MCU is to insert a delay loop after the event of interest occurs. In our case the event of interest is the human finger pressing S1. The delay will give time for the human to remove their finger from S1 before reentering the polling loop. The amount of delay depends on how responsive you want the button to be. Consider the new logic for polling S1 in Fig. 9.6 that inserts some delay after the LED1 toggling action.

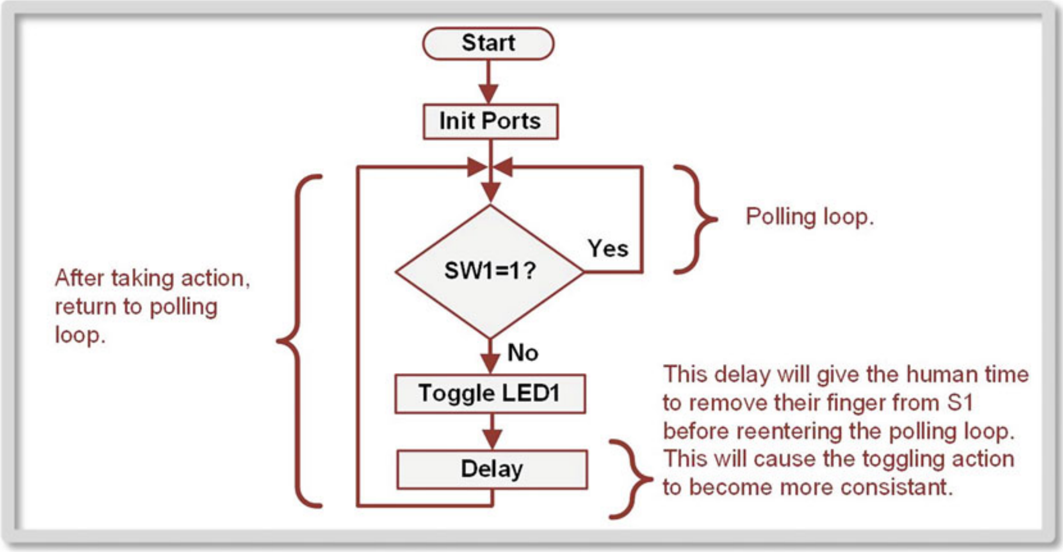

Start
Init Ports
SW1=1?
Yes
No
Toggle LED1
Delay
Polling loop.
After taking action, return to polling loop.
This delay will give the human time to remove their finger from S1 before reentering the polling loop. This will cause the toggling action to become more consistant.

Fig. 9.6
Flow chart for polling SW1 with delay

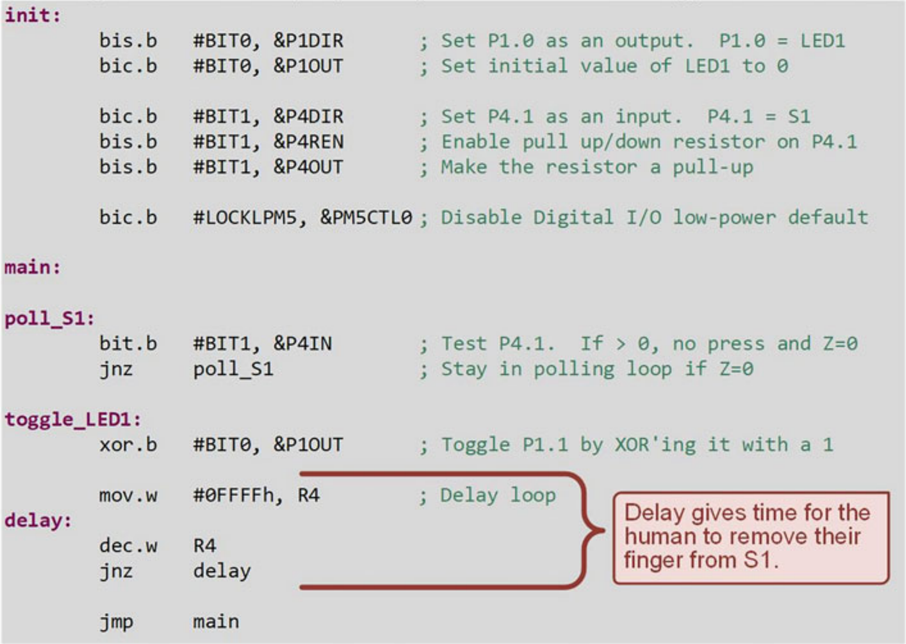
Now follow Example 9.4 to see the impact of adding delay after the polling loop.

Example: Polling The Input S1 with Delay

The MCU can poll an input much faster than a human can interact with it. This leads to glitchy behavior when using push buttons because the polling loop responds millions of times before the human can even remove their finger from the button. One solution to this is to insert delay after the polling loop to allow time for the human to remove their finger. Consider the following polling program with delay inserted.

1) Open the project from the last example titled: **Asm_Dig_IO_Inputs_n_Polling_S1**. If you didn't do the last example, create a new Empty Assembly-only CCS project of this name.

2) Type in the following code into the main.asm file where the comments say "Main loop here". If you already did the last example, you will just need to add the delay portion.



```
init:
        bis.b   #BIT0, &P1DIR       ; Set P1.0 as an output.  P1.0 = LED1
        bic.b   #BIT0, &P1OUT       ; Set initial value of LED1 to 0

        bic.b   #BIT1, &P4DIR       ; Set P4.1 as an input.  P4.1 = S1
        bis.b   #BIT1, &P4REN       ; Enable pull up/down resistor on P4.1
        bis.b   #BIT1, &P4OUT       ; Make the resistor a pull-up

        bic.b   #LOCKLPM5, &PM5CTL0 ; Disable Digital I/O low-power default

main:

poll_S1:
        bit.b   #BIT1, &P4IN        ; Test P4.1.  If > 0, no press and Z=0
        jnz     poll_S1             ; Stay in polling loop if Z=0

toggle_LED1:
        xor.b   #BIT0, &P1OUT       ; Toggle P1.1 by XOR'ing it with a 1

        mov.w   #0FFFFh, R4         ; Delay loop
delay:
        dec.w   R4
        jnz     delay

        jmp     main
```

Delay gives time for the human to remove their finger from S1.

3) Debug your program. Run your program and test whether LED1 toggles when you press SW1 more reliably than before.

Did it work? You should see LED1 toggle when you press S1, but this time it should be less glitchy.
Hold down S1 and you'll notice that LED1 continually toggles. Can you explain why?

Example 9.4
Polling the input S1 with delay

Concept Check

CC9.3 What is a downside of making the delay after a polling loop too long?

A) There isn't any downside. The more the better, always.

B) The button will appear unresponsive because it will be in the delay loop much of the time and not polling the input.

C) You will use up all your data memory with the delay loop.

D) The program might not fit in program memory with the additional delay instructions.

Summary

- Digital I/O gives an MCU the ability to read and write logic levels to the pins of a device. This system is highly versatile because it can be used for nearly any application.
- MCUs share functionality on their pins to save space on the package. Digital I/O is typically the first option on each pin.
- The MSP430FR2355TPT has six digital I/O ports named P1 (8-bit), P2 (8-bit), P3 (8-bit), P4 (8-bit), P5 (5-bit), and P6 (7-bit). All 44 I/O are brought out to pins on the MCU package.
- 16-bit access to the ports can be accomplished using the labels PA (P1:P2), PB (P3:P4), and PC (P5:P6).
- All 44 I/O of MSP430FR2355 MCU are brought out to pins, LEDs, connectors, or sensors on the LaunchPad™ board.
- There are six registers for each port in the MSP430 that configure the port operation. They are PxDIR, PxIN, PxOUT, PxREN, PxSEL1, and PxSEL0.
- PxDIR configures each bit within a port to either an input or output. The logic for PxDIR is: 0 = input (default); 1 = output.
- PxIN contains the logic values for each pin of a port configured as an input.
- PxOUT can be written in order to set the logic values for any signals configured as outputs. PxOUT has a secondary role when a port bit is configured as an input and uses a pull-up/pull-down resistor. In this case, PxOUT controls the polarity of the resistor.
- PxREN is used to control whether an optional pull-up/pull-down resistor is added to pins configured as inputs. The logic is: 0 = no resistor (default); 1 = resistor enabled. When the resistor is enabled, the PxOUT register dictates whether it is a pull-up or pull-down using the logic: 0 = pull-down; 1 = pull-up.
- PxSEL1 and PxSEL0 select the function to be used on each pin. The default value for PxSEL1:PxSEL0 = 00, which selects the digital I/O function. The secondary and tertiary functions for each pin are listed in the MSP430FR2355 device-specific data sheet.
- Upon power-up or reset, the digital I/O system puts all pins of the digital I/O system into a high-impedance input mode with Schmitt triggers in order to reduce power consumption and avoid errant current flow. To take the digital I/O system out of this low-power mode, the programmer needs to clear the LOCKLPM5 bit in the PM5CTL0 register.
- The procedure to initialize the ports is to first configure the PxDIR, PxREN, PxOUT (if using a pull-up/pull-down resistor), and PxSEL1:PxSEL0 registers. Next, the LOCKLPM5 bit in the PM5CTL0 register is cleared to take the digital I/O system out of low-power mode. After this, the PxIN or PxOUT registers are ready for use by the main program.
- Programmers can take advantage of the default settings of some configuration registers upon startup; however, explicitly configuring some bits makes your code more readable. It is common to explicitly configure PxDIR so it is obvious how the pin is being used. It is also common to accept the default values for PxSEL1:PxSEL0 = 00, which selects digital I/O as the pin function.
- Each new main.asm file created by CCS includes a msp430.h header file. This header

file points to a device-specific header file, in our case the msp430fr2355.h. The device-specific header file contains literal names for each configuration register in the memory map plus some common bit masks. Using the literal definitions from the header files makes a program easier to understand and also potentially portable to another MSP430 MCU.

- The LaunchPad™ board contains two user LEDs labeled LED1 and LED2. These are connected to P1.0 and P6.6 of the MCU, respectively. These LEDs can be driven directly as digital outputs to turn them on and off (0 = OFF, 1 = ON).
- The LaunchPad™ board contains two push-button switches labeled S1 and S2. These are connected to P4.1 and P2.3 of the MCU, respectively. These switches are SPST with their inputs connected to ground. This provides a logic LOW when pressed. A pull-up resistor is required to be enabled on the MCU pin in order to provide a logic HIGH when the switch is not pressed.
- Polling is the process of continually checking the value of an input. This is accomplished by creating a program loop that will continually check the value of the pin. The program will stay in the polling loop as long as no event has occurred (i.e., the switch has not been pressed). When an event occurs (i.e., the switch has been pressed), the program exits the polling loop and performs some task. It then reenters the polling loop to wait for the next event.
- When polling a human interaction such as a button press, the mismatch in speed between the MCU bit checking and the human motion can create glitchy behavior. This is because the MCU can poll the input millions of times per second. Even the fastest human button press can be observed by the polling loop as thousands of presses.
- To avoid the glitchy behavior, a delay loop can be inserted after the program exits the polling loop to give time for the human to remove their finger from the button.

Exercise Problems

Section 9.1: The Digital I/O System

9.1.1 What does the PxDIR register configure?

9.1.2 What is the logic for the PxDIR register?

9.1.3 What is the function of the PxIN register?

9.1.4 What is the primary function of the PxOUT register?

9.1.5 What is the secondary function of the PxOUT register?

9.1.6 What does the PxREN register configure?

9.1.7 What is the logic for the PxREN register?

9.1.8 What do the PxSEL1:PxSEL0 registers configure?

9.1.9 What function will be selected for an MCU pin if PxSEL1:PxSEL0 = 00?

9.1.10 What is the default value for PxSEL1:PxSEL0 after power-on or reset?

9.1.11 What does the msp430.h header file provide for the programmer?

9.1.12 Why is it a good idea to use the literals from the msp430.h header file?

Section 9.1: Digital Output Programming

9.2.1 If you rely on the function select registers' default value choosing digital I/O as a pin function (PxSEL1:PxSEL0 = 00), what are the only two configuration steps that are needed to initialize a pin to be an output?

9.2.2 Give the assembly program code to configure P1.4 as an output. Use the literal definitions from the msp430.h header file for the register names and bit masks.

9.2.3 Give the assembly program code to configure P2.0 as an output. Use the literal definitions from the msp430.h header file for the register names and bit masks.

9.2.4 Give the assembly program code to configure P3.7 as an output. Use the literal definitions from the msp430.h header file for the register names and bit masks.

9.2.5 Give the assembly program code to configure P4.5 as an output. Use the literal definitions from the msp430.h header file for the register names and bit masks.

9.2.6 Give the assembly program code to drive a logic LOW to P1.4. Use the literal definitions from the msp430.h header file for the register names and bit masks.

9.2.7 Give the assembly program code to drive a logic LOW to P2.0. Use the literal definitions from the msp430.h header file for the register names and bit masks.

9.2.8 Give the assembly program code to drive a logic HIGH to P3.7. Use the literal definitions from the msp430.h header file for the register names and bit masks.

9.2.9 Give the assembly program code to drive a logic HIGH to P4.5. Use the literal definitions from the msp430.h header file for the register names and bit masks.

Section 9.3: Digital Input Programming

9.3.1 If an SPST switch is connected to an MCU pin and has its input tied to GND, does the MCU need to enable a pull-up or pull-down resistor? Why or why not?

9.3.2 If an SPST switch is connected to an MCU pin and has its input tied to the power supply, does the MCU need to enable a pull-up or pull-down resistor? Why or why not?

9.3.3 What is the concept of *polling*?

9.3.4 Why does delay help with polling when reading an input with human interaction?

9.3.5 What is the downside of putting too large of a delay after the polling loop is exited?

Chapter 10: The Stack and Subroutines

This chapter introduces the concept of a stack as a means to dynamically store information in data memory [1]. It then looks into the details of how the stack is implemented on the MSP430. The subroutine is then introduced with details on how the stack is used to store the return address of the routine upon subroutine exit.

Learning Outcomes—After completing this chapter you will be able to:

10.1 Use the stack on the MSP430 to dynamically access data memory.
10.2 Use subroutines on the MSP430.

10.1 The Stack

The *stack* is a system that allows us to dynamically allocate data memory. The term *dynamically* means that we can access memory without initializing it or reserving it using assembler directives such as .short and .space. A stack is a *last-in, first-out* (LIFO) storage structure. One common analogy for how a stack operates is to imagine plates held in a spring-loaded dispenser found in a cafeteria. Let's assume that we are going to load clean plates into the dispenser one by one. We place the first plate into the dispenser. Next, we place the second plate on top of the first plate. Next, we place the third plate on top of the second plate, and so on. At any given time, the plate on top is the last plate that was placed into the dispenser. When you are ready to grab a plate, you take the top plate. The top plate was the last one placed into the dispenser, but the first one taken out (i.e., a LIFO). The next person in line then grabs the next available plate, which was the second to last plate placed in the dispenser, and so on. Figure 10.1 shows a graphical depiction of the stack LIFO concept.

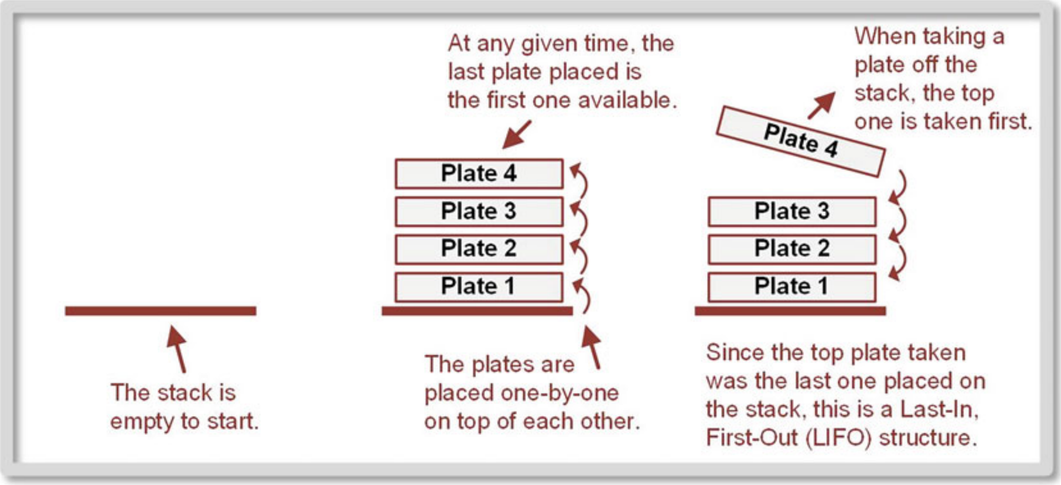

At any given time, the last plate placed is the first one available.
When taking a plate off the stack, the top one is taken first.
Plate 4
Plate 4
Plate 3
Plate 2
Plate 1
Plate 3
Plate 2
Plate 1
The stack is empty to start.
The plates are placed one-by-one on top of each other.
Since the top plate taken was the last one placed on the stack, this is a Last-In, First-Out (LIFO) structure.

Fig. 10.1
The last-in, first-out (LIFO) stack concept

B. J. LaMeres, *Embedded Systems Design using the MSP430FR2355 LaunchPad™*,
https://doi.org/10.1007/978-3-031-20888-1_10

There are a few traits of the plate dispenser analogy that highlight the operation of a stack in a computer. First, the order of the plates is inherently kept by the stack. As long as you count how many plates you have placed onto the stack and how many have been taken out, you know exactly which plate is on top. Second, the stack of plates can be replenished at any time. We don't need to wait until we run out of plates to put more plates on it. Third, the number of plates that we can put on the stack is deterministic. The dispenser will only hold so many plates before it is full.

Now let's compare these traits to the operation of a stack in a computer in which the stack is implemented as a way to access data memory. First, we need to keep track of the address of where we are putting information onto the stack. This is done using a dedicated register (SP) that is incremented and decremented to track where we are in the stack structure. Second, we can store information on the stack structure in memory at any time since we know the location of where the last information resides, so we don't need to worry about overriding existing information. We simply move to the next location in memory above the last data stored and place the new information. And finally, since the amount of data memory is finite in a computer, we also know that we cannot put an infinite amount of information onto the stack in memory. If we put too much information on the stack, it can override other information in memory and even move into addresses that aren't in data memory. This is called *stack overflow*.

Now let's examine the details of how a stack is implemented in data memory on the MSP430. When information is stored on the stack, it is called a *push* operation. When information is retrieved off of the stack, it is called a *pop* operation. The stack starts at the end of the data memory range and works its way backward through the memory range as information is pushed. In the MSP430FR2355, the primary 4kb data memory block is located at addresses 2000h → 2FFFh. This means the first 16-bit word of information pushed will be stored at address 2FFEh. The stack resides at the end of data memory to allow the maximum potential size of the stack and also avoids overriding reserved locations in memory that are placed at the beginning address of data memory (i.e., 2000h); however, if too many pushes occur, it is possible to overwrite variables that were allocated at the beginning of data memory (i.e., stack overflow).

A stack pointer (SP) register is used to provide the address when accessing the stack. On the MSP430, the stack pointer is a CPU register that can be referred to in a program as SP or R1. The SP must be initialized by the user and the SP is always aligned to even addresses. Setting up the starting address of the stack is one of the operations that is automatically done for us when we create a new project in CCS. This is accomplished using a move instruction and a global constant called *__STACK_END*. Upon startup, the SP is initialized to address 3000h, which in the MSP430FR2355 is the address immediately after the 4k data memory range. The first push is then stored to 2FFEh. The second push is to address 2FFCh.

The SP operates using a pre-decrement, post-increment scheme. This means that when a push is performed, the SP is decremented by 2 and then the 16-bit word is stored to the address pointed to by SP. When a pop is performed, the 16-bit word is retrieved from the address that SP is pointing to, and then SP is incremented by 2. Figure 10.2 shows a graphical depiction of the stack memory space and how pushes and pops work.

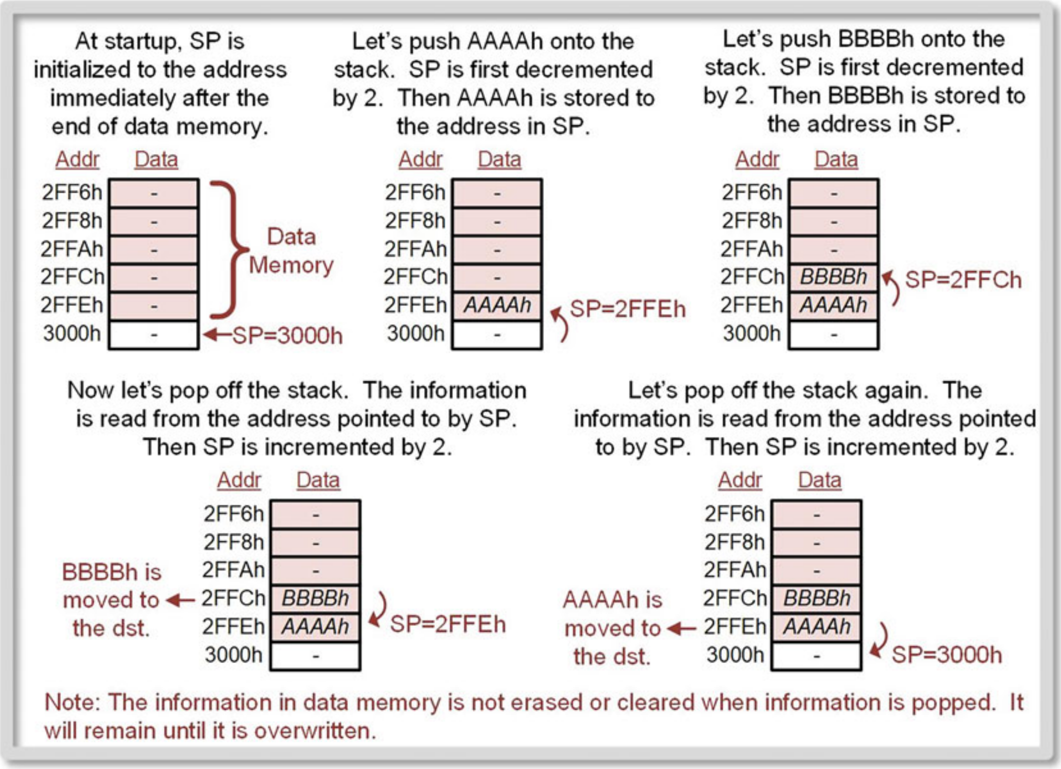

At startup, SP is initialized to the address immediately after the end of data memory.
Addr
Data
2FF6h
2FF8h
2FFAh
2FFCh
2FFEh
3000h
-
Data Memory
SP=3000h
Let's push AAAAh onto the stack. SP is first decremented by 2. Then AAAAh is stored to the address in SP.
AAAAh
SP=2FFEh
Let's push BBBBh onto the stack. SP is first decremented by 2. Then BBBBh is stored to the address in SP.
BBBBh
SP=2FFCh
Now let's pop off the stack. The information is read from the address pointed to by SP. Then SP is incremented by 2.
BBBBh is moved to the dst.
SP=2FFEh
Let's pop off the stack again. The information is read from the address pointed to by SP. Then SP is incremented by 2.
AAAAh is moved to the dst.
SP=3000h
Note: The information in data memory is not erased or cleared when information is popped. It will remain until it is overwritten.

Fig. 10.2
Stack implementation in the MSP430 memory system

Follow Example 10.1 to experiment with the operation of the stack on the MSP430FR2355.

Example: Using the Stack

The stack begins at the end of data memory and works backwards through the addresses as information is pushed. The SP tracks where information is being pushed and popped. Let's look at an example program that will push some values onto the stack and then pop them.

1) Create a new Empty Assembly-only CCS project titled: **Asm_Stack**.

2) Type in the following code into the main.asm file where the comments say "Main loop here".

```
main:
        mov.w   #0AAAAh, R4
        mov.w   #0BBBBh, R5

        push    R4
        push    R5

        pop     R6
        pop     R7

        mov.w   #0CCCCh, R4
        mov.w   #0DDDDh, R5

        push    R4
        push    R5

        pop     R6
        pop     R7

        jmp     main
```

First we'll put values into R4 and R5.

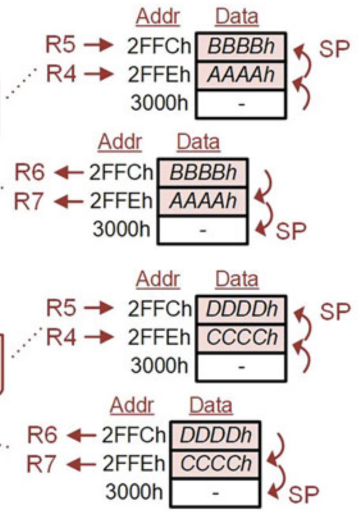
Then we'll push them onto the stack.

Then we'll pop them into R6 and R7.

Next we'll put new values into R4 and R5.

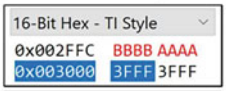
Then we'll push them onto the stack.

Then we'll pop them into R6 and R7.



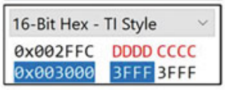
3) Debug your program. Run your program to the breakpoint. Open the Register Viewer so that you can see SP and R4 → R7. Open the Memory Browser and go to 0x3000. Then scroll up so you can see the values 2FFEh and 2FFCh.

4) Step your program. As you step, look at the values of SP and the values in data memory before 0x3000. In the Memory Browser you should see the following as the values are pushed.



```
16-Bit Hex - TI Style
0x002FFC  BBBB AAAA
0x003000  3FFF 3FFF
```

First two pushes.



```
16-Bit Hex - TI Style
0x002FFC  DDDD CCCC
0x003000  3FFF 3FFF
```

Second two pushes.

Did it work? Did you see SP decrement from 3000h → 2FFEh → 2FFECh as values were pushed?

Did you see SP increment from 2FFCh → 2FFEh → 3000h as values were popped?

Did you see the values in data memory at addresses 2FFCh and 2FFEh change as values were pushed?

Did you see the values of R6 and R7 change as values were popped?

Example 10.1
Using the stack on the MSP430FR2355

Concept Check

CC10.1 In theory, if no variables were initialized or reserved at the beginning of data memory, how many bytes can be pushed onto the stack before stack overflow occurs on the MSP430FR2355?

A) 4k. We can push values into the entire primary data memory block.

B) 64k. We can push as much as we want, regardless of whether it is data memory or not. It is our program.

C) 16 words. The stack is implemented as CPU registers, so we only have 16 words.

D) 32k. I prefer to use the program memory for the stack. That way I have more storage than when using data memory. I do realize that I can't write to program memory, but that doesn't stop me from dreaming.

10.2 Subroutines

A *subroutine* is a piece of code that will be used repeatedly in a program. A subroutine typically accomplishes a very specific task. While the code to accomplish this task could certainly be placed in the main program anytime it was needed, it would be inefficient to copy and paste the same code segment many times throughout the main program. In addition, this would lead to increased program size and less readability by subsequent programmers. In the subroutine approach, the code to accomplish the task is implemented only once outside of the main program loop. Whenever it is needed, it can be executed by jumping to it. Other common names for subroutines used in programming are *procedures, functions, routines, methods,* or *subprograms.* While all of these other names may have slightly different features, they are all essentially the same as a subroutine in that they are designed to accomplish a specific task that is implemented outside of the main program loop and then called when needed. Once the subroutine completes, a return jump is used to move the PC back to the next location in the main program loop to continue operation.

A subroutine starts with an address label to mark its location in memory. Additional steps must be taken when jumping to a subroutine because while the starting address of the subroutine is always the same, the return address in the main program will vary depending on where in the main program it is called. In order to track the dynamic return address when calling a subroutine, the stack is used. The MSP430 provides an instruction named **`call`** that is used to jump to the subroutine address label. This instruction handles storing the return address on the stack prior to jumping to the subroutine starting address. An additional instruction named ret (for return) is used at the end of the subroutine that pops the return address off of the stack and places it into PC to return to the main program. The return address is calculated by the `call` instruction so that the address returned to is the next instruction in program memory after the call.

Variables can be passed to subroutines using three different approaches. The first is using CPU registers. The second is using the stack. The third is using dedicated variables in data memory.

Let's take a look at a programming example using a subroutine on the MSP430. Follow Example 10.2 to observe the computer operation during subroutine usage.

EXAMPLE: USING SUBROUTINES

Let's create a subroutine called "complement_it" that will perform a bitwise inversion of R4. In the main program we will put different values into R4 and then observe the PC, SP, and R4 as the subroutine is called from different locations in the main program loop.

1) Create a new Empty Assembly-only CCS project titled: **Asm_Subroutines**.

2) Type in the following code into the main.asm file where the comments say "Main loop here".

```
Addr    main:
800Ah ●         mov.w   #0AAAAh, R4
800Eh           call    #complement_it

8012h           mov.w   #0BBBBh, R4
8016h           call    #complement_it

801Ah           mov.w   #0CCCCh, R4
801Eh           call    #complement_it

8022h           jmp     main

        ;-------------------------------
        ; Subroutines
        ;-------------------------------
        complement_it:
8024h           inv.w   R4
8026h           ret
```

Addresses in Program Memory

When this call is made, 8012h is pushed onto the stack as the return address.

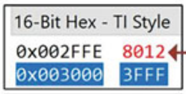
When this call is made, 801Ah is pushed onto the stack as the return address.

When this call is made, 8022h is pushed onto the stack as the return address.

Upon return, the dynamic return address is popped from the stack and put into PC.

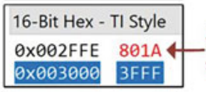
3) Debug your program. Run your program to the breakpoint. Open the Register Viewer so that you can see PC, SP and R4. Open the Memory Browser and go to 0x3000. Then scroll up so you can see the first location on the stack (address 2FFEh).

4) Step your program using **Step Into**. As you step, look at the values of PC, SP and the values in data memory for the stack. In the memory viewer you should see the following as the value are pushed.



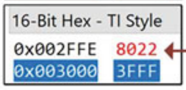
```
16-Bit Hex - TI Style
0x002FFE  8012
0x003000  3FFF
```

Stack after first call.



```
16-Bit Hex - TI Style
0x002FFE  801A
0x003000  3FFF
```

Stack after second call.



```
16-Bit Hex - TI Style
0x002FFE  8022
0x003000  3FFF
```

Stack after third call.

5) Now step your program using **Step Over**. This time when you step you'll see the program still executes the subroutine, but it doesn't move into the subroutine code. This is the first time we have been able to use *step over*.

Did it work? Did you see PC jump to the subroutine starting address when it was called and then return to the main program when it returned?

Did you see the stack store the return address when the subroutine was called?

Did you see the difference between Step Over and Step Into? Pretty neat, huh?

Example 10.2
Using subroutines on the MSP430FR2355

Concept Check

CC10.2 If during assembly the assembler goes through and replaces all address labels with their absolute values, wouldn't it be more efficient for the assembler to just calculate the absolute return address for each subroutine call instead of using the stack?

A) Yes. That way we could use unconditional jumps instead of messing with the stack and `ret`.

B) No. Then the subroutine would need to keep track of all of the locations where it was called from in the main program and look up their absolute return address for the return. This additional lookup code would be much more complicated than using the stack.

Summary

- A stack is a last-in, first-out (LIFO) storage structure. This is used for dynamic memory allocation.
- Putting information on the stack is called a *push*. A push always puts information on the top of the stack. The MSP430 contains an instruction called `push`.
- Retrieving information from the stack is called a *pop*. A pop always gets information from the top of the stack. The MSP430 contains an instruction called `pop`.
- The MSP430 implements a stack at the bottom of its data memory range.
- The SP register in the CPU tracks the address of where information is pushed and popped from the stack. The SP is always aligned to even addresses.
- On the MSP430FR2355, SP is initialized by the user to 3000h, which is the address immediately after the last location in data memory.
- The SP works backwards through the data memory range in order to give the maximum storage size and avoid overwriting explicitly defined variables that occur at the beginning of data memory.
- If the stack grows too large due to a large number of pushes, it can overwrite variables at the beginning of data memory and even move into memory outside of the data memory. This is called *stack overflow*.
- The SP operates in a pre-decrement, post-increment scheme. When a push is performed, the address that SP is pointing to is decremented by 2 and then information is stored to the new SP address. When a pop is performed, information is read from the address pointed to by SP and then SP is incremented by 2.
- A subroutine is a sequence of instructions that will be used many times during a program, so instead of inserting it multiple times in the main program loop, it is inserted outside of the main loop once and jumped to multiple times.
- A subroutine uses an address label to indicate its starting location in program memory. This address label is used when the program is called and is the same regardless of where in the main program it is called.
- Since a subroutine can be called from any location in the main program, the return address from the subroutine is dynamic. The stack is used to store the dynamic return address when a subroutine is called.
- The `call` instruction is used to call a subroutine. It will calculate the return address, which is the location of the next instruction in the main program, and push it onto the stack. It will then move PC to the starting address of the subroutine.
- The `ret` instruction is used to return from a subroutine. It will pop the return address from the stack and place it in PC so that the program will return to the next instruction in the main program after the call.

Exercise Problems

Section 10.1: The Stack

10.1.1 Where is information always stored to on the stack (i.e., top or bottom)?

10.1.2 Where is information always retrieved from on the stack (i.e., top or bottom)?

10.1.3 What is the instruction called that stores information to the stack?

10.1.4 What is the instruction called that retrieves information from the stack?

10.1.5 What does LIFO stand for?

10.1.6 Why does the stack start at the bottom of data memory?

10.1.7 What address should the user initialize the stack to upon startup on the MSP430FR2355?

10.1.8 Is SP always aligned to even or odd addresses?

10.1.9 When information is pushed to the stack, is the SP decremented before or after the store?

10.1.10 When information is popped from the stack, is the SP incremented before or after the retrieve?

10.1.11 What is stack overflow?

10.1.12 When a new CCS project is created, it automatically initializes SP. What is the name of the global label that it uses to load into SP?

Section 10.2: Subroutines

10.2.1 What is a subroutine?

10.2.2 Why use a subroutine instead of just inserting the code for the task in the main program loop any time it is needed?

10.2.3 Is the starting address of a subroutine always constant? Why or why not?

10.2.4 Is the return address for a subroutine always constant? Why or why not?

10.2.5 What does the `call` instruction do?

10.2.6 What does the `ret` instruction do?

10.2.7 What are the ways that variables can be passed to subroutines?

10.2.8 If nothing is on the stack on the MSP430FR2355 and a subroutine is called, at what location is the return address placed?

Chapter 11: Introduction to Interrupts

This chapter introduces the concept of an interrupt (IRQ) as a way to efficiently deal with asynchronous external events that the CPU must handle [1–3]. The details of the MSP430 interrupt system are then presented and an example of using the port interrupts on the LaunchPad^{TM} board is examined.

Learning Outcomes—After completing this chapter, you will be able to:

11.1 Describe MCU operation during an interrupt.
11.2 Design a program to use the port interrupts on the MSP430FR2355.

11.1 The Concept of an Interrupt

In Chap. 9, we looked at the concept of polling to read from an external pin on the MCU. The idea of polling highlighted a variety of inefficiencies that necessitate a different approach to interfacing with systems outside of the CPU. The first issue is that external systems are asynchronous to the CPU clock. This means that the CPU never knows when these events are going to occur. Additionally, the external events are often much slower than the speed of the CPU. In the example of polling the push-button input, the MCU spent the vast majority of its time doing nothing but actively checking the input and rarely was executing instructions to react to a push-button press. The idea of polling was highly inefficient because the CPU spent so much of its time executing instructions that did nothing but actively check for an infrequent event. Additionally, when an event did occur, the polling loop would continuously see the input as asserted since it only looks at the logic level of the signal and the slow external event would keep the input asserted for many clock cycles. This required inserting a delay in the program to allow time for the slower external signal to be de-asserted. A more efficient approach to handling this type of external input is to only react one time when a transition is observed on the input pin; however, polling does not give us that ability.

11.1.1 Interrupt Flags (IFG)

An interrupt is an approach to dealing with external, asynchronous events by building hardware into the MCU that handles identifying and prioritizing events to be serviced by the CPU. The interrupt system uses the concept of a flag to notify the CPU that an external event on a peripheral has occurred and action is requested. This approach allows the CPU to continue its normal instruction execution and only act when a flag is seen. In this way, the CPU does not have to explicitly poll each external peripheral to see if it needs servicing. Instead, the CPU can execute its main program, and then when a flag is observed, it can act once it comes to a natural stopping point in the main program.

When a flag is observed, the CPU completes its current instruction and then executes a sequence of instructions (written by the programmers) that accomplishes the desired action for the peripheral. The term *interrupt* stems from the fact that the CPU takes a break from executing the main program and instead executes instructions specifically for the peripheral event. While an interrupt does momentarily halt the execution of the main program, it is highly efficient because the CPU does not have to spend execution cycles polling each external system. The code that is executed when an interrupt occurs is called an *interrupt service routine* (ISR) or *interrupt handler*. When the CPU is in the act of handling the interrupt, it is called *serving the interrupt*. When an interrupt has asserted its flag but the CPU has not had an opportunity to service it, the interrupt is said to be *pending*. Figure 11.1 shows a modified state diagram of the CPU fetch → decode → execute cycle with a new path that handles taking care of an interrupt event.

B. J. LaMeres, *Embedded Systems Design using the MSP430FR2355 LaunchPad™*,
https://doi.org/10.1007/978-3-031-20888-1_11

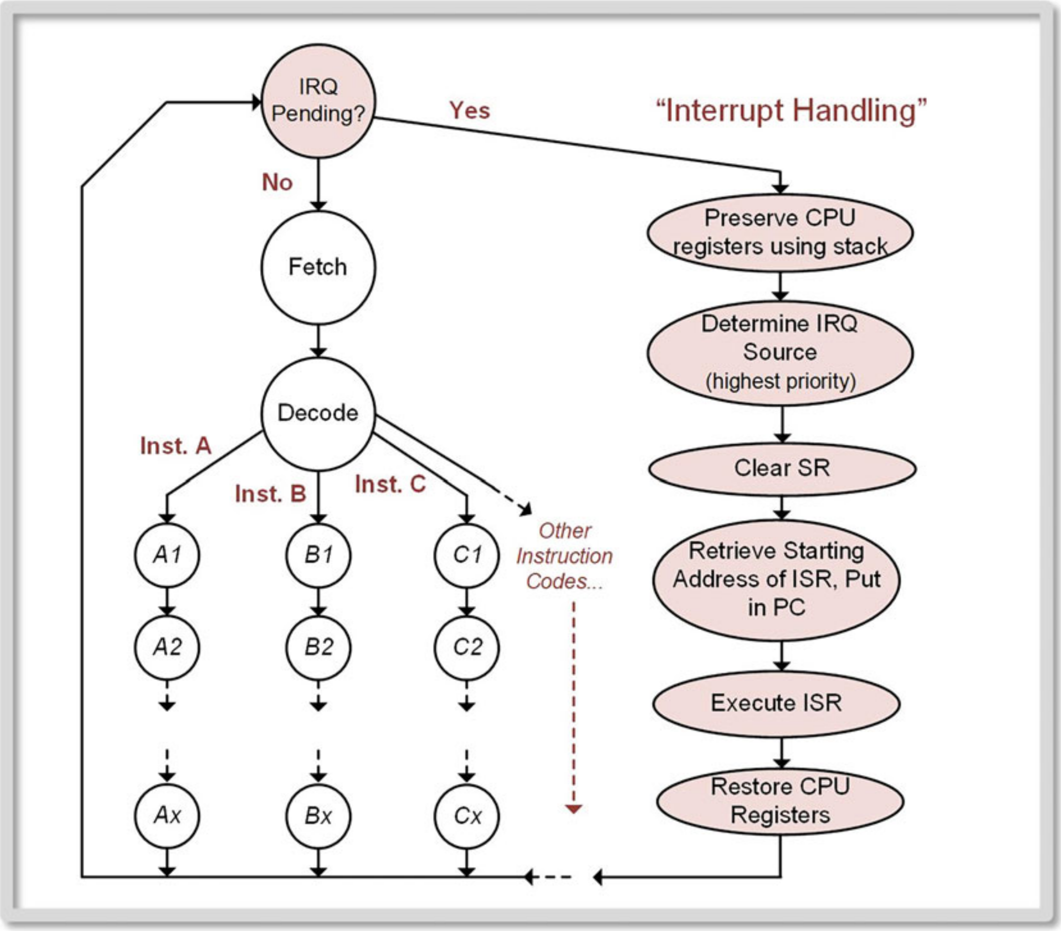

IRQ Pending?
Yes
"Interrupt Handling"
No
Fetch
Decode
Inst. A
Inst. B
Inst. C
Other Instruction Codes...
A1
A2
Ax
B1
B2
Bx
C1
C2
Cx
Preserve CPU registers using stack
Determine IRQ Source (highest priority)
Clear SR
Retrieve Starting Address of ISR, Put in PC
Execute ISR
Restore CPU Registers

Fig. 11.1
Interrupting the fetch-decode-execute cycle in the CPU

Once a CPU is architected to support interrupts, then the entire approach to MCU software development changes. Since MCUs are typically used in applications that respond to inputs from humans and/or sensors which produce events that are slow and infrequent, interrupts become the primary method to implement functionality in the software. Many times, an MCU main program loop will just consist of a main loop with no instructions or a main loop that does nothing but put the MCU into low power mode. This changes the main program loop into more of a background process, and the interrupt service routines are considered the foreground processes.

Another approach to MCU software development is to produce routines that are scheduled. MCU timers can be used to trigger interrupts on a periodic basis that cause the MCU to act at fixed intervals. This leads to the concept of a real-time operating system, which is simply a set of scheduled routines that are at fixed intervals and have determinant execution times.

11.1.2 Interrupt Priority and Enabling

Interrupts have a priority system that ranks each external peripheral from highest to lowest. This provides a means to handle multiple interrupts that occur at the same time and are simultaneously requesting service from the CPU. When the CPU is ready to service an interrupt, it always executes the highest priority peripheral. Once that routine completes, it moves to the next highest priority peripheral that is pending and so forth. Interrupts *can* interrupt other interrupts if they have a higher priority, but some restrictions apply that are described in subsequent sections.

An MCU has three categories of interrupts: (1) system resets; (2) non-maskable interrupts (NMIs); and (3) maskable interrupts. System resets are the highest priority interrupts and are always enabled. System resets are the most critical because they cause the MCU to start its operation from the beginning. This includes putting all configuration registers at their default values, initializing the program counter and entering the main program at its first instruction. An MCU typically has a variety of system resets including power-on reset (POR), a power-up reset (PUR), an external reset, and a power supply monitor violation. System resets do not have developer written ISRs that are executed; instead, the MCU performs a set of predetermined operations to prepare the CPU for first use. The only action needed by the developer for system resets is to tell the interrupt system where the starting address of the main program is (typically the first address of program memory).

Non-maskable interrupts are the second highest priority interrupts and typically handle fault conditions on the MCU. Examples of non-maskable interrupts are memory access errors and oscillator faults. Non-maskable interrupts are always enabled but are different from system resets in that they *do* execute developer written ISRs instead of a set of predetermined actions.

Maskable interrupts are the third category of interrupts and are the interrupts that handle all of the common peripherals on an MCU (i.e., ports, timers, serial communication, ADC, and DACs). Maskable interrupts have both global and local interrupt enables. The GIE bit in the status register is used as the global enable for all maskable interrupts. When it is set (GIE = 1), then maskable interrupts are allowed. Upon reset GIE = 0, meaning no maskable interrupts are enabled. The MSP430 provides two instructions to explicitly enable (`eint`) and disable (`dint`) maskable interrupts. These instructions simply set or clear the GIE bit in the SR accordingly. Each peripheral system then has a local interrupt enable (IE) that is configured in the control/status registers within the memory map. The global and local interrupt enable bits can be thought of as gating switches that allow the peripheral's flag to be observed by the CPU when configured. Figure 11.2 shows a conceptual model for the global and local interrupt enables.

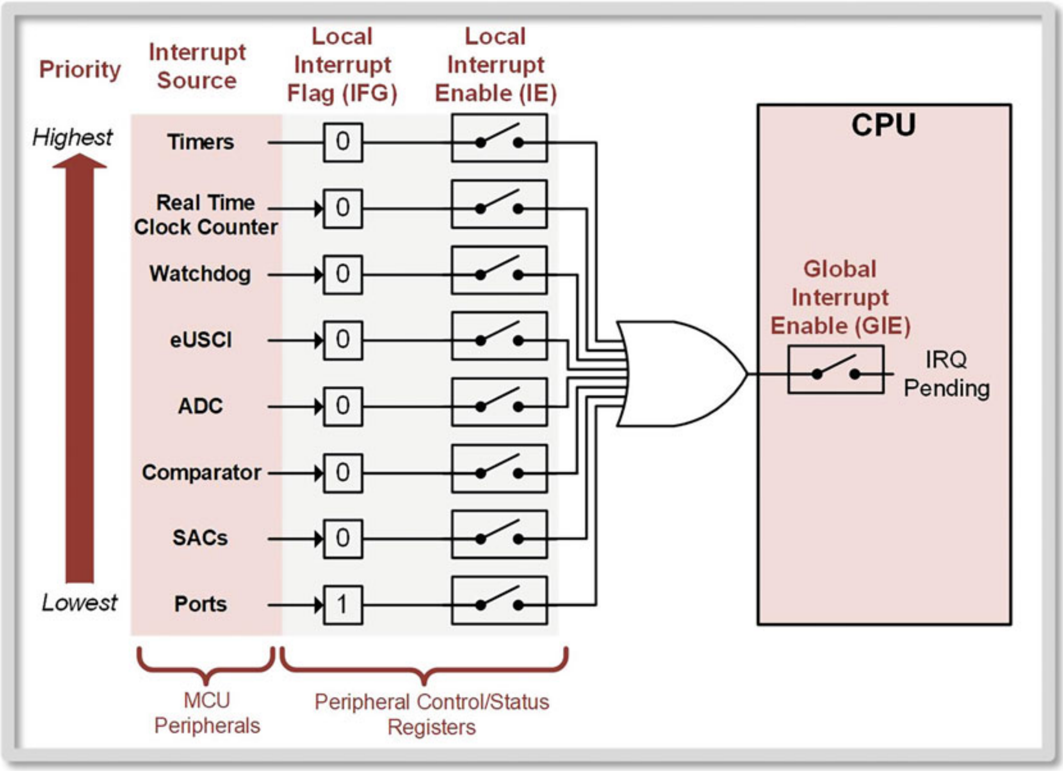


Fig. 11.2
Conceptual model for global and local interrupt enables

11.1.3 Interrupt Vectors

The beginning of an ISR is marked with an address label in the main.asm file. This address label serves as the starting address to be put into the PC when the ISR is called. The way that the CPU retrieves the starting address of the ISR to put into the program counter uses the concept of an *interrupt vector*. Each peripheral system that is capable of generating an interrupt is assigned a dedicated address location at the end of the program memory space. The address is called the peripheral's *interrupt vector address* and will hold the starting address of the ISR. Since there are numerous peripherals that each require a unique vector, the addresses consume a block of memory called the *interrupt vector table*. The starting address of the ISR is put into the interrupt vector table when the program is downloaded. The developer is responsible for putting the ISR starting address into the correct vector location using assembler directives. Figure 11.3 shows a graphical depiction of the interrupt vector table concept.

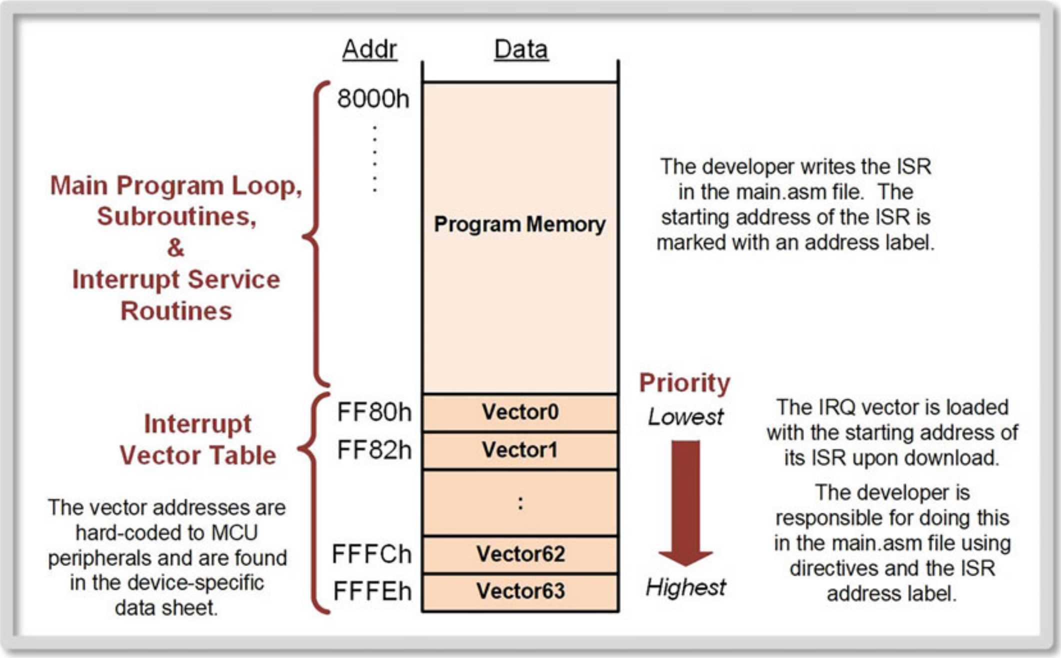
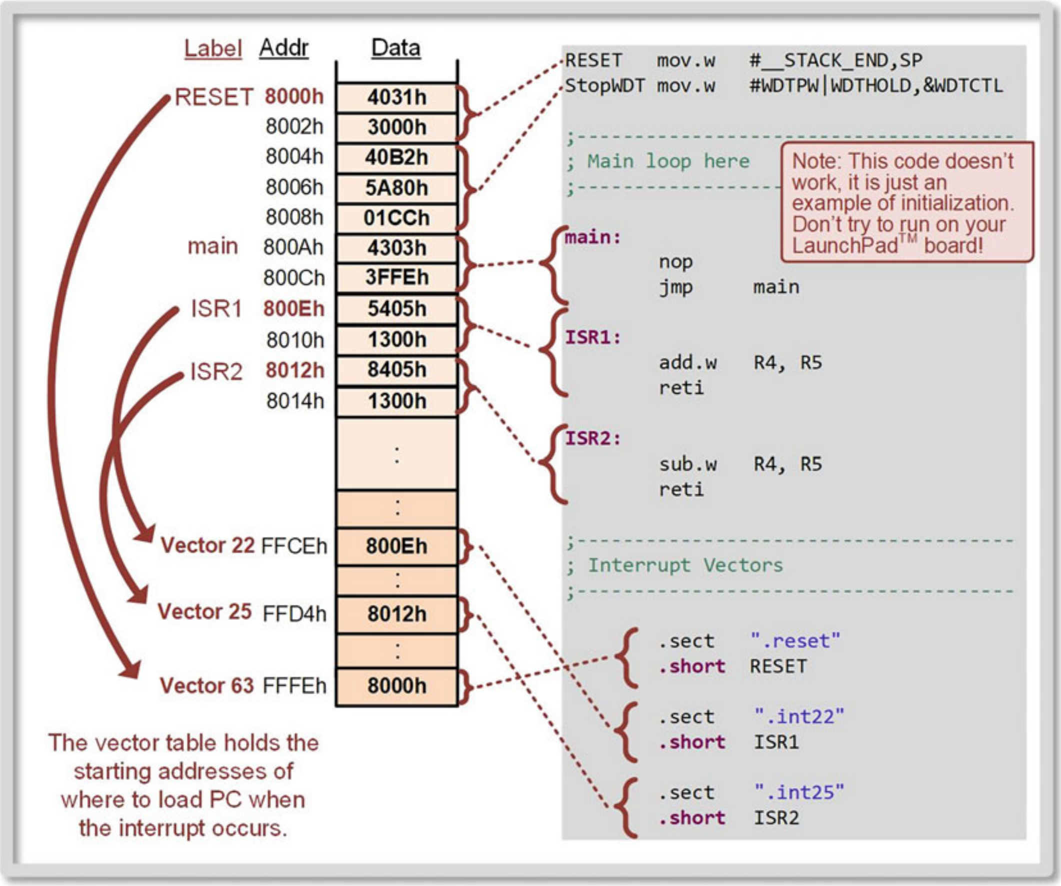

Addr
Data
8000h
Main Program Loop,
Subroutines,
&
Interrupt Service
Routines
Program Memory
The developer writes the ISR in the main.asm file. The starting address of the ISR is marked with an address label.
Priority
Lowest
Highest
FF80h
FF82h
FFFCh
FFFEh
Vector0
Vector1
:
Vector62
Vector63
Interrupt
Vector Table
The vector addresses are hard-coded to MCU peripherals and are found in the device-specific data sheet.
The IRQ vector is loaded with the starting address of its ISR upon download.
The developer is responsible for doing this in the main.asm file using directives and the ISR address label.

Fig. 11.3
Interrupt vector table concept

The full MSP430 architecture supports up to 64 separate interrupt vectors whose addresses are located within the range FF80h → FFFFh; however, very few MCUs use all of these vectors due to the varying amounts of peripherals implemented on the specific devices. That means the developer needs to determine which peripherals are assigned to which vector locations using the device-specific data sheet. Only peripherals that are going to be used need to be enabled and have their vectors initialized.

Figure 11.4 shows a graphical depiction of how the interrupt vector table is initialized using the starting addresses of ISRs and assembler directives. In this figure, three interrupt vectors are initialized: reset, vector 22, and vector 25. Vectors 22 and 25 represent maskable interrupts that have ISRs that are executed when serviced. The starting addresses of these routines (named ISR1 and ISR2) are placed into their respective vector locations using the assembler directives `.section` and `.short`. The literals that represent the absolute vector address (i.e., .reset, .int22, and .int25) come from the linker file. In this figure, vector 63 handles the highest priority interrupt in the MCU, which is reset. This vector holds the starting address of where to begin executing code when the MCU is reset or powered up. This vector holds the first address of program memory (8000h) where the main program is downloaded to.



Label	Addr	Data
RESET	8000h	4031h
	8002h	3000h
	8004h	40B2h
	8006h	5A80h
	8008h	01CCh
main	800Ah	4303h
	800Ch	3FFEh
ISR1	800Eh	5405h
	8010h	1300h
ISR2	8012h	8405h
	8014h	1300h
		:
		:
Vector 22	FFCEh	800Eh
		:
Vector 25	FFD4h	8012h
		:
Vector 63	FFFEh	8000h

```
RESET   mov.w   #__STACK_END,SP
StopWDT mov.w   #WDTPW|WDTHOLD,&WDTCTL

;-------------------------------------
; Main loop here
;-----------------
main:
        nop
        jmp     main

ISR1:
        add.w   R4, R5
        reti

ISR2:
        sub.w   R4, R5
        reti

;-------------------------------------
; Interrupt Vectors
;-------------------------------------
        .sect   ".reset"
        .short  RESET

        .sect   ".int22"
        .short  ISR1

        .sect   ".int25"
        .short  ISR2
```

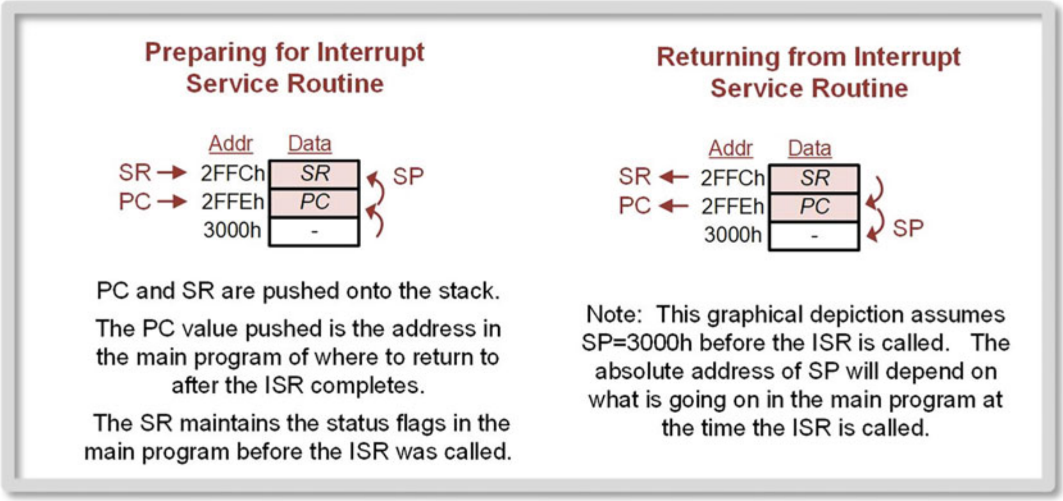
Note: This code doesn't work, it is just an example of initialization. Don't try to run on your LaunchPad™ board!

The vector table holds the starting addresses of where to load PC when the interrupt occurs.

Fig. 11.4
A graphical depiction of initializing the interrupt vector table

11.1.4 Operation of the Stack during an IRQ

When an interrupt is serviced, the CPU needs a way to save the current status of the main program so that it can resume operation after the ISR completes. The stack is used to accomplish this. When an IRQ is to be serviced, the CPU finishes its current instruction. This puts the program counter at the address of the next instruction in the main program to execute. At that point, the PC and SR are pushed onto the stack. The MCU then proceeds to clear the SR, so that it can be used by the ISR. The MCU then retrieves the starting address of the ISR from the interrupt's vector address and loads it into PC. This now puts the CPU in a position where it can execute the instructions in the ISR. Once the ISR completes, the CPU pops the SR and PC from the stack. This then moves the program operation back to the original place in the main program where it left off. Figure 11.5 shows a graphical depiction of the operation of a stack during an interrupt.


Preparing for Interrupt Service Routine
Addr
Data
SR
PC
2FFCh
2FFEh
3000h
SR
PC
-
SP
Returning from Interrupt Service Routine
Addr
Data
SR
PC
2FFCh
2FFEh
3000h
SR
PC
-
SP

PC and SR are pushed onto the stack.

The PC value pushed is the address in the main program of where to return to after the ISR completes.

The SR maintains the status flags in the main program before the ISR was called.

Note: This graphical depiction assumes SP=3000h before the ISR is called. The absolute address of SP will depend on what is going on in the main program at the time the ISR is called.

Fig. 11.5
Operation of the stack during an interrupt

11.1.5 Interrupt Service Routines (ISR)

An interrupt service routine is written in a similar manner as a subroutine. They both need to start with an address label and contain instructions to be executed when called; however, an ISR must end with a dedicated instruction called *return from interrupt* (`reti`). The `reti` instruction will pop the SR and PC off of the stack in order to return the CPU execution back to the main program. While the MCU automatically pushes PC and SR onto the stack when an interrupt is to be serviced, it is the job of the developer to explicitly pop the SR and PC off of the stack at the end of the ISR using the `reti` instruction.

Another critical role of a maskable ISR is that it must clear the peripheral's local interrupt flag (IFG) that caused the interrupt in the first place. If the IFG is not cleared, then as soon as the ISR completes and the CPU returns to the main program, it will be immediately interrupted again because the flag is still asserted. This leads to an infinite ISR loop that the CPU can never get out of.

ISRs should be short, fast, and dedicated to only performing the functionality needed by the peripheral at that time. A good ISR should impact the rest of the MCU as little as possible. One common programming approach to minimizing the impact of an ISR on the main program is to push the general-purpose CPU registers to the stack as its first act and then pop the CPU registers as its last action. This action can be omitted if the CPU registers are not used in the ISR to speed up ISR execution. The developer should pay close attention to whether the instructions in their ISR use the CPU registers, especially emulated instructions. When a CPU register is used by the ISR in any capacity other than passing variables back and forth, they should be preserved by pushing/popping them using the stack.

11.1.6 Nested Interrupts

As mentioned earlier, another one of the automatic steps that is accomplished by the interrupt system is to clear SR before entering the ISR. This means that maskable interrupts are disabled while executing an ISR because GIE = 0. Once the ISR completes and the SR is popped off the stack, the original value of GIE = 1 is restored and maskable interrupts are again enabled when the CPU returns to the main program. System resets and non-maskable interrupts can always interrupt other lower priority ISRs because they are not enabled by GIE. This is by design because system resets and non-maskable interrupts are high priority events that must be serviced immediately to ensure the proper operation of the MCU and/or prevent damage to the device. If it is desired to allow a maskable interrupt to interrupt another maskable IRQ, then the developer must explicitly set the GIE bit within the ISR. Creating *nested* ISRs is not recommended because it can lead to stack overflow or infinite ISR loops. A better approach is to create ISRs that are short and fast and then rely on the MCU's prioritization scheme to allow higher priority interrupts to be serviced while lower priority interrupts wait in the pending state until the CPU is ready to act on them.

11.1.7 Interrupt Servicing Summary

Figure 11.6 shows a flow chart of the steps that are taken when an IRQ is serviced. Note that some of the steps are taken automatically by the CPU while some are up to the developer.

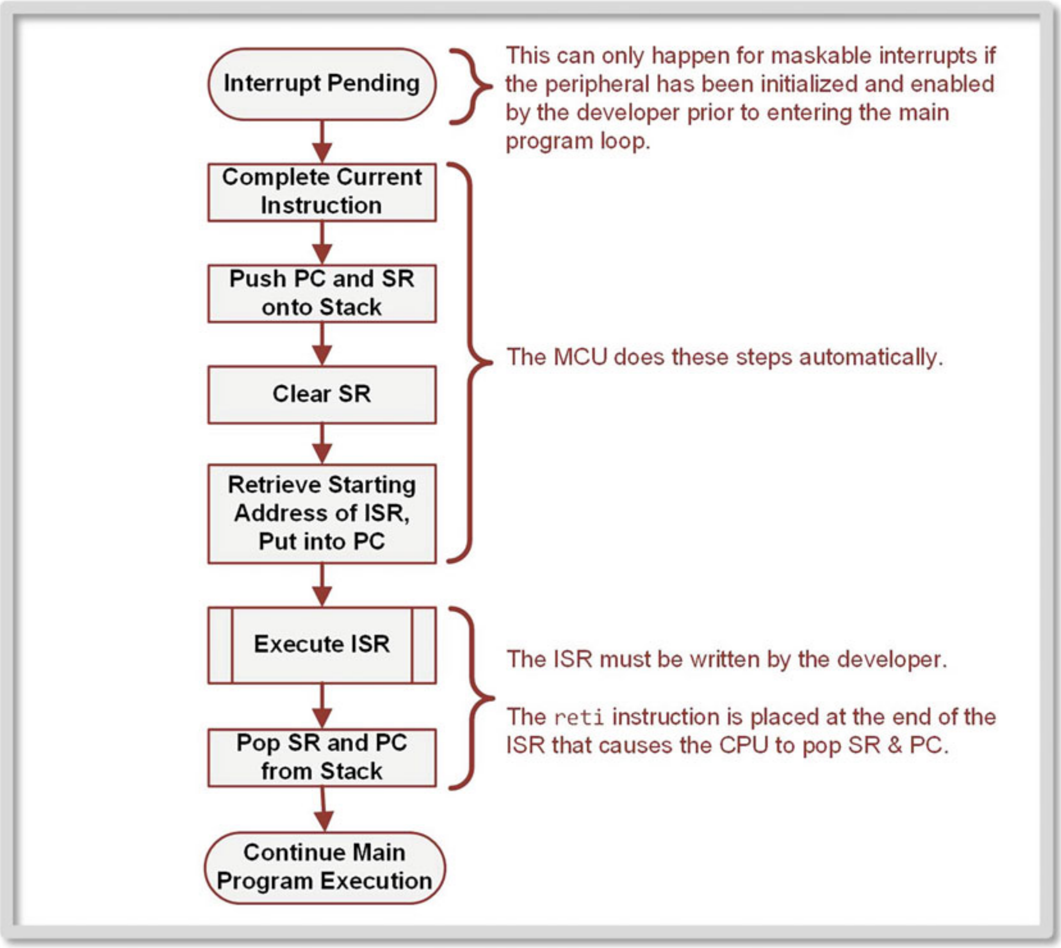

Interrupt Pending
This can only happen for maskable interrupts if the peripheral has been initialized and enabled by the developer prior to entering the main program loop.
Complete Current Instruction
Push PC and SR onto Stack
Clear SR
Retrieve Starting Address of ISR, Put into PC
The MCU does these steps automatically.
Execute ISR
Pop SR and PC from Stack
The ISR must be written by the developer.
The reti instruction is placed at the end of the ISR that causes the CPU to pop SR & PC.
Continue Main Program Execution

Fig. 11.6
Sequence of tasks performed when servicing an interrupt

When using maskable interrupts, it is important to keep in mind which tasks are taken care of automatically by the MCU and those that the developer must do. When using a specific peripheral with a maskable interrupt, the developer has the following responsibilities:

1. Configure the peripheral for the desired functionality.
2. Clear the peripheral's interrupt flag (IFG).
3. Assert the local interrupt enable (IE) for the peripheral.
4. Assert the global interrupt enable (GIE) in the status register.
5. Write the ISR with an address label to mark its starting location and the `reti` instruction to denote its end. Remember that the ISR must clear the peripherals local interrupt flag (IFG) so that when the ISR completes, the peripheral doesn't inadvertently trigger another IRQ.
6. Initialize the vector address for the peripheral using the ISR address label and assembler directives.

11.1.8 MSP430FR2355 Interrupts

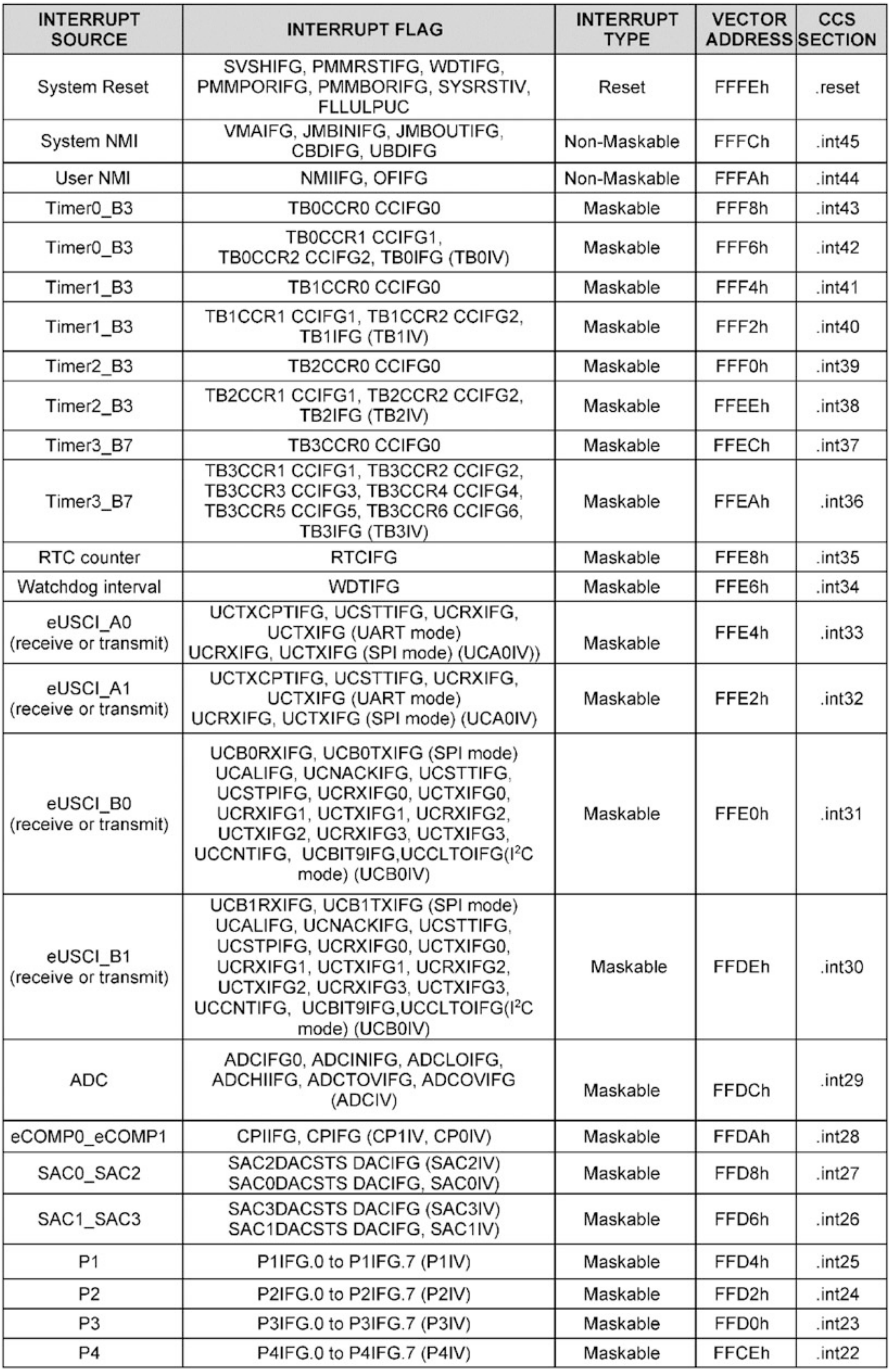
The MSP430FR2355 implements 25 unique interrupt vector addresses. Each address is used for multiple interrupt flags. If multiple interrupts that share a vector address are enabled, then functionality must be placed in the ISR to first determine which flag has been asserted and then execute the appropriate service routine code.

The highest priority vector address implemented (FFFEh) is dedicated to system resets. This vector is always initialized with the starting address of program memory so that if any type of reset occurs, the MCU begins executing the main program code from its beginning. As mentioned before, the MCU does not execute an ISR for system resets. Instead, it sets all configuration registers to their default faults and loads PC with the starting address of program memory. In the case of the MSP430FR2355, PC is set to 8000h, which is the beginning of nonvolatile FRAM program memory.

The second and third highest priority addresses implemented (FFFCh and FFFAh) are dedicated to non-maskable interrupts. Again, multiple situations are tied to each address. The vector FFFCh is used for system non-maskable interrupts, which are hardware-level failures such as accessing memory addresses that don't have any systems mapped to them, memory access timing errors, and memory bit-error detection. The vector FFFAh is used for user non-maskable interrupts, which include an external input trigger and oscillator faults. Both NMI interrupt vectors execute ISRs when triggered, so it is the developer's responsibility to ensure the ISRs exist and the vector addresses are initialized. During prototyping, the ISRs for the NMIs are often omitted due to the low likelihood of a system failure. But any MCU put into an embedded application needs to have ISRs for NMIs to handle these failure conditions.

The remaining 22 interrupt vectors are used for maskable interrupts. None of these interrupts are enabled unless explicitly done so by the developer. The MSP430FR2355 assigns eight of the vectors to the timer system, one to the real time clock counter, one for the watchdog timer, four for the serial communication system, one for the ADC, one for the comparator, two for the smart analog combo DACs, and one for each of the four digital I/O ports.

Table 11.1 gives the interrupt vector table addresses, the associated sources, and the flags that are associated with the vector. Also in this table are the CCS section names that can be used when initializing the vector table using assembler directives.



INTERRUPT SOURCE	INTERRUPT FLAG	INTERRUPT TYPE	VECTOR ADDRESS	CCS SECTION
System Reset	SVSHIFG, PMMRSTIFG, WDTIFG, PMMPORIFG, PMMBORIFG, SYSRSTIV, FLLULPUC	Reset	FFFEh	.reset
System NMI	VMAIFG, JMBINIFG, JMBOUTIFG, CBDIFG, UBDIFG	Non-Maskable	FFFCh	.int45
User NMI	NMIIFG, OFIFG	Non-Maskable	FFFAh	.int44
Timer0_B3	TB0CCR0 CCIFG0	Maskable	FFF8h	.int43
Timer0_B3	TB0CCR1 CCIFG1, TB0CCR2 CCIFG2, TB0IFG (TB0IV)	Maskable	FFF6h	.int42
Timer1_B3	TB1CCR0 CCIFG0	Maskable	FFF4h	.int41
Timer1_B3	TB1CCR1 CCIFG1, TB1CCR2 CCIFG2, TB1IFG (TB1IV)	Maskable	FFF2h	.int40
Timer2_B3	TB2CCR0 CCIFG0	Maskable	FFF0h	.int39
Timer2_B3	TB2CCR1 CCIFG1, TB2CCR2 CCIFG2, TB2IFG (TB2IV)	Maskable	FFEEh	.int38
Timer3_B7	TB3CCR0 CCIFG0	Maskable	FFECh	.int37
Timer3_B7	TB3CCR1 CCIFG1, TB3CCR2 CCIFG2, TB3CCR3 CCIFG3, TB3CCR4 CCIFG4, TB3CCR5 CCIFG5, TB3CCR6 CCIFG6, TB3IFG (TB3IV)	Maskable	FFEAh	.int36
RTC counter	RTCIFG	Maskable	FFE8h	.int35
Watchdog interval	WDTIFG	Maskable	FFE6h	.int34
eUSCI_A0 (receive or transmit)	UCTXCPTIFG, UCSTTIFG, UCRXIFG, UCTXIFG (UART mode) UCRXIFG, UCTXIFG (SPI mode) (UCA0IV))	Maskable	FFE4h	.int33
eUSCI_A1 (receive or transmit)	UCTXCPTIFG, UCSTTIFG, UCRXIFG, UCTXIFG (UART mode) UCRXIFG, UCTXIFG (SPI mode) (UCA0IV)	Maskable	FFE2h	.int32
eUSCI_B0 (receive or transmit)	UCB0RXIFG, UCB0TXIFG (SPI mode) UCALIFG, UCNACKIFG, UCSTTIFG, UCSTPIFG, UCRXIFG0, UCTXIFG0, UCRXIFG1, UCTXIFG1, UCRXIFG2, UCTXIFG2, UCRXIFG3, UCTXIFG3, UCCNTIFG, UCBIT9IFG,UCCLTOIFG(I^{2}C mode) (UCB0IV)	Maskable	FFE0h	.int31
eUSCI_B1 (receive or transmit)	UCB1RXIFG, UCB1TXIFG (SPI mode) UCALIFG, UCNACKIFG, UCSTTIFG, UCSTPIFG, UCRXIFG0, UCTXIFG0, UCRXIFG1, UCTXIFG1, UCRXIFG2, UCTXIFG2, UCRXIFG3, UCTXIFG3, UCCNTIFG, UCBIT9IFG,UCCLTOIFG(I^{2}C mode) (UCB0IV)	Maskable	FFDEh	.int30
ADC	ADCIFG0, ADCINIFG, ADCLOIFG, ADCHIIFG, ADCTOVIFG, ADCOVIFG (ADCIV)	Maskable	FFDCh	.int29
eCOMP0_eCOMP1	CPIIFG, CPIFG (CP1IV, CP0IV)	Maskable	FFDAh	.int28
SAC0_SAC2	SAC2DACSTS DACIFG (SAC2IV) SAC0DACSTS DACIFG, SAC0IV)	Maskable	FFD8h	.int27
SAC1_SAC3	SAC3DACSTS DACIFG (SAC3IV) SAC1DACSTS DACIFG, SAC1IV)	Maskable	FFD6h	.int26
P1	P1IFG.0 to P1IFG.7 (P1IV)	Maskable	FFD4h	.int25
P2	P2IFG.0 to P2IFG.7 (P2IV)	Maskable	FFD2h	.int24
P3	P3IFG.0 to P3IFG.7 (P3IV)	Maskable	FFD0h	.int23
P4	P4IFG.0 to P4IFG.7 (P4IV)	Maskable	FFCEh	.int22

Table 11.1
MSP430FR2355 interrupt vector table (for assembly)

Concept Check

CC11.1 Why are interrupts given a priority?

A) They aren't. The priority just gives a way to make documentation easier to read.

B) Priority allows the MCU hardware designers to spend less time on some circuits compared to other, more important peripherals.

C) Some events are more important than others and need immediate attention.

D) Priority gives the software designers an idea of which services routines to implement first.

11.2 MSP430FR2355 Port Interrupts

Each bit within ports 1 → 4 on the MSP430FR2355 has the ability to trigger an interrupt when configured as an input and there is a transition on its pin. Ports 1 → 4 each have their own dedicated vector address; however, each bit within each port all share the port's vector. So, it is the job of the developer to determine which bit triggered the interrupt manually and which part of the associated ISR to execute to service that bit.

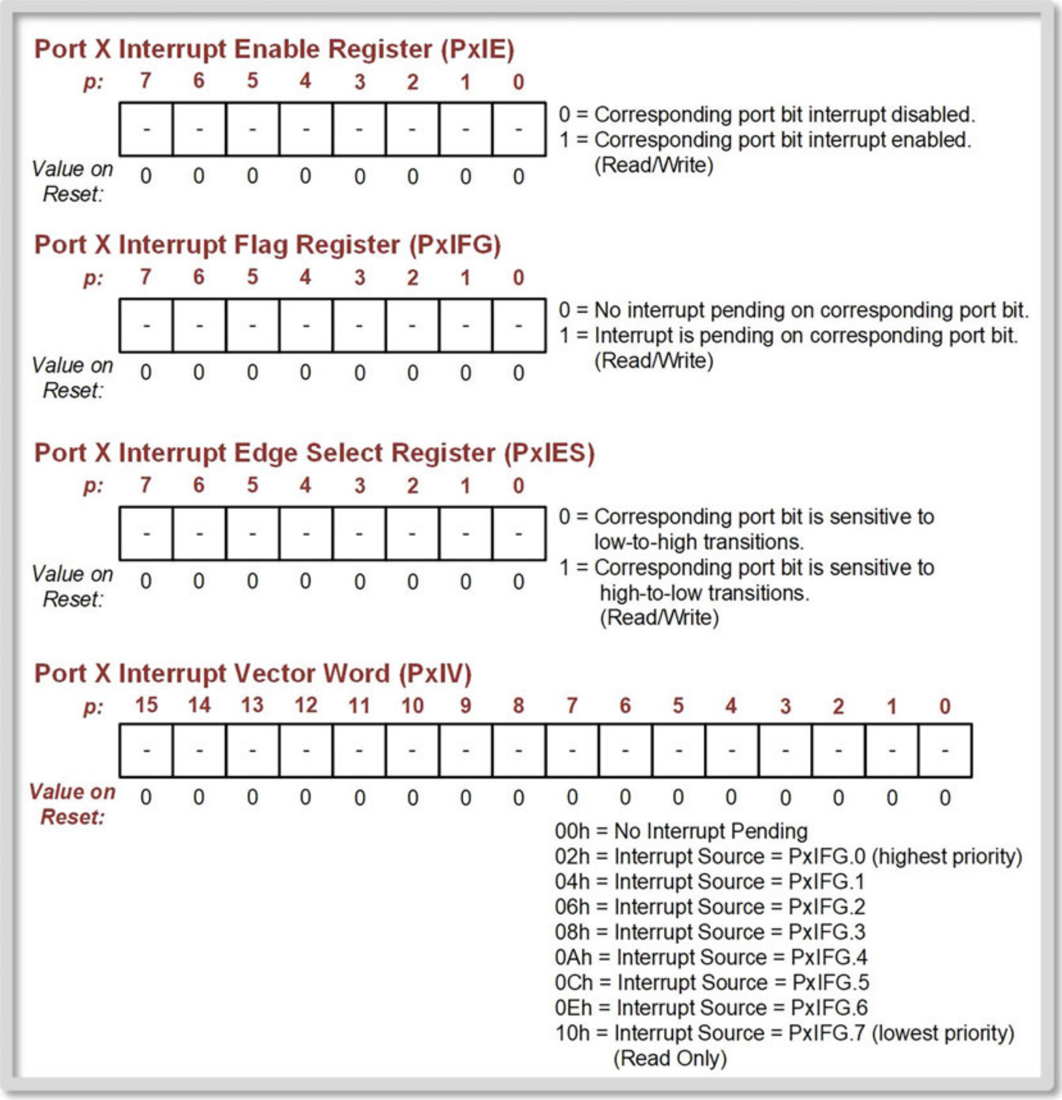
The local enable for the port interrupts are configured in the *Port Px Interrupt Enable* (PxIE, or P1IE, P2IE, P3IE, and P4IE) registers. Each bit within PxIE corresponds to the bit of the port (i.e., bit 0 of P1IE enables the interrupt on bit 0 of P1). A 0 in PxIE indicates that the interrupt for that bit of the port is disabled. A 1 in PxIE indicates that the interrupt for that bit of the port is enabled. PxIE is cleared on reset, disabling the port interrupts. Since the port interrupts are maskable, the global interrupt for all bits is the GIE bit in the status register.

The flags for the port interrupts are held in the *Port Px Interrupt Flag* (PxIFG, or P1IFG, P2IFG, P3IFG, and P4IFG) registers. Upon reset, all bits in PxIFG are set to 0. Upon an interrupt, the flag is asserted. Each bit within PxIFG corresponds to the bit of the port (i.e., if bit 0 of P1IFG is asserted it means an interrupt has occurred on bit 0 of P1). Once a port IRQ is serviced, the bit's flag needs to be cleared in PxIFG by the developer.

The port interrupt system provides a prioritization scheme that can speed up determining which bit of the port should be serviced first if multiple IRQs occur on a port simultaneously. The port interrupt system prioritizes bit 0 as the highest priority and bit 7 as the lowest priority within the port. Dedicated registers called the *Port Px Interrupt Vector Word* (PxIV, or P1IV, P2IV, P3IV, and P4IV) registers are used to indicate priority when simultaneous port IRQs have occurred. PxIV is loaded with a unique number corresponding to the bit that has just triggered an IRQ and also has the highest priority of any bits within the port that may have also triggered an IRQ. This number can be used within the service routine to quickly jump to the code associated with the highest priority bit. PxIV does not have a bitwise correspondence to the port bit that caused the interrupt. The values it takes on represent which of the 8 inputs is pending with the highest priority. The values it will take on are: bit0 = 02h, bit1 = 04h, bit2 = 06h, bit3 = 08h, bit4 = 0Ah, bit5 = 0Ch, bit6 = 0Ch, and bit7 = 10h. This register is read-only; however, any access to PxIV (either read or write) will clear the port interrupt flag in PxIFG corresponding to the highest priority being displayed in PxIV. This allows the system to handle the interrupts in the order of their priority. Once the highest priority flag is cleared, it is expected that the ISR will complete and the next highest priority flag will trigger and the next highest priority bit will be serviced within the same vector address. Note that using this prioritization feature is optional and the developer can simply use PxIFG to determine which port bit has triggered the interrupt and which one to service first.

The last configuration setting for port interrupts is the ability to select which transition polarity triggers the interrupt (i.e., rising or falling). The *Port Px Interrupt Edge Select* (PxIES or P1IES, P2IES, P3IES, and P4IES) registers. A 0 in this register means the IRQ will be triggered on a low-to-high transition on the pin. A 1 in this register means the IRQ will be triggered on a high-to-low transition. Each bit within this register corresponds to the bit in the port it configures (i.e., if bit 0 of P1IES is a 1, and then a high-to-low transition on bit 0 of P1 will trigger an IRQ).

Figure 11.7 gives a summary of the port interrupt configuration registers.



Port X Interrupt Enable Register (PxIE)

p:	7	6	5	4	3	2	1	0
	-	-	-	-	-	-	-	-
Value on Reset:	0	0	0	0	0	0	0	0

0 = Corresponding port bit interrupt disabled.
1 = Corresponding port bit interrupt enabled.
(Read/Write)

Port X Interrupt Flag Register (PxIFG)

p:	7	6	5	4	3	2	1	0
	-	-	-	-	-	-	-	-
Value on Reset:	0	0	0	0	0	0	0	0

0 = No interrupt pending on corresponding port bit.
1 = Interrupt is pending on corresponding port bit.
(Read/Write)

Port X Interrupt Edge Select Register (PxIES)

p:	7	6	5	4	3	2	1	0
	-	-	-	-	-	-	-	-
Value on Reset:	0	0	0	0	0	0	0	0

0 = Corresponding port bit is sensitive to low-to-high transitions.
1 = Corresponding port bit is sensitive to high-to-low transitions.
(Read/Write)

Port X Interrupt Vector Word (PxIV)

p:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Value on Reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

00h = No Interrupt Pending
02h = Interrupt Source = PxIFG.0 (highest priority)
04h = Interrupt Source = PxIFG.1
06h = Interrupt Source = PxIFG.2
08h = Interrupt Source = PxIFG.3
0Ah = Interrupt Source = PxIFG.4
0Ch = Interrupt Source = PxIFG.5
0Eh = Interrupt Source = PxIFG.6
10h = Interrupt Source = PxIFG.7 (lowest priority)
(Read Only)

Fig. 11.7
Summary of port interrupt configuration registers

When using port interrupts, there is a recommended initialization sequence to avoid inadvertent bit assertions of flags due to the nature of power on. After reset, all ports are put into high-impedance input mode with Schmitt triggers. In this state, the interrupts flags are susceptible to their values being changed due to cross-currents associated with writing to other port configuration registers. As such, there are some configuration steps that should be done prior to clearing the LOCKLPM5 bit (taking the inputs out of high-impedance mode with Schmitt trigger mode) and some that are done after. The recommended sequence from the MSP430FR2355 data sheet to configure a port interrupt is as follows:

1. Initialize the port direction (PxDIR), pull-up/down resistor (PxREN), the pull-up/down resistor polarity (PxOUT), and the port interrupt edge select (PxIES).
2. Clear LOCKLPM5 bit.
3. Clear the port interrupt flags (PxIFG) for first use. Note that the reset value for PxIFG=00h, but often bits will be asserted inadvertently due to step 1.
4. Assert the local port interrupt enable (PxIE).
5. Assert the global enable for maskable interrupts (GIE bit in SR).

Let's now look at configuring the push-button switch S1 on the LaunchPad^{TM} board to trigger a port interrupt. Let's design a program that will toggle LED1 each time S1 is pressed using an interrupt. First, let's look at the signal behavior of S1 when pressed. Figure 11.8 shows a graphical depiction of the logic levels and transitions that occur when S1 is pressed and released.

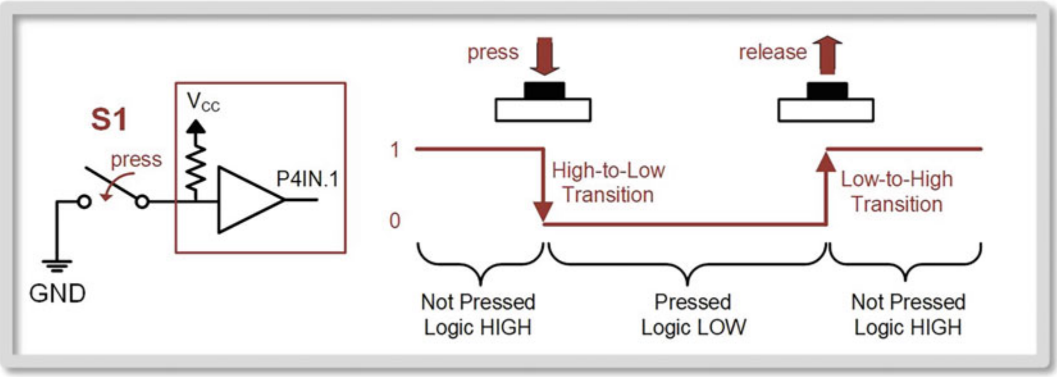


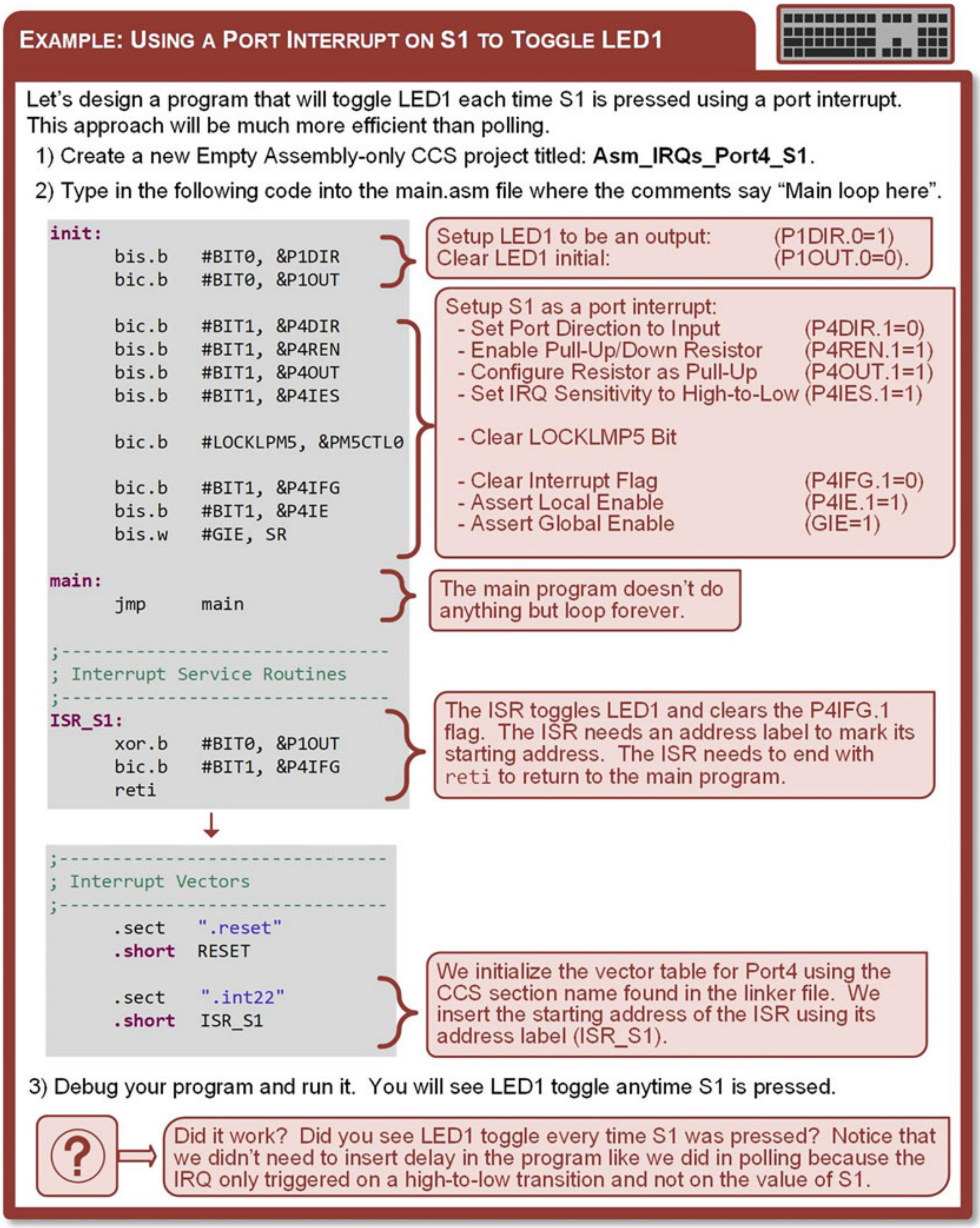
Fig. 11.8
Signal behavior of P4IN.1 when S1 is pressed

A few things to keep in mind when setting up a push-button interrupt:

- S1 is connected to Port4, bit 1. While the logic level of S1 can be observed on P4IN.1, when using an interrupt, we don't have to look at this bit. We instead allow a transition to assert an interrupt flag and have the CPU execute an ISR accordingly.
- S1 is an SPST switch that is connected to ground. This means we need a pull-up resistor on the MCU to provide the logic HIGH state when S1 is not pressed.
- If we want the IRQ to trigger immediately upon a button *press*, then we need to configure the interrupt edge sensitivity (P4IES.1) to be high to low. When S1 is not pressed, P4.1 is at a logic high. When the button is pressed, P4.1 goes to a logic low. If we leave the P4IES.1 sensitivity at its default value of low-to-high sensitivity, the interrupt will only trigger once S1 is released (we'll look at this behavior in a following example).
- Since we are only using one bit within P4 to trigger an IRQ, we don't need to use the P4IV register to determine the highest priority bit that caused the IRQ. We will simply use the P4IFG register knowing that we only care about bit1.

- The port 4 interrupt vector address is FFCEh. This has a CCS section name of .int22. This is the name we will use when we initialize the vector address using assembler directives.
- In the ISR, we will need to toggle LED1, clear the P4IFG.4 flag, and use `reti` to return from the interrupt.

Follow Example 11.1 to see how a port interrupt can accomplish this functionality.



EXAMPLE: USING A PORT INTERRUPT ON S1 TO TOGGLE LED1

Let's design a program that will toggle LED1 each time S1 is pressed using a port interrupt. This approach will be much more efficient than polling.

1) Create a new Empty Assembly-only CCS project titled: **Asm_IRQs_Port4_S1**.

2) Type in the following code into the main.asm file where the comments say "Main loop here".

```
init:
        bis.b   #BIT0, &P1DIR
        bic.b   #BIT0, &P1OUT

        bic.b   #BIT1, &P4DIR
        bis.b   #BIT1, &P4REN
        bis.b   #BIT1, &P4OUT
        bis.b   #BIT1, &P4IES

        bic.b   #LOCKLPM5, &PM5CTL0

        bic.b   #BIT1, &P4IFG
        bis.b   #BIT1, &P4IE
        bis.w   #GIE, SR

main:
        jmp     main

;-------------------------------
; Interrupt Service Routines
;-------------------------------
ISR_S1:
        xor.b   #BIT0, &P1OUT
        bic.b   #BIT1, &P4IFG
        reti
```

```
;-------------------------------
; Interrupt Vectors
;-------------------------------
        .sect   ".reset"
        .short  RESET

        .sect   ".int22"
        .short  ISR_S1
```

Setup LED1 to be an output: (P1DIR.0=1)
Clear LED1 initial: (P1OUT.0=0).

Setup S1 as a port interrupt:
- Set Port Direction to Input (P4DIR.1=0)
- Enable Pull-Up/Down Resistor (P4REN.1=1)
- Configure Resistor as Pull-Up (P4OUT.1=1)
- Set IRQ Sensitivity to High-to-Low (P4IES.1=1)
- Clear LOCKLMP5 Bit
- Clear Interrupt Flag (P4IFG.1=0)
- Assert Local Enable (P4IE.1=1)
- Assert Global Enable (GIE=1)

The main program doesn't do anything but loop forever.

The ISR toggles LED1 and clears the P4IFG.1 flag. The ISR needs an address label to mark its starting address. The ISR needs to end with `reti` to return to the main program.

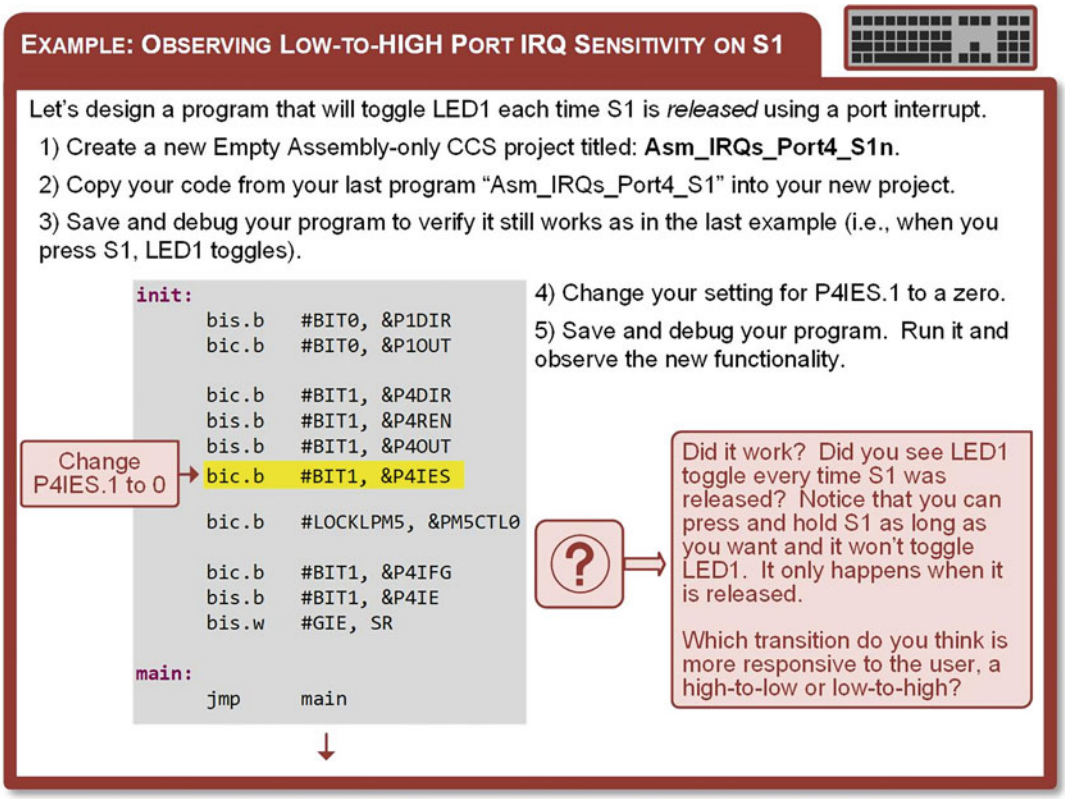
We initialize the vector table for Port4 using the CCS section name found in the linker file. We insert the starting address of the ISR using its address label (ISR_S1).

3) Debug your program and run it. You will see LED1 toggle anytime S1 is pressed.

Did it work? Did you see LED1 toggle every time S1 was pressed? Notice that we didn't need to insert delay in the program like we did in polling because the IRQ only triggered on a high-to-low transition and not on the value of S1.

Example 11.1
Using a port interrupt on S2 to toggle LED1

Now let's take a look at the impact of the edge sensitivity on the button press. Follow Example 11.2 to see how changing the edge select value to be sensitive to a low-to-high transition makes the IRQ trigger upon a button *release*.



EXAMPLE: OBSERVING LOW-TO-HIGH PORT IRQ SENSITIVITY ON S1

Let's design a program that will toggle LED1 each time S1 is *released* using a port interrupt.

1) Create a new Empty Assembly-only CCS project titled: **Asm_IRQs_Port4_S1n**.

2) Copy your code from your last program "Asm_IRQs_Port4_S1" into your new project.

3) Save and debug your program to verify it still works as in the last example (i.e., when you press S1, LED1 toggles).

```
init:
        bis.b   #BIT0, &P1DIR
        bic.b   #BIT0, &P1OUT

        bic.b   #BIT1, &P4DIR
        bis.b   #BIT1, &P4REN
        bis.b   #BIT1, &P4OUT
        bic.b   #BIT1, &P4IES

        bic.b   #LOCKLPM5, &PM5CTL0

        bic.b   #BIT1, &P4IFG
        bis.b   #BIT1, &P4IE
        bis.w   #GIE, SR

main:
        jmp     main
```

Change P4IES.1 to 0

4) Change your setting for P4IES.1 to a zero.

5) Save and debug your program. Run it and observe the new functionality.

Did it work? Did you see LED1 toggle every time S1 was released? Notice that you can press and hold S1 as long as you want and it won't toggle LED1. It only happens when it is released.

Which transition do you think is more responsive to the user, a high-to-low or low-to-high?

Example 11.2
Observing low-to-high port IRQ sensitivity on S1

Now let's take a look at a common programming error that occurs when using IRQs. Follow Example 11.3 to see what happens when the developer forgets to clear the IRQ flag in the ISR.

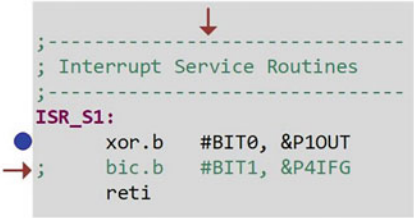
Example: Observing the Impact of Not Clearing the IRQ Flag

Let's look at what happens when the ISR doesn't clear the IRQ flag.

1) Open your last example project: **Asm_IRQs_Port4_S1n**.

2) In your ISR, comment out the instruction that clears the P4IFG.1 flag.

3) Save and debug your program. Set a breakpoint at the first instruction in your ISR. Run your program.



```
;-------------------------------
; Interrupt Service Routines
;-------------------------------
ISR_S1:
        xor.b    #BIT0, &P1OUT
;       bic.b    #BIT1, &P4IFG
        reti
```

4) Now press and release S1. Your program will run to the breakpoint in your ISR and pause.

5) Now step your program. You will see that the ISR continually triggers and you can never get out of it.

Do you see the problem? Not remembering to clear the flag in the ISR is one of the most common mistakes developers make when using IRQs. They are very hard to debug without setting breakpoints in the ISR.

6) Make sure to remove your comment and save your program so that you have a functioning project in your workspace.

Example 11.3
Observing the impact of not clearing the IRQ flag

Concept Check

CC11.2 If the port vector (PxIV) register isn’t used in the ISR to determine which pin interrupt has triggered, what other method can be used?

A) Probing the signal with an oscilloscope.

B) Looking at the PxIE register settings.

C) Checking the PxIFG register values in the ISR bit by bit.

D) Checking the return address pushed onto the stack.

Summary

- Interrupts provide a way to efficiently deal with peripherals that are asynchronous to the CPU and slower than the CPU clock.
- When an interrupt occurs, it asserts a flag and waits to be serviced by the CPU. While waiting for the CPU to respond to it, the interrupt is said to be pending.
- The CPU will service an interrupt only after it finishes executing its current instruction. When the interrupt is serviced, the CPU stops execution of the main program and instead executes an interrupt service routine that is dedicated to the peripheral that triggered the IRQ.
- Interrupts are prioritized in terms of their importance. When a CPU is ready to service an IRQ, it handles the highest priority first. Multiple IRQs can be pending at once.

- There are three classes of interrupts: system resets, non-maskable interrupts, and maskable interrupts.
- System resets are the highest interrupt. They do not have developer written ISR; instead, they initialize all configuration registers to their default values and insert the starting address of the main program into PC.
- Non-maskable interrupts are the next highest priority interrupts. These handle memory access or oscillator faults. They also handle a user interrupt that can come from an external signal. NMIs are always enabled and execute a developer written ISR.
- Maskable interrupts have a local and global enable. The local enable is unique to each feature of each peripheral that can trigger and interrupt. The global interrupt is the GIE bit in the SR. For an interrupt to be active, it must have both its local enable and GIE asserted.
- The starting address of an ISR is put into its interrupt vector address. The interrupt vectors reside at dedicated addresses reserved at the end of program memory. The starting address of the ISR is placed into the vector location using assembler directives and the address label of the start of the ISR.
- When an IRQ occurs, the CPU automatically pushes the PC and SR onto the stack to preserve the operation of the main program. At the end of an ISR, the developer must use the `reti` instruction to pop SR and PC off the stack to return execution to the main program.
- Interrupts can interrupt other interrupts. Since system resets and NMIs are always enabled, they can interrupt each other with higher priority IRQs being able to interrupt lower priority IRQs. When a maskable interrupt occurs, the CPU automatically clears SR, disabling other maskable interrupts.
- If a developer wishes to allow nested maskable interrupts, they need to explicitly set the GIE bit in the ISR. A better approach is to write short ISRs and allow the IRQ prioritization scheme handle the order the ISRs are executed.
- When an IRQ is served, the MCU completes the current instruction, pushes the PC and SR to the stack, clears the SR, retrieves the starting address of the ISR from the vector table, executes the ISR, and then pops SR and PC off the stack to return to normal execution in the main program.
- The MSP430FR2355 implements 25 unique interrupt vectors. Each vector is associated with multiple flags. If multiple peripheral features are enabled and share the same vector address, it is the job of the developer to determine which flag caused the IRQ within the ISR.
- A port interrupt is an IRQ that is triggered when there is an input transition observed on a bit of a port. This allows external signals to be handled more efficiently compared to polling.
- Each bit within ports 1 → 4 can trigger an interrupt; however, each port only has one unique vector address.
- Port interrupts are configured using the PxIFG, PxIES, PxIE, and PxIV (optional) registers. To use a port interrupt, the pin must also be configured as an input with an optional pull-up/down resistor.

Exercise Problems

Section 11.1: The Concept of an Interrupt

11.1.1 Why is an interrupt more efficient than polling?

11.1.2 When an interrupt has raised its flag, but is waiting for the CPU to service it, what state is the interrupt said to be in?

11.1.3 What is the sequence of instructions that is executed for an interrupt called?

11.1.4 How does the MCU decide which interrupt to handle if multiple occur at the same time?

11.1.5 Can a system reset interrupt be disabled?

11.1.6 Can a non-maskable interrupt be disabled?

11.1.7 Are maskable interrupts enabled or disabled after reset?

11.1.8 What is the global interrupt enable for maskable interrupts?

11.1.9 What is the role of an interrupt vector?

11.1.10 How does the interrupt vector get initialized?

11.1.11 Where in the memory map does the interrupt vector table reside?

11.1.12 What address value should be put into the vector table location associated with the reset condition so that after a reset occurs, the MSP430FR2355 begins executing instructions at the beginning of program memory?

11.1.13 What two registers are automatically pushed onto the stack during an interrupt?

11.1.14 What instruction is used at the end of an interrupt service routine to return to the main program?

11.1.15 By default, can a maskable interrupt interrupt another maskable interrupt?

11.1.16 How many interrupt vectors are implemented on the MSP430FR2355 MCU?

11.1.17 What is the vector address for reset on the MSP430FR2355 MCU?

11.1.18 What is the vector address for Port 1 on the MSP430FR2355 MCU?

11.1.19 What is the CCS section name for the reset vector on the MSP430FR2355 MCU?

11.1.20 What is the CCS section name for the Port 1 vector on the MSP430FR2355 MCU?

Section 11.2: MSP430FR2355 Port Interrupts

11.2.1 Are port interrupts maskable or non-maskable?

11.2.2 When using a port interrupt, what is the purpose of the PxIE register?

11.2.3 When using a port interrupt, what is the purpose of the PxIFG register?

11.2.4 When using a port interrupt, what is the purpose of the PxIV register?

11.2.5 When using a port interrupt, what is the purpose of the PxIES register?

11.2.6 In the recommended sequence of steps for using a port interrupt, does configuring PxIES occur before or after clearing the LOCKLPM5 bit?

11.2.7 In the recommended sequence of steps for using a port interrupt, does configuring PxIFG occur before or after clearing the LOCKLPM5 bit?

11.2.8 When using the SPST push-button switches on the LaunchPad^{TM} board, does a low-to-high or high-to-low transition setting on PxIES provide a more responsive experience for the user?

11.2.9 Why is it important for the developer to clear the interrupt flag in the ISR?

11.2.10 Does a program that uses a port interrupt have to use PxIV to determine which bit triggered the port interrupt? Why or why not?

Chapter 12: Introduction to Timers

This chapter introduces the concept of a timer to independently track and/or trigger events based on the time elapsed [1–3]. A timer uses an independent, free-running, binary counter to track the passage of time. Timers are a peripheral to the CPU, which allows the CPU to handle other tasks while the timer runs separately. Timers can trigger interrupts that are then serviced by the CPU through routines. This chapter presents the concept of a timer and then looks at the details of the timers available on the MSP430FR2355 MCU.

Learning Outcomes—After completing this chapter, you will be able to:

12.1 Describe the basic operation of an MCU timer.
12.2 Use the timer system on the MSP430FR2355 to generate periodic events based on timer overflows.
12.3 Use the timer system on the MSP430FR2355 to generate periodic events based on timer compares.
12.4 Use the timer system on the MSP430FR2355 to generate pulse width modulated signals based on multiple timer compares.
12.5 Use the timer system on the MSP430FR2355 to perform timer captures.

12.1 Timer Overview

A timer is a binary counter that is clocked from a free-running clock with a known frequency. Since the binary counter will increment on the triggering edge of the clock and the clock frequency is known, then the time between count values is deterministic. The time elapsed can be found by simply multiplying the period of the clock ($T = 1/f$) by the number of counts that have occurred (N). Time can be monitored using timers in a variety of ways including tracking the time between when the counter is cleared and a specific count value, the time it takes for the counter to reach its maximum value and rollover (aka, overflow), or the difference between two nonzero count values. In all cases, the amount of time elapsed is $\Delta t = T{\cdot}N$. Figure 12.1 shows a timing waveform of a 16-bit timer highlighting some key timing characteristics.

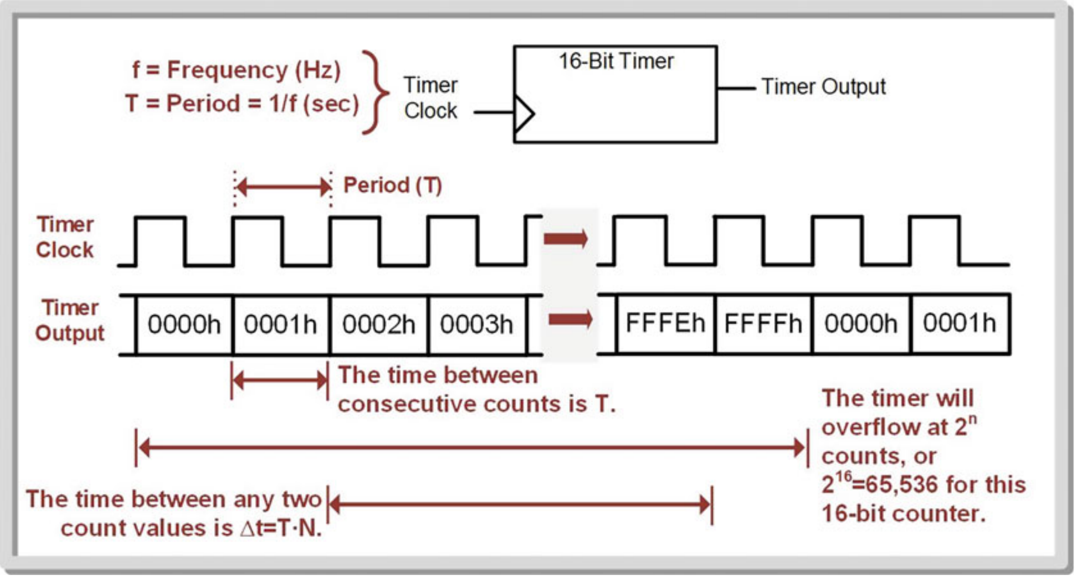
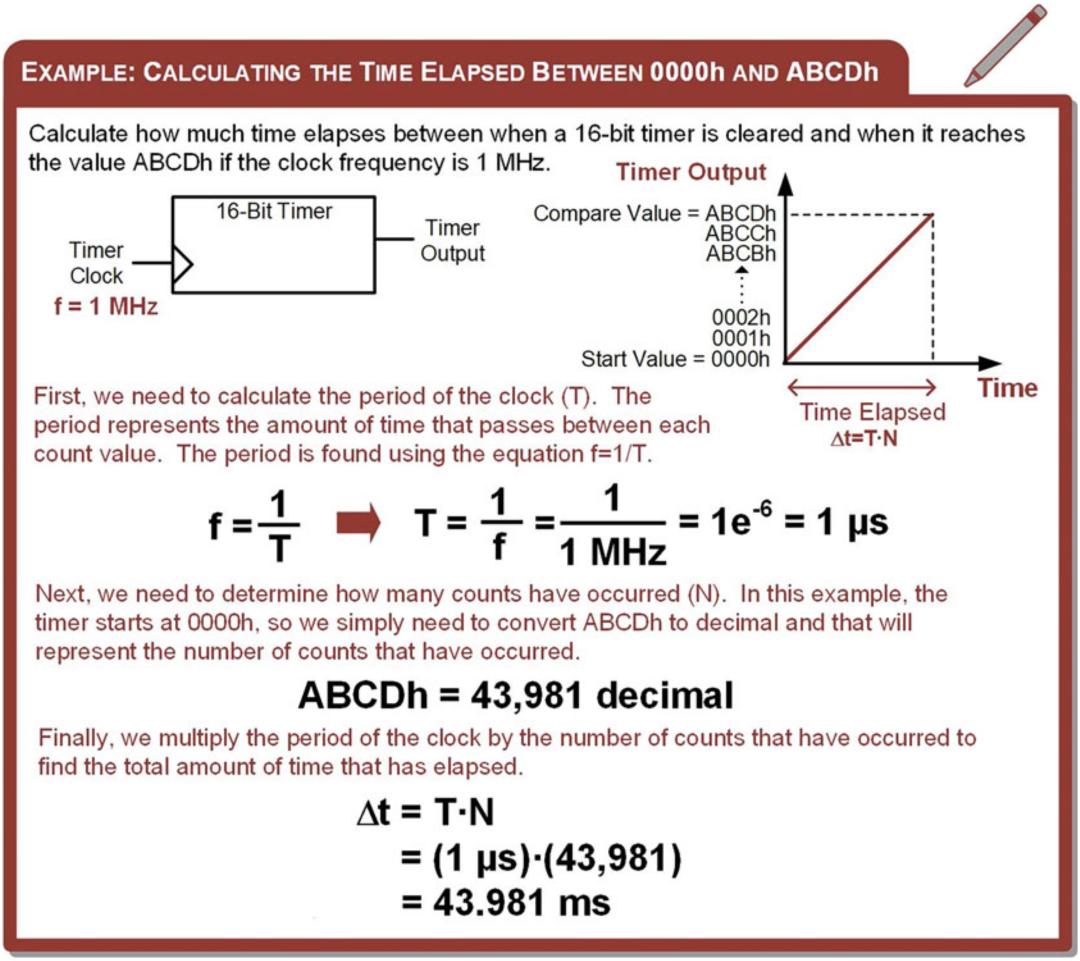

f = Frequency (Hz)
T = Period = 1/f (sec)
Timer Clock
16-Bit Timer
Timer Output
Period (T)
Timer Clock
Timer Output
0000h
0001h
0002h
0003h
FFFEh
FFFFh
0000h
0001h
The time between consecutive counts is T.
The timer will overflow at 2^n counts, or 2^16=65,536 for this 16-bit counter.
The time between any two count values is Δt=T·N.

Fig. 12.1
Timer overview

B. J. LaMeres, *Embedded Systems Design using the MSP430FR2355 LaunchPad™*,
https://doi.org/10.1007/978-3-031-20888-1_12

Let's look a few examples of calculating how much time has passed using a timer. First, let's start with finding the amount of time between when the timer starts (or is cleared) and a specific value. When the counter starts at 0, then the specific value we are comparing the timer to represents the total number of counts that have occurred (N). Example 12.1 shows an example of how to find the amount of time that has elapsed between a counter that starts at 0000h and reaches ABCDh on a 16-bit counter running off of a 1 MHz clock.



EXAMPLE: CALCULATING THE TIME ELAPSED BETWEEN 0000h AND ABCDh

Calculate how much time elapses between when a 16-bit timer is cleared and when it reaches the value ABCDh if the clock frequency is 1 MHz.

First, we need to calculate the period of the clock (T). The period represents the amount of time that passes between each count value. The period is found using the equation f=1/T.

$$f = \frac{1}{T} \Rightarrow T = \frac{1}{f} = \frac{1}{1\text{ MHz}} = 1e^{-6} = 1\ \mu s$$

Next, we need to determine how many counts have occurred (N). In this example, the timer starts at 0000h, so we simply need to convert ABCDh to decimal and that will represent the number of counts that have occurred.

ABCDh = 43,981 decimal

Finally, we multiply the period of the clock by the number of counts that have occurred to find the total amount of time that has elapsed.

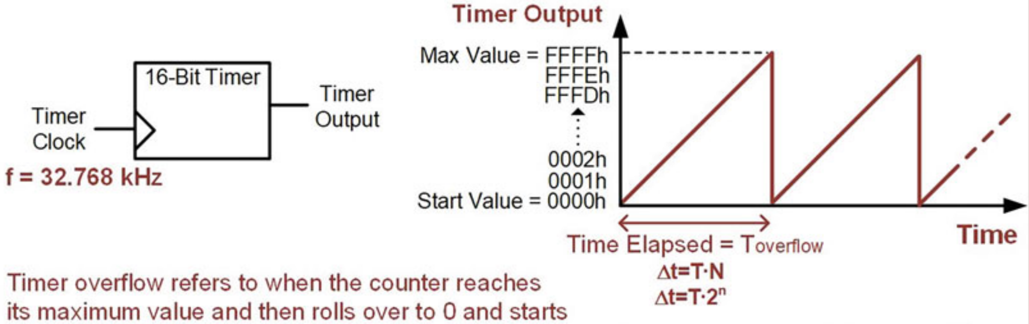
$$\begin{aligned}\Delta t &= T\cdot N \\ &= (1\ \mu s)\cdot(43{,}981) \\ &= 43.981\text{ ms}\end{aligned}$$

Example 12.1
Calculating the time elapsed between 0000h and ABCDh

Next, let's look at determining the amount of time between when a timer starts at 0 and when it reaches its maximum value and rolls over back to 0. This is called the *timer overflow period* ($T_{overflow}$). As always, the amount of time that has elapsed is found using $\Delta t = T{\cdot}N$. In the case of timer overflow, $N = 2^n$ where n is the width of the timer. Example 12.2 shows how to calculate $T_{overflow}$ of a 16-bit counter running off of a 32.768 kHz clock.

Example: Calculating the Timer Overflow Period

Calculate the timer overflow period of a 16-bit timer if the clock frequency is 32.768 kHz.



Timer overflow refers to when the counter reaches its maximum value and then rolls over to 0 and starts counting again. The time between rollovers is called the timer overflow period ($T_{overflow}$). The time elapsed on a counter is always $\Delta t = T \cdot N$, but when calculating overflow, $N = 2^n$. Let's start with finding the period of the clock.

$$f = \frac{1}{T} \Rightarrow T = \frac{1}{f} = \frac{1}{32.768\ \text{kHz}} = 30.518e^{-6} = 30.518\ \mu s$$

Next, we find the number of counts that will occur between when the timer starts at zero and when it reaches its maximum value and rolls-over (N). This is found using the expression for the number of unique codes in a binary number (i.e., 2^n where "n" is the width of the binary counter).

$$N = 2^n = 2^{16} = 65{,}536$$

Finally, we multiply the period of the clock by the number of counts in the timer to find the timer overflow period.

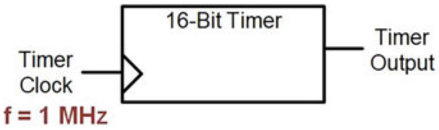
$$\begin{aligned} T_{overflow} &= T \cdot N \\ &= T \cdot 2^n \\ &= (30.518\ \mu s) \cdot (65{,}536) \\ &= 2\ s \end{aligned}$$

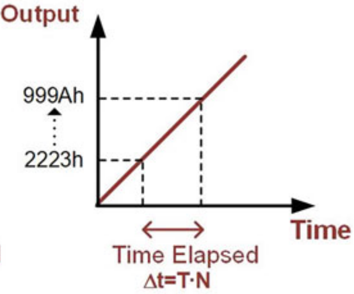
Example 12.2
Calculating the timer overflow period

Next, let's look at finding the amount of time that elapses between two specific values (i.e., where the first value is not 0). Example 12.3 shows how to calculate the time elapsed between 2223h and 999Ah on a timer with a clock frequency of 1 MHz.

Example: Calculating the Time Elapsed Between 2223h and 999Ah

Calculate how much time elapses between the values 2223h and 999Ah on a 16-bit timer if the clock frequency is 1 MHz.





First, we need to calculate the period of the clock. The period is found using the equation f=1/T.

$$f = \frac{1}{T} \Rightarrow T = \frac{1}{f} = \frac{1}{1\ \text{MHz}} = 1e^{-6} = 1\ \mu s$$

Next, we need to determine how many counts have occurred between 2223h and 999Ah (N). We can find this by simply subtracting the first value from the second value. We will need to convert the numbers to decimal at some point. This conversion can be done before or after the subtraction. Let's do the conversion before the subtraction.

2223h = 8,739 decimal **999Ah = 39,322 decimal**

Next, we subtract the first value from the second value to find the number of counts that have occurred (N).

$$\begin{aligned} N &= 39{,}322 - 8{,}739 \\ &= 30{,}583 \end{aligned}$$

Finally, we multiply the period of the clock by the number of counts that have occurred to find the total amount of time that elapsed.

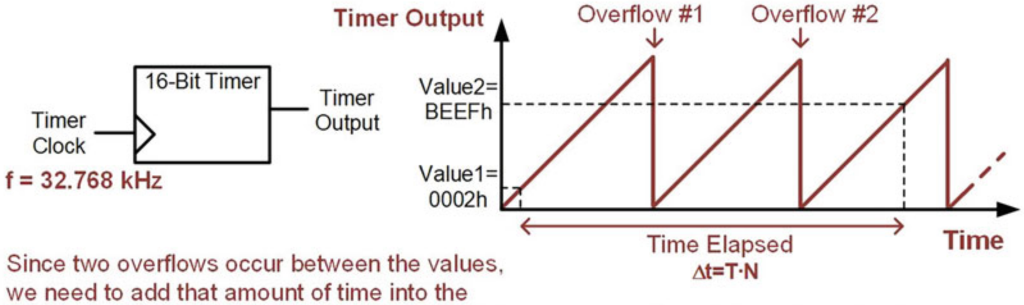
$$\begin{aligned} \Delta t &= T \cdot N \\ &= (1\ \mu s) \cdot (30{,}583) \\ &= 30.583\ \text{ms} \end{aligned}$$

Example 12.3
Calculating the time elapsed between 2223h and 999Ah

Often when tracking the time elapsed between two external events with a timer, there can be multiple timer overflows between the events. This means that we must account for the additional time beyond simply subtracting two specific timer values. Example 12.4 shows how to calculate the time elapsed between two timer values when multiple overflows occur between the events.

Example: Calculating the Time Between Values with Overflows

Calculate how much time elapses between the values 0002h and BEEFh on a 16-bit timer if two overflows occur between the values and the clock frequency is 32.768 kHz.



Since two overflows occur between the values, we need to add that amount of time into the calculation of how much time has elapsed. We can assume that we have put code into our program to count the number of overflows and it is provided to us as an integer. We simply add the number of counts in each overflow to the second timer value and then subtract the first timer value. The total amount of time elapsed then becomes:

$$\Delta t = T\cdot[((\#\text{ of Overflows})\cdot 2^{n} + \text{Value2}) - \text{Value1}]$$

Let's first calculate the period of the clock.

$$f = \frac{1}{T} \quad ➡ \quad T = \frac{1}{f} = \frac{1}{32.768\text{ kHz}} = 30.518e^{-6} = 30.518\ \mu s$$

Next, let's convert our two timer values into decimal so that the calculations are simpler.

$$0002h = 2\text{ decimal} \qquad BEEFh = 48{,}879\text{ decimal}$$

Finally, we add all of the counts together and multiply by the period of the clock to find the total amount of time elapsed.

$$\begin{aligned}\Delta t &= T\cdot N\\ &= T\cdot[((\#\text{ of Overflows})\cdot 2^{n} + \text{Value2}) - \text{Value1}]\\ &= (30.518\text{ us})\cdot[((2)\cdot 2^{16} + 48{,}879) - 2]\\ &= 5.491\text{ s}\end{aligned}$$

Example 12.4
Calculating the time between values with overflows

The full MSP430 architecture contains three distinct timer sub-systems: Timer_A, Timer_B, and the real-time clock counter (RTC). Within the Timer_A and Timer_B systems, there are multiple, independent binary counters that provide separate timing capability. Each timer can generate interrupts when its value either matches a value placed into a compare register or when it overflows. The timers also have the ability to capture the current count value and store it into a register upon a triggering event. The capture and compare registers (CCRs) are shared and referred to as capture/compare blocks in the MSP430 documentation. The MSP430FR2355 does not implement the Timer_A system, and we will cover the RTC system later. As such, this chapter focuses on the details of the MSP430FR2355 Timer_B system.

Concept Check

CC12.1 If we use a clock frequency of 1 MHz to drive a counter and attempt to generate an event every 1ms, what could possibly create error in our timer that would prevent us from not getting an event exactly every 1 ms?

A) The counter might skip a few counts.

B) The counter could get interrupted by another peripheral.

C) Nothing. We will get exactly 1 ms events if we calculate the count value correctly.

D) The clock frequency might not be exactly 1 MHz.

12.2 Timer Overflows on the MSP430FR2355

The MSP430FR2355 Timer_B system provides four independent timers (TB0, TB1, TB2, and TB3), each with selectable clock inputs and the ability to divide down the clock to get slower counting frequencies. Timers TB0, TB1, and TB2 each have three capture/compare registers associated with them. As such, these timers are often referred to as Timer0_B3, Timer1_B3, and Timer2_B3 in the MSP430 documentation. Timer TB3 has seven capture/compare registers so it is often referred to as Timer3_B7. Note that capture/compares will be covered in the next sections. Figure 12.2 shows an overview of the Timer_B architecture implemented on the MSP430FR2355.

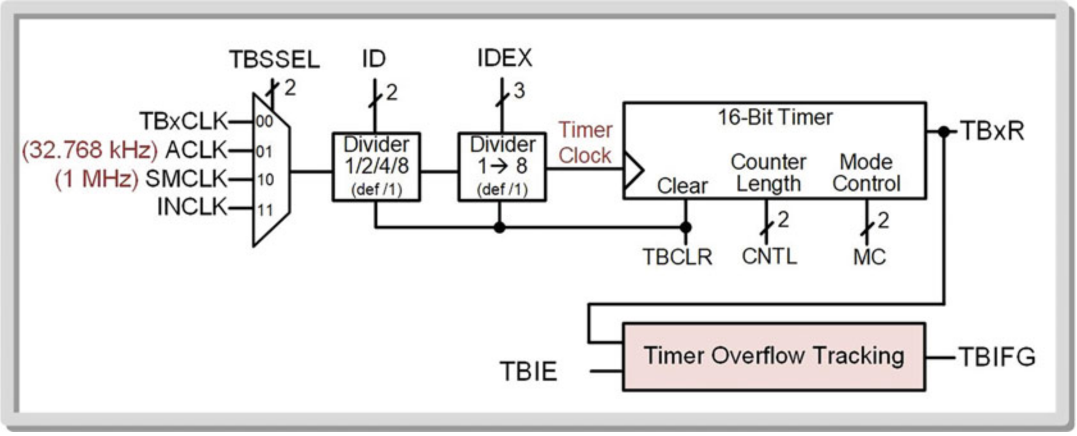
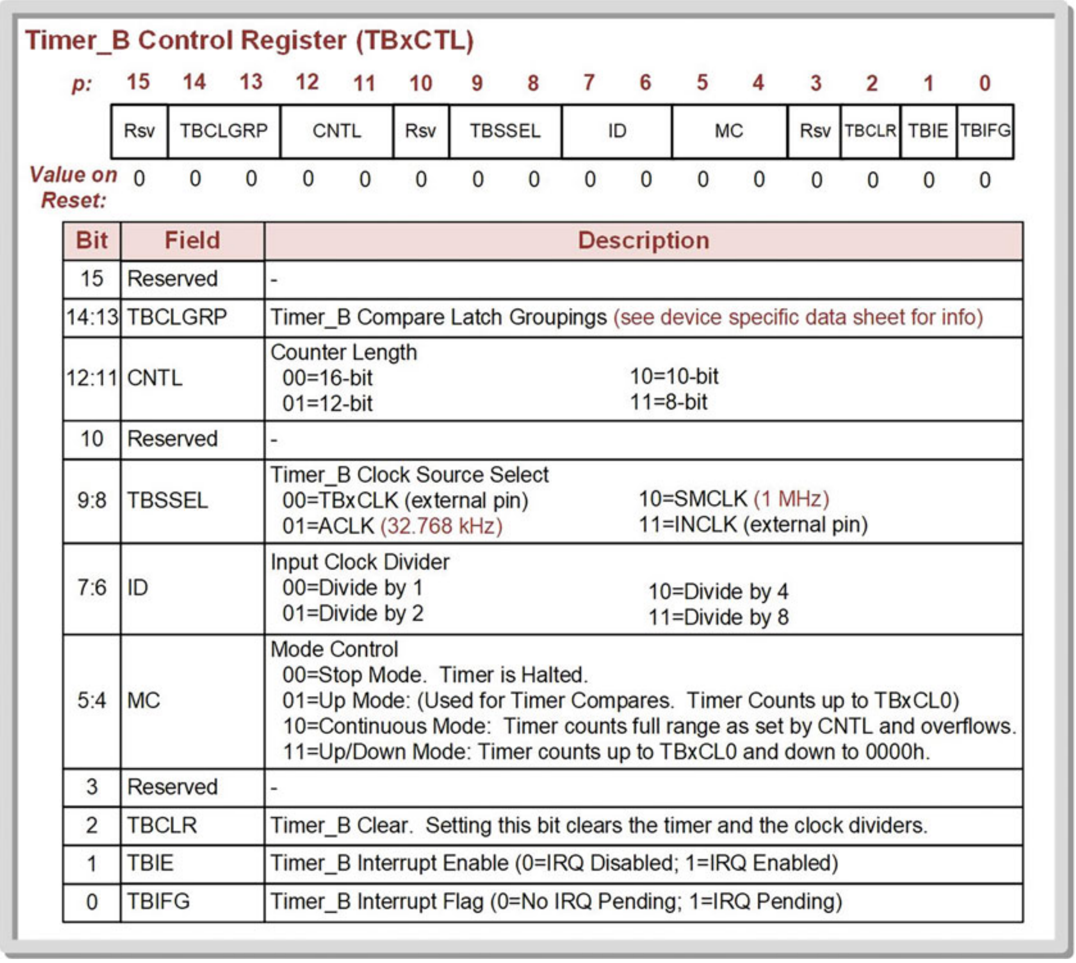


Fig. 12.2
MSP430 Timer B overview

At the core of the Timer_B system is a 16-bit, binary counter. The counter has a variety of settings that are controlled by the user including the ability to program its counter length (16-bit, 12-bit, 10-bit, or 8-bit), the counting mode (halted, count up to a value, continuous counting, or up/down counting), and the ability to clear the counter. The timer clock also has a variety of settings that allow the user to select a frequency that is appropriate for their application. The first setting available for the timer clock is its source. The timer system clock can come from one of two external pins (TBxCLK or INCLK) or from one of two on-chip clock sources (ACLK or SMCLK). On the MSP430FR2355, ACLK has a frequency of 32.768 kHz and SMCLK has a frequency of 1 MHz. The timer system also allows the user to divide down the incoming clock source in order to achieve even slower counting frequencies. There are two clock dividers implemented in series in the Timer_B system. The first divider can divide the clock by 1, 2, 4, or 8. The second divider can divide the clock by 1, 2, 3, 4, 5, 6, 7, or 8. Since these two dividers are in series, there are 32 different divider settings that can be applied to the clock ranging from a minimum divider of 1 to a maximum divider of 64.

When the Timer_B is put into continuous counting mode, it will count up to its maximum value and then rollover to 0. When it goes from its maximum value (i.e., FFFFh for 16-bit counting mode) to 0000h, a timer overflow is detected and can generate an interrupt. The local enable for this overflow interrupt is TBIE. This interrupt is maskable, so its global enable is GIE. When enabled, the interrupt will assert the timer overflow flag TBIFG.

All of the settings to control the Timer_B system(s) and use its timer overflow interrupts are held in two configuration registers, the *Timer_B Control Register* (TBxCTL) and the *Timer_B Expansion Register 0* (TBxEX0). Figures 12.3 and 12.4 give the details of the TBxCTL and TBxEX0 registers respectively. Note that these figures describe the control registers in general terms in order to describe the functionality for multiple Timer_B timers. The "x" in these register names can take on a value of 0 → 3 to denote a specific binary counter. The MSP430FR2355 contains four Timer_Bs (TB0, TB1, TB2, and TB3), each with their own control register (TB0CTL, TB1CTL, TB2CTL, and TB3CTL) and expansion register (TB0EX0, TB1EX0, TB2EX0, and TB3EX0).

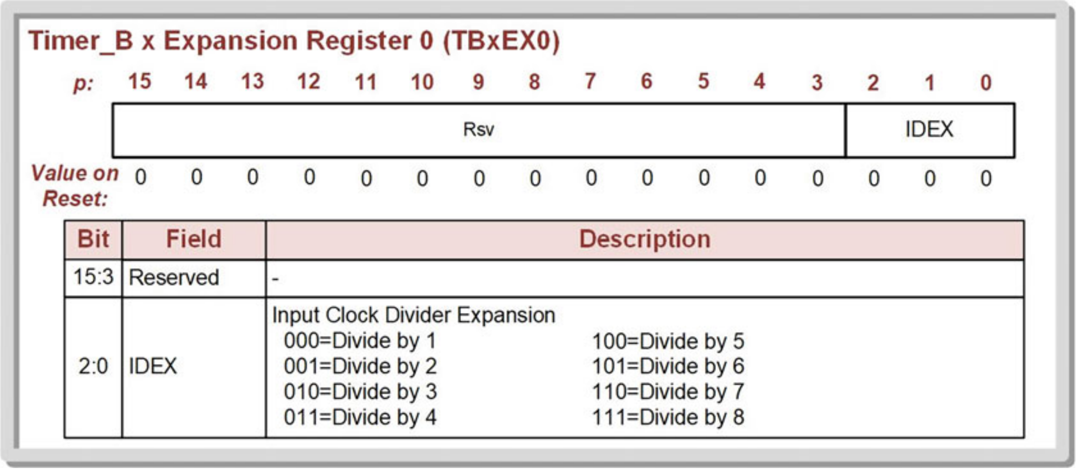


Timer_B Control Register (TBxCTL)

p:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Rsv	TBCLGRP		CNTL		Rsv	TBSSEL		ID		MC		Rsv	TBCLR	TBIE	TBIFG
Value on Reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	Field	Description
15	Reserved	-
14:13	TBCLGRP	Timer_B Compare Latch Groupings (see device specific data sheet for info)
12:11	CNTL	Counter Length 00=16-bit 10=10-bit 01=12-bit 11=8-bit
10	Reserved	-
9:8	TBSSEL	Timer_B Clock Source Select 00=TBxCLK (external pin) 10=SMCLK (1 MHz) 01=ACLK (32.768 kHz) 11=INCLK (external pin)
7:6	ID	Input Clock Divider 00=Divide by 1 10=Divide by 4 01=Divide by 2 11=Divide by 8
5:4	MC	Mode Control 00=Stop Mode. Timer is Halted. 01=Up Mode: (Used for Timer Compares. Timer Counts up to TBxCL0) 10=Continuous Mode: Timer counts full range as set by CNTL and overflows. 11=Up/Down Mode: Timer counts up to TBxCL0 and down to 0000h.
3	Reserved	-
2	TBCLR	Timer_B Clear. Setting this bit clears the timer and the clock dividers.
1	TBIE	Timer_B Interrupt Enable (0=IRQ Disabled; 1=IRQ Enabled)
0	TBIFG	Timer_B Interrupt Flag (0=No IRQ Pending; 1=IRQ Pending)

Fig. 12.3
Timer_B control register (TBxCTL) details



Timer_B x Expansion Register 0 (TBxEX0)

p:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Rsv													IDEX		
Value on Reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	Field	Description
15:3	Reserved	-
2:0	IDEX	Input Clock Divider Expansion 000=Divide by 1 100=Divide by 5 001=Divide by 2 101=Divide by 6 010=Divide by 3 110=Divide by 7 011=Divide by 4 111=Divide by 8

Fig. 12.4
Timer_B expansion register 0 (TBxEX0) details

Let's now look at using a timer overflow to generate an event at a specific time interval. The recommended sequence of programming steps to configure the counter is as follows:

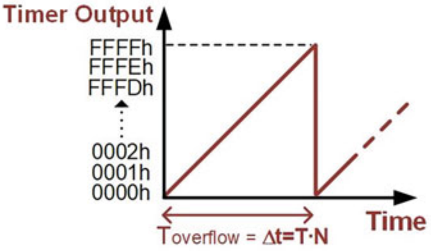
1. Write a 1 to the TBCLR bit (TBCLR = 1) to clear TBxR, the clock divider states, and the counter direction.
2. Apply desired configurations to TBxCTL.

Examples 12.5 and 12.6 provide an example of using the TB0 timer to generate an interrupt every 2 seconds. In this example, TB0 will use ACLK as its source and use the default settings of the two clock dividers (i.e., divide-by-1). The timer will run with a 16-bit length (default) and in continuous mode so that overflows happen indefinitely. When a timer overflow occurs, an interrupt will be triggered and the ISR will toggle LED1.

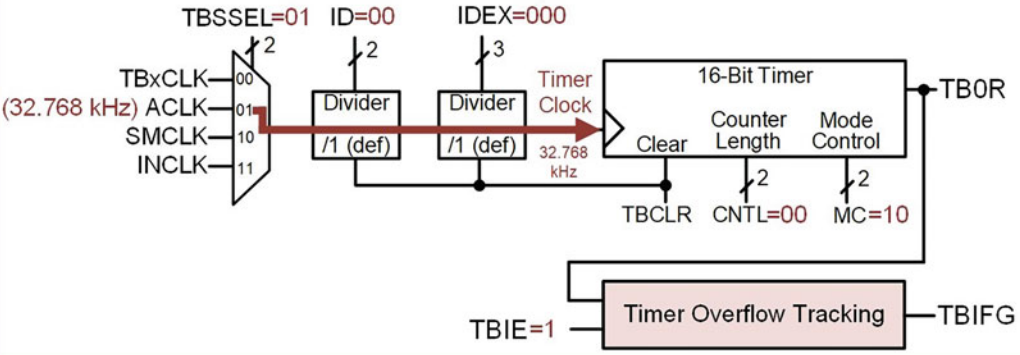
Example: Toggling LED1 on TB0 Overflow Using ACLK (part 1)

Let's design a program that will toggle LED1 every time TB0 overflows when clocked with **ACLK**. First, let's calculate how long the timer overflow period is using this configuration. We will not divide down ACLK so the timer clock will be 32.768 kHz.

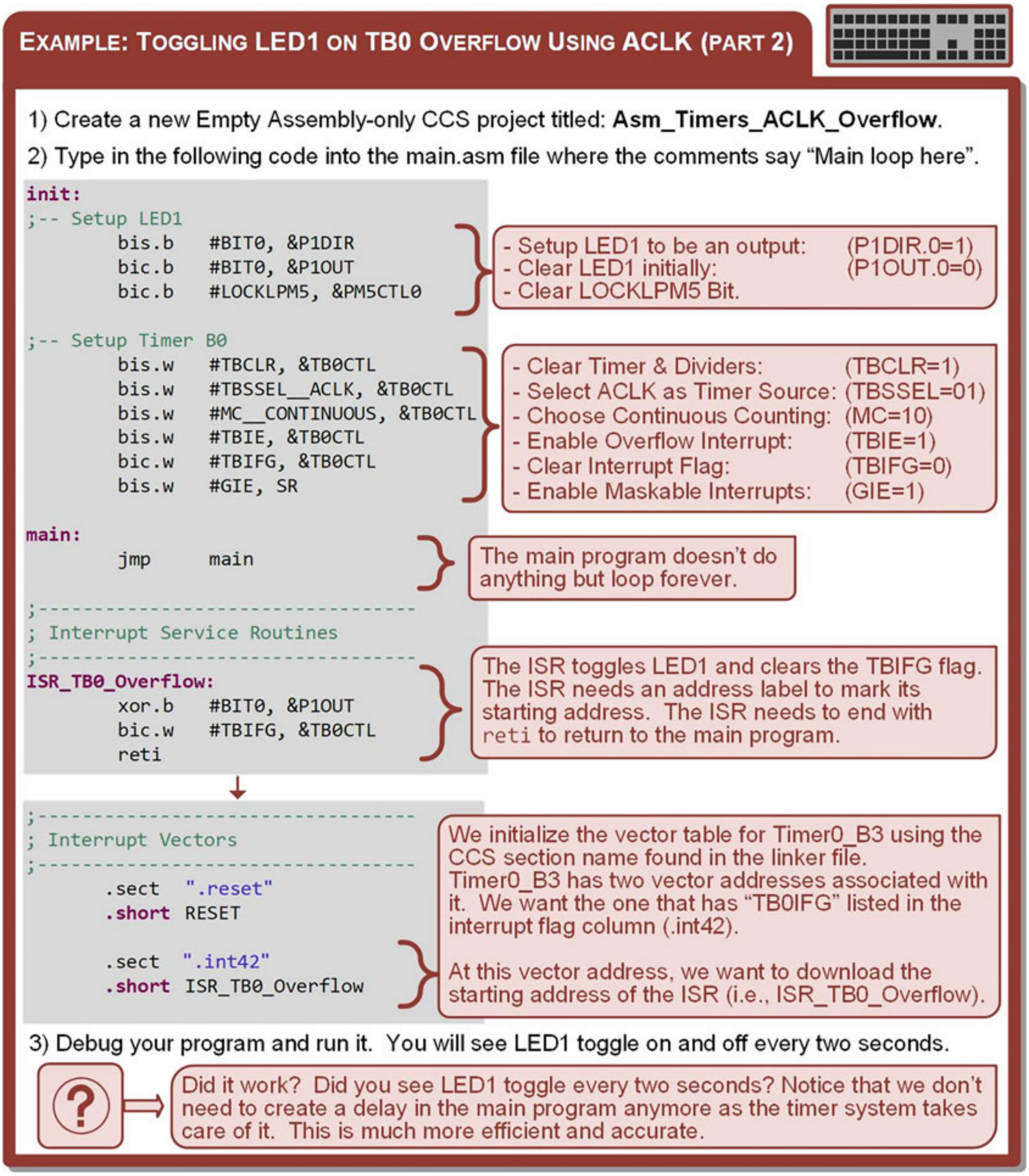
$$\mathbf{T_{overflow} = T\cdot N = (1/f)\cdot 2^{n} = (1/32.768k)\cdot 2^{16}}$$
$$\mathbf{= 2\ seconds}$$



Next, we need to decide on the settings for TB0. We want the clock source to be ACLK with both dividers set to 1. We also want the counter length to be 16-bits and the counting mode to be continuous. We then need to enable TBIE. We can use the default values for the dividers and the counter length to make our program more compact. The following is our timer setup:



Example 12.5
Toggling LED1 on TB0 overflow using ACLK (part 1)



EXAMPLE: TOGGLING LED1 ON TB0 OVERFLOW USING ACLK (PART 2)

1) Create a new Empty Assembly-only CCS project titled: **Asm_Timers_ACLK_Overflow**.

2) Type in the following code into the main.asm file where the comments say “Main loop here”.

```
init:
;-- Setup LED1
        bis.b   #BIT0, &P1DIR
        bic.b   #BIT0, &P1OUT
        bic.b   #LOCKLPM5, &PM5CTL0
```

- Setup LED1 to be an output: (P1DIR.0=1)
- Clear LED1 initially: (P1OUT.0=0)
- Clear LOCKLPM5 Bit.

```
;-- Setup Timer B0
        bis.w   #TBCLR, &TB0CTL
        bis.w   #TBSSEL__ACLK, &TB0CTL
        bis.w   #MC__CONTINUOUS, &TB0CTL
        bis.w   #TBIE, &TB0CTL
        bic.w   #TBIFG, &TB0CTL
        bis.w   #GIE, SR
```

- Clear Timer & Dividers: (TBCLR=1)
- Select ACLK as Timer Source: (TBSSEL=01)
- Choose Continuous Counting: (MC=10)
- Enable Overflow Interrupt: (TBIE=1)
- Clear Interrupt Flag: (TBIFG=0)
- Enable Maskable Interrupts: (GIE=1)

```
main:
        jmp     main
```

The main program doesn’t do anything but loop forever.

```
;-----------------------------------
; Interrupt Service Routines
;-----------------------------------
ISR_TB0_Overflow:
        xor.b   #BIT0, &P1OUT
        bic.w   #TBIFG, &TB0CTL
        reti
```

The ISR toggles LED1 and clears the TBIFG flag. The ISR needs an address label to mark its starting address. The ISR needs to end with `reti` to return to the main program.

```
;-----------------------------------
; Interrupt Vectors
;-----------------------------------
       .sect  ".reset"
       .short RESET

       .sect  ".int42"
       .short ISR_TB0_Overflow
```

We initialize the vector table for Timer0_B3 using the CCS section name found in the linker file. Timer0_B3 has two vector addresses associated with it. We want the one that has “TB0IFG” listed in the interrupt flag column (.int42).

At this vector address, we want to download the starting address of the ISR (i.e., ISR_TB0_Overflow).

3) Debug your program and run it. You will see LED1 toggle on and off every two seconds.

? ⇨ Did it work? Did you see LED1 toggle every two seconds? Notice that we don’t need to create a delay in the main program anymore as the timer system takes care of it. This is much more efficient and accurate.

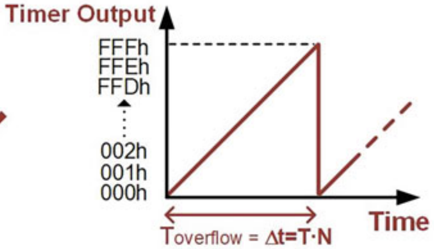
Example 12.6
Toggling LED1 on TB0 overflow using ACLK (part 2)

Now let’s look at an example where we want to use ACLK as the timer clock, but we want to *speed up the timer overflow period*. We can do this by configuring the timer to have a smaller counter length. We have the ability to choose either a 16-bit, 12-bit, 10-bit, or 8-bit length using the CNTL bits in the TB0CTL register. Let’s change the length to 12 bits using the CNTL settings. This will result in a timer overflow period of 125 ms. Examples 12.7 and 12.8 provide this example of using the TB0 timer to generate an overflow using ACLK and a 12-bit counter length.

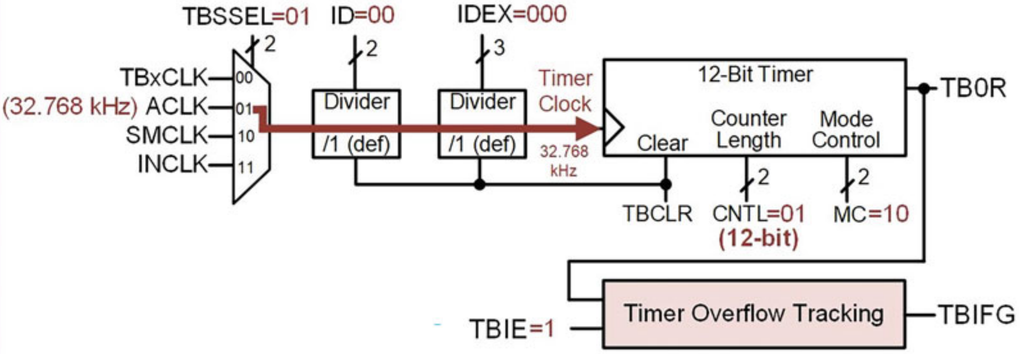
Example: Toggling LED1 on TB0 Overflow Using ACLK and a 12-bit Counter Configuration (Part 1)

Let's design a program that will toggle LED1 every time TB0 overflows when clocked with **ACLK**, but in order to **speed up the overflow period**, we'll configure the timer as 12-bits. First, let's calculate the timer overflow period.

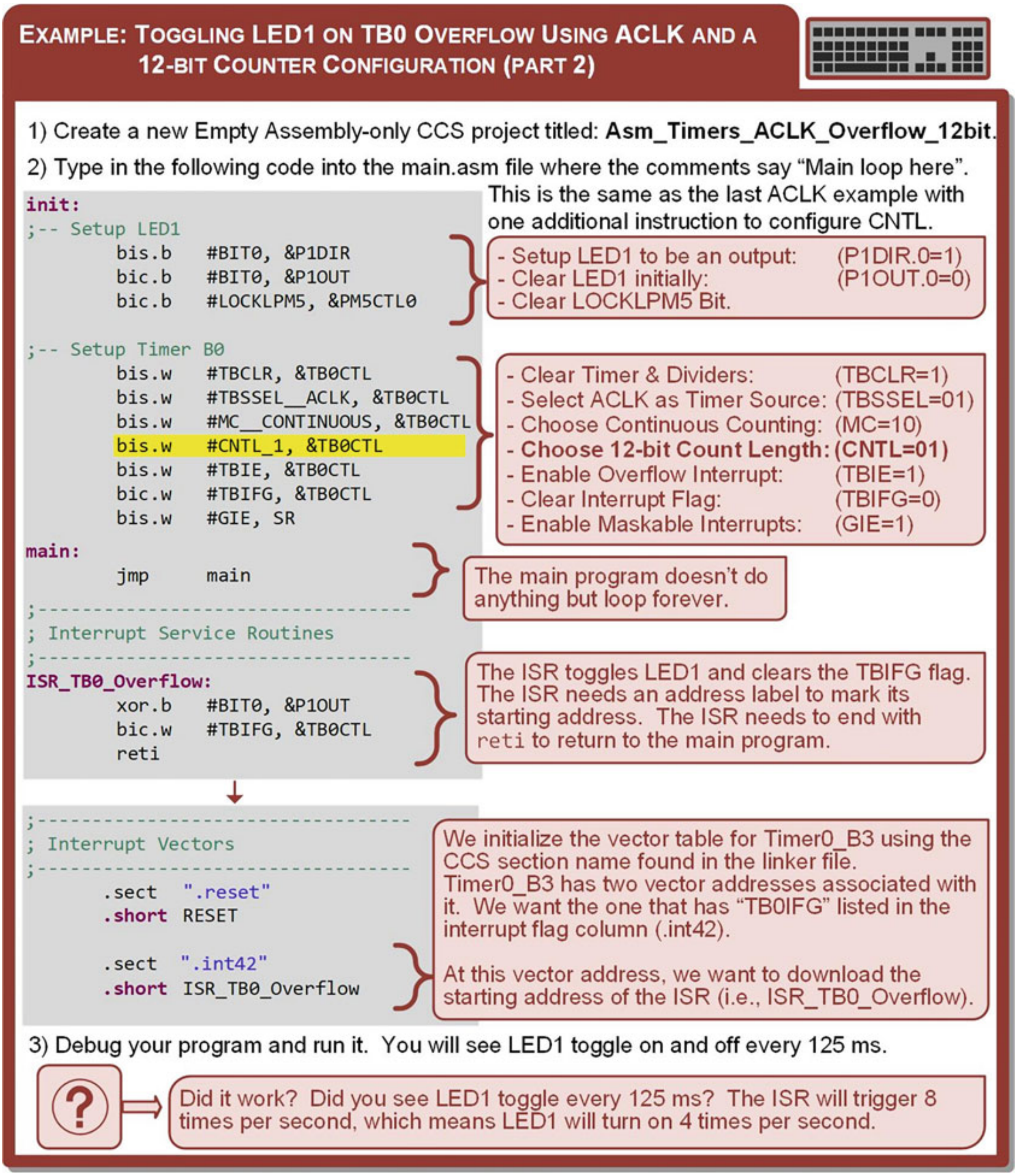
$$T_{overflow} = T\cdot N = (1/f)\cdot 2^n = (1/32.768k)\cdot 2^{12}$$
$$= 125 \text{ ms } (\sim 8 \text{ times per second})$$



Next, we need to decide on the settings for TB0. We want the clock source to be ACLK with both dividers set to 1. We also want the counter length to be **12-bits** and the counting mode to be continuous. We then need to enable TBIE. We can use the default values for the dividers and the counter length to make our program more compact. The following is our timer setup:



Example 12.7
Toggling LED1 on TB0 overflow using ACLK and a 12-bit counter configuration (part 1)



EXAMPLE: TOGGLING LED1 ON TB0 OVERFLOW USING ACLK AND A 12-BIT COUNTER CONFIGURATION (PART 2)

1) Create a new Empty Assembly-only CCS project titled: **Asm_Timers_ACLK_Overflow_12bit**.

2) Type in the following code into the main.asm file where the comments say "Main loop here".

This is the same as the last ACLK example with one additional instruction to configure CNTL.

```
init:
;-- Setup LED1
        bis.b   #BIT0, &P1DIR
        bic.b   #BIT0, &P1OUT
        bic.b   #LOCKLPM5, &PM5CTL0

;-- Setup Timer B0
        bis.w   #TBCLR, &TB0CTL
        bis.w   #TBSSEL__ACLK, &TB0CTL
        bis.w   #MC__CONTINUOUS, &TB0CTL
        bis.w   #CNTL_1, &TB0CTL
        bis.w   #TBIE, &TB0CTL
        bic.w   #TBIFG, &TB0CTL
        bis.w   #GIE, SR
main:
        jmp     main
;----------------------------------
; Interrupt Service Routines
;----------------------------------
ISR_TB0_Overflow:
        xor.b   #BIT0, &P1OUT
        bic.w   #TBIFG, &TB0CTL
        reti
```

- Setup LED1 to be an output: (P1DIR.0=1)
- Clear LED1 initially: (P1OUT.0=0)
- Clear LOCKLPM5 Bit.

- Clear Timer & Dividers: (TBCLR=1)
- Select ACLK as Timer Source: (TBSSEL=01)
- Choose Continuous Counting: (MC=10)
- **Choose 12-bit Count Length: (CNTL=01)**
- Enable Overflow Interrupt: (TBIE=1)
- Clear Interrupt Flag: (TBIFG=0)
- Enable Maskable Interrupts: (GIE=1)

The main program doesn't do anything but loop forever.

The ISR toggles LED1 and clears the TBIFG flag. The ISR needs an address label to mark its starting address. The ISR needs to end with `reti` to return to the main program.

```
;----------------------------------
; Interrupt Vectors
;----------------------------------
      .sect   ".reset"
      .short  RESET

      .sect   ".int42"
      .short  ISR_TB0_Overflow
```

We initialize the vector table for Timer0_B3 using the CCS section name found in the linker file. Timer0_B3 has two vector addresses associated with it. We want the one that has "TB0IFG" listed in the interrupt flag column (.int42).

At this vector address, we want to download the starting address of the ISR (i.e., ISR_TB0_Overflow).

3) Debug your program and run it. You will see LED1 toggle on and off every 125 ms.

? ⇒ Did it work? Did you see LED1 toggle every 125 ms? The ISR will trigger 8 times per second, which means LED1 will turn on 4 times per second.

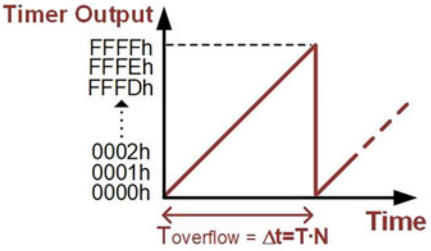
Example 12.8
Toggling LED1 on TB0 overflow using ACLK and a 12-bit counter configuration (part 2)

Let's now look at using a very similar setup as in the past examples, but this time using SMCLK as the timer clock. Examples 12.9 and 12.10 provide an example of using the TB0 timer to generate an event every 65.5 ms. In this example, TB0 will use SMCLK as its source and use the default settings of the two clock dividers (i.e., divide by 1). The timer will run with a 16-bit length (default) and in continuous mode so that overflows happen indefinitely. When a timer overflow occurs, an interrupt will be triggered and the ISR will toggle LED1. This will blink much faster than when using ACLK, but you will still be able to see LED1 blink with the naked eye.

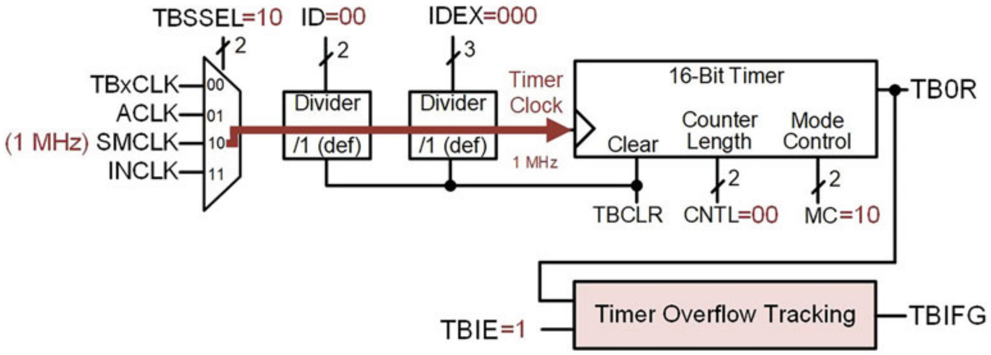
Example: Toggling LED1 on TB0 Overflow Using SMCLK (Part 1)

Let's design a program that will toggle LED1 every time TB0 overflows when clocked with **SMCLK**. First, let's calculate how long the timer overflow period is using this configuration. We will not divide down SMCLK so the timer clock will be 1 MHz.

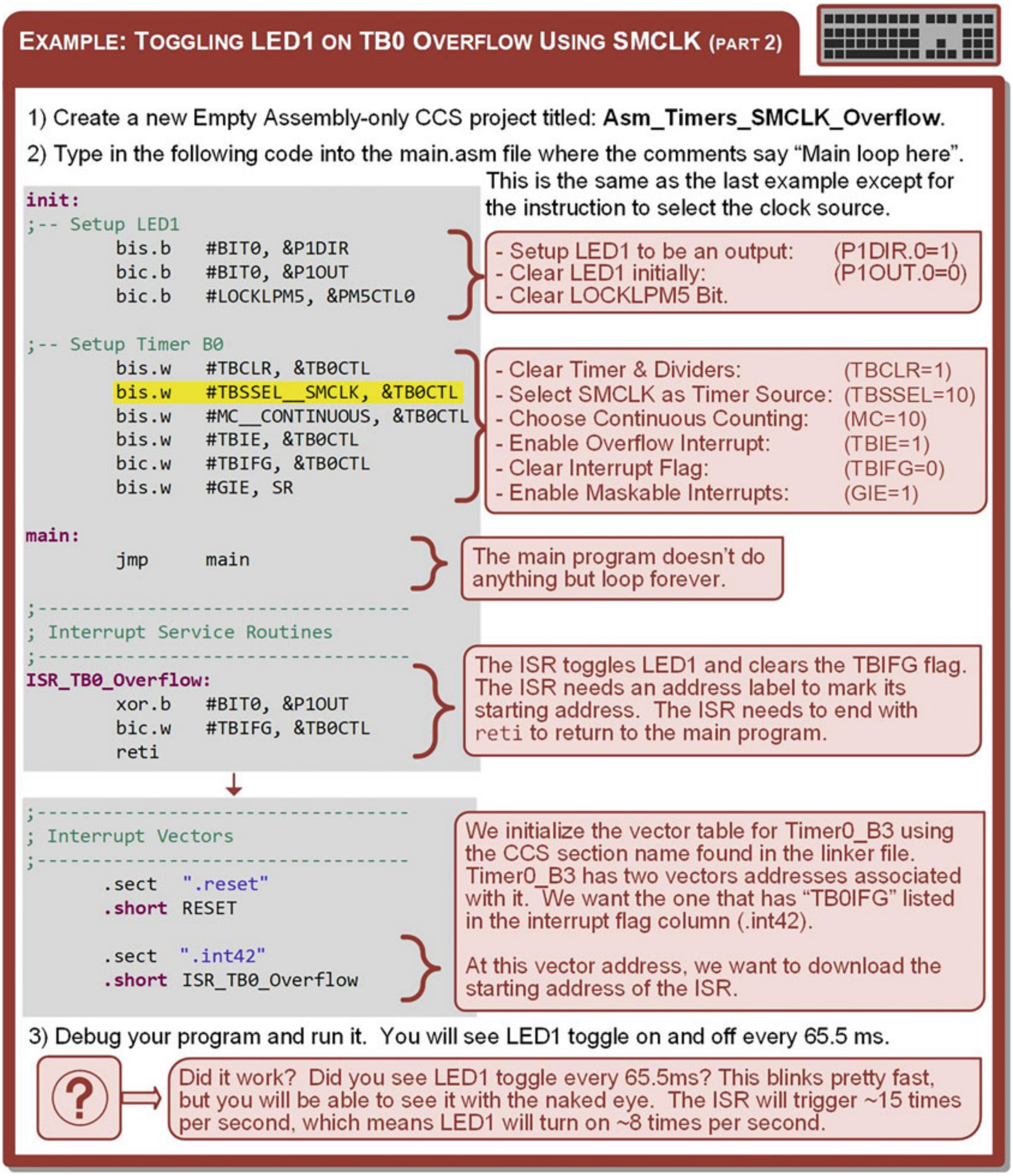
$$T_{overflow} = T\cdot N = (1/f)\cdot 2^{n} = (1/1M)\cdot 2^{16}$$
$$= 65.5\text{ ms (\~15 times per second)}$$



Next, we need to decide on the settings for TB0. We want the clock source to be SMCLK with both dividers set to 1. We also want the counter length to be 16-bits and the counting mode to be continuous. We then need to enable TBIE. We can use the default values for the dividers and the counter length to make our program more compact. The following is our timer setup:



Example 12.9
Toggling LED1 on TB0 overflow using SMCLK (part 1)



Example: Toggling LED1 on TB0 Overflow Using SMCLK (part 2)

1) Create a new Empty Assembly-only CCS project titled: **Asm_Timers_SMCLK_Overflow**.

2) Type in the following code into the main.asm file where the comments say "Main loop here". This is the same as the last example except for the instruction to select the clock source.

```
init:
;-- Setup LED1
        bis.b   #BIT0, &P1DIR
        bic.b   #BIT0, &P1OUT
        bic.b   #LOCKLPM5, &PM5CTL0

;-- Setup Timer B0
        bis.w   #TBCLR, &TB0CTL
        bis.w   #TBSSEL__SMCLK, &TB0CTL
        bis.w   #MC__CONTINUOUS, &TB0CTL
        bis.w   #TBIE, &TB0CTL
        bic.w   #TBIFG, &TB0CTL
        bis.w   #GIE, SR

main:
        jmp     main

;---------------------------------
; Interrupt Service Routines
;---------------------------------
ISR_TB0_Overflow:
        xor.b   #BIT0, &P1OUT
        bic.w   #TBIFG, &TB0CTL
        reti
```

- Setup LED1 to be an output: (P1DIR.0=1)
- Clear LED1 initially: (P1OUT.0=0)
- Clear LOCKLPM5 Bit.

- Clear Timer & Dividers: (TBCLR=1)
- Select SMCLK as Timer Source: (TBSSEL=10)
- Choose Continuous Counting: (MC=10)
- Enable Overflow Interrupt: (TBIE=1)
- Clear Interrupt Flag: (TBIFG=0)
- Enable Maskable Interrupts: (GIE=1)

The main program doesn't do anything but loop forever.

The ISR toggles LED1 and clears the TBIFG flag. The ISR needs an address label to mark its starting address. The ISR needs to end with reti to return to the main program.

```
;---------------------------------
; Interrupt Vectors
;---------------------------------
       .sect   ".reset"
       .short  RESET

       .sect   ".int42"
       .short  ISR_TB0_Overflow
```

We initialize the vector table for Timer0_B3 using the CCS section name found in the linker file. Timer0_B3 has two vectors addresses associated with it. We want the one that has "TB0IFG" listed in the interrupt flag column (.int42).

At this vector address, we want to download the starting address of the ISR.

3) Debug your program and run it. You will see LED1 toggle on and off every 65.5 ms.

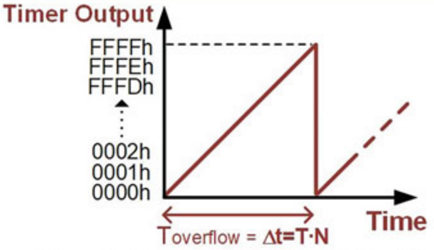
? ⇒ Did it work? Did you see LED1 toggle every 65.5ms? This blinks pretty fast, but you will be able to see it with the naked eye. The ISR will trigger ~15 times per second, which means LED1 will turn on ~8 times per second.

Example 12.10
Toggling LED1 on TB0 overflow using SMCLK (part 2)

Now let's look at an example where we want to use SMCLK as the timer clock, but we want to *slow down the timer overflow period*. We can do this by configuring one of the clock dividers to divide the incoming clock. Let's change the first divider stage to divide SMCLK by 4 (ID = 10), giving a timer clock frequency of 250 kHz. This will result in a timer overflow of 262 ms in the 16-bit configuration. Examples 12.11 and 12.12 provide this example of using the TB0 timer to generate an overflow using SMCLK with a divide-by-4 clock setting.

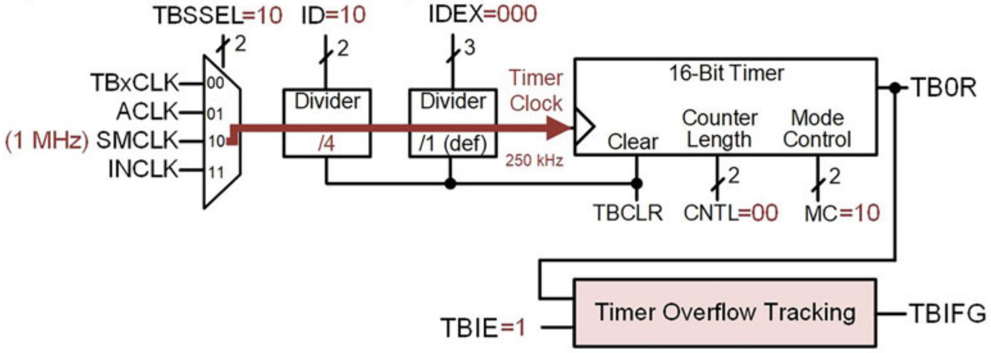
Example: Toggling LED1 on TB0 Overflow Using SMCLK and a Divide-by-4 Clock Configuration (Part 1)

Let's design a program that will toggle LED1 every time TB0 overflows when clocked with **SMCLK**, but in order to **slow down the overflow period**, we'll divide the incoming SMCLK by 4. First, let's calculate the timer overflow period.

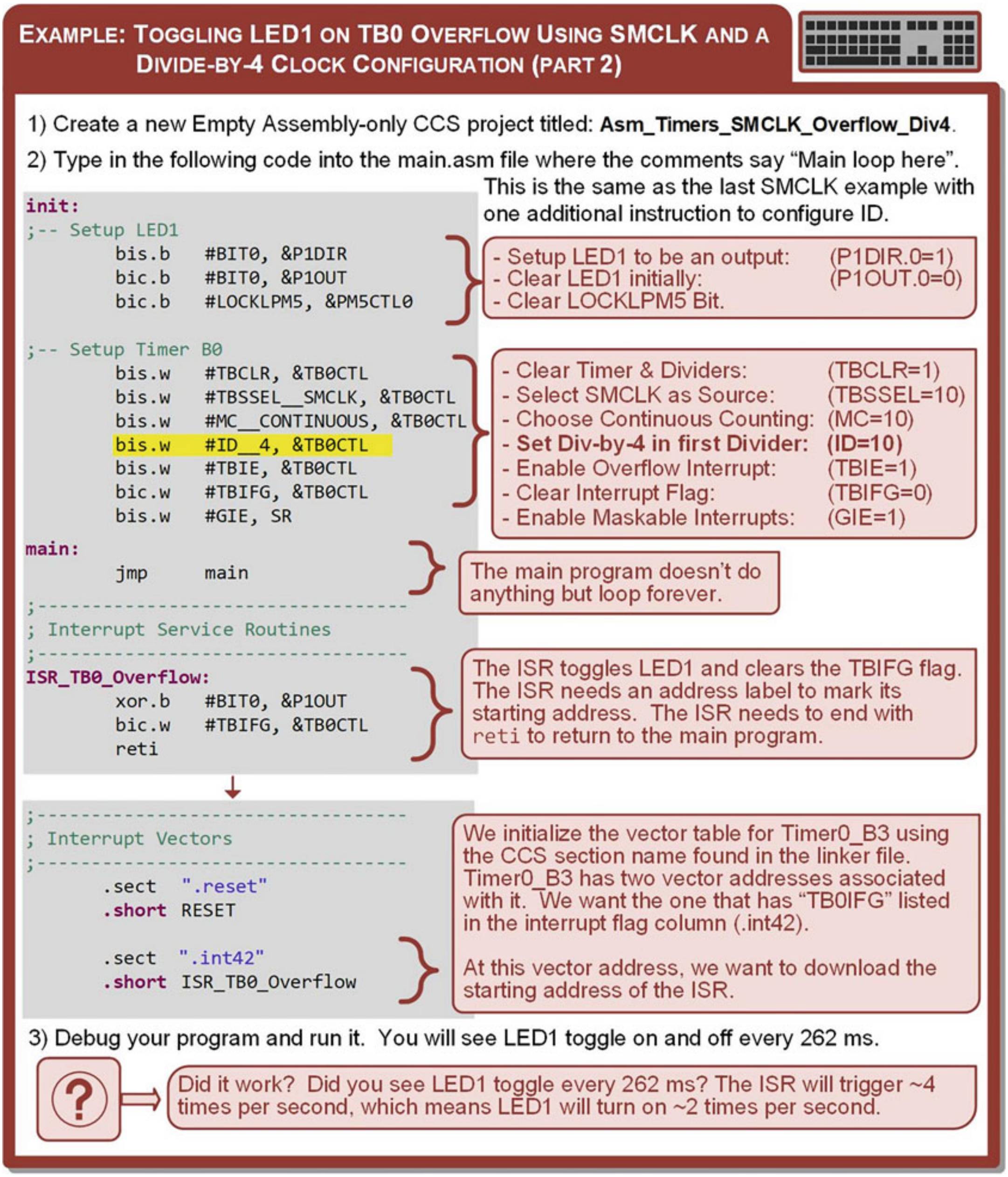


$$T_{overflow} = T\cdot N = (1/(f/4))\cdot 2^{n} = (1/(1M/4))\cdot 2^{16}$$
$$= 262 \text{ ms} \ (\sim 4 \text{ times per second})$$

Next, we need to decide on the settings for TB0. We want the clock source to be SMCLK. We also want to set the **first divider to 4** using the ID bits in TB0CTL. We want the counter length to be 16-bits and the counting mode to be continuous. We then need to enable TBIE. We can use the default values for second divider and the counter length to make our program more compact. The following is our timer setup:



Example 12.11
Toggling LED1 on TB0 overflow using SMCLK and a divide-by-4 clock configuration (part 1)



Example: Toggling LED1 on TB0 Overflow Using SMCLK and a Divide-by-4 Clock Configuration (Part 2)

1) Create a new Empty Assembly-only CCS project titled: **Asm_Timers_SMCLK_Overflow_Div4**.

2) Type in the following code into the main.asm file where the comments say “Main loop here”.

This is the same as the last SMCLK example with one additional instruction to configure ID.

```
init:
;-- Setup LED1
        bis.b   #BIT0, &P1DIR
        bic.b   #BIT0, &P1OUT
        bic.b   #LOCKLPM5, &PM5CTL0

;-- Setup Timer B0
        bis.w   #TBCLR, &TB0CTL
        bis.w   #TBSSEL__SMCLK, &TB0CTL
        bis.w   #MC__CONTINUOUS, &TB0CTL
        bis.w   #ID__4, &TB0CTL
        bis.w   #TBIE, &TB0CTL
        bic.w   #TBIFG, &TB0CTL
        bis.w   #GIE, SR
main:
        jmp     main
;----------------------------------
; Interrupt Service Routines
;----------------------------------
ISR_TB0_Overflow:
        xor.b   #BIT0, &P1OUT
        bic.w   #TBIFG, &TB0CTL
        reti
```

- Setup LED1 to be an output: (P1DIR.0=1)
- Clear LED1 initially: (P1OUT.0=0)
- Clear LOCKLPM5 Bit.

- Clear Timer & Dividers: (TBCLR=1)
- Select SMCLK as Source: (TBSSEL=10)
- Choose Continuous Counting: (MC=10)
- **Set Div-by-4 in first Divider: (ID=10)**
- Enable Overflow Interrupt: (TBIE=1)
- Clear Interrupt Flag: (TBIFG=0)
- Enable Maskable Interrupts: (GIE=1)

The main program doesn’t do anything but loop forever.

The ISR toggles LED1 and clears the TBIFG flag. The ISR needs an address label to mark its starting address. The ISR needs to end with reti to return to the main program.

```
;----------------------------------
; Interrupt Vectors
;----------------------------------
      .sect   ".reset"
      .short  RESET

      .sect   ".int42"
      .short  ISR_TB0_Overflow
```

We initialize the vector table for Timer0_B3 using the CCS section name found in the linker file. Timer0_B3 has two vector addresses associated with it. We want the one that has “TB0IFG” listed in the interrupt flag column (.int42).

At this vector address, we want to download the starting address of the ISR.

3) Debug your program and run it. You will see LED1 toggle on and off every 262 ms.

? ⇨ Did it work? Did you see LED1 toggle every 262 ms? The ISR will trigger ~4 times per second, which means LED1 will turn on ~2 times per second.

Example 12.12
Toggling LED1 on TB0 overflow using SMCLK and a divide-by-4 clock configuration (part 2)

CONCEPT CHECK

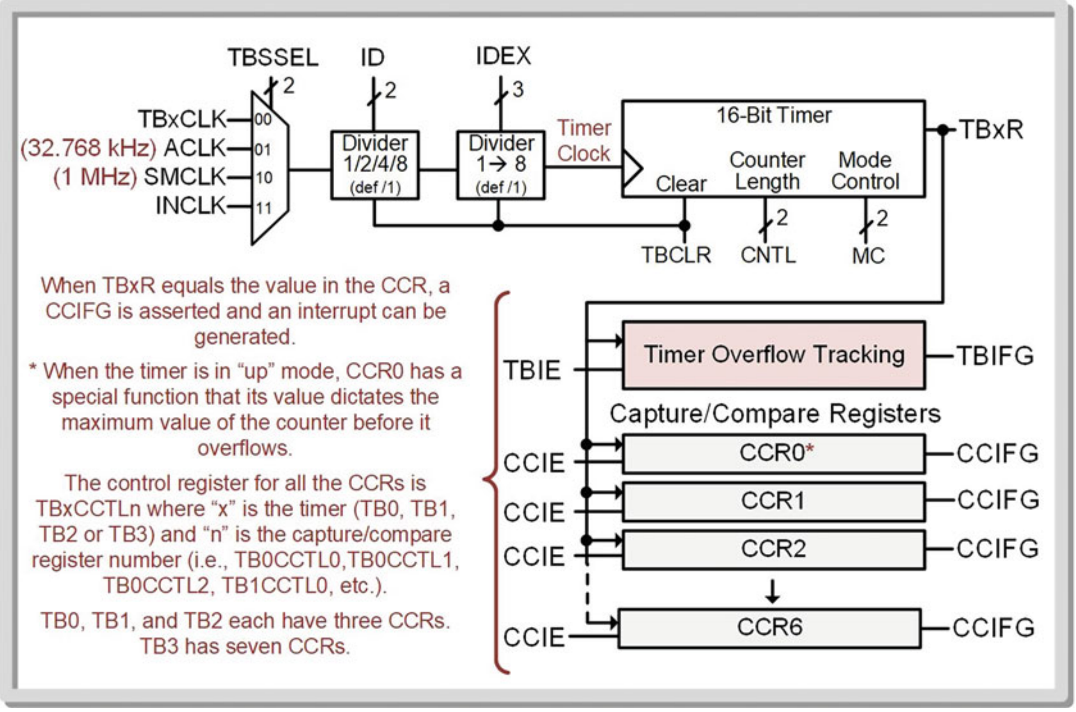
CC12.2 Can we use a 16-bit timer overflow by itself to trigger every 1 second if we clock the system off of 1 MHz?

A) Yes. We simply alter the timer to have a maximum value of 1 million.

B) No. A 16-bit timer overflow will trigger every 65,536 μs. To get exactly 1 second, the clock frequency would need to change, or multiple overflows plus extra counts would need to be used.

12.3 Timer Compares on the MSP430FR2355

A timer *compare* will trigger an event when the main timer value equals a value stored in one of the MSP430's capture/compare registers (CCR). These registers are used for either the compare function or the capture function, which is why they are always referred to as CCRs and not simply compare registers. When the values match, the CCR will assert a flag (CCIFG = capture/compare flag) and can trigger an interrupt if enabled. Each CCR has its own enable (CCIE = capture/compare interrupt enable) and is maskable with the GIE bit. The MSP430FR2355 implements three CCRs for TB0, TB1, and TB2, which is why these timers are often referred to as Timer_B3. The MSP430FR2355 implements seven CCRs for TB3, which is why this timer is often referred to as Timer_B7. The interrupt sources share the two interrupt vectors for each timer. Figure 12.5 shows an updated diagram of the Timer_B system with the capture/compare registers.



When TBxR equals the value in the CCR, a CCIFG is asserted and an interrupt can be generated.

* When the timer is in "up" mode, CCR0 has a special function that its value dictates the maximum value of the counter before it overflows.

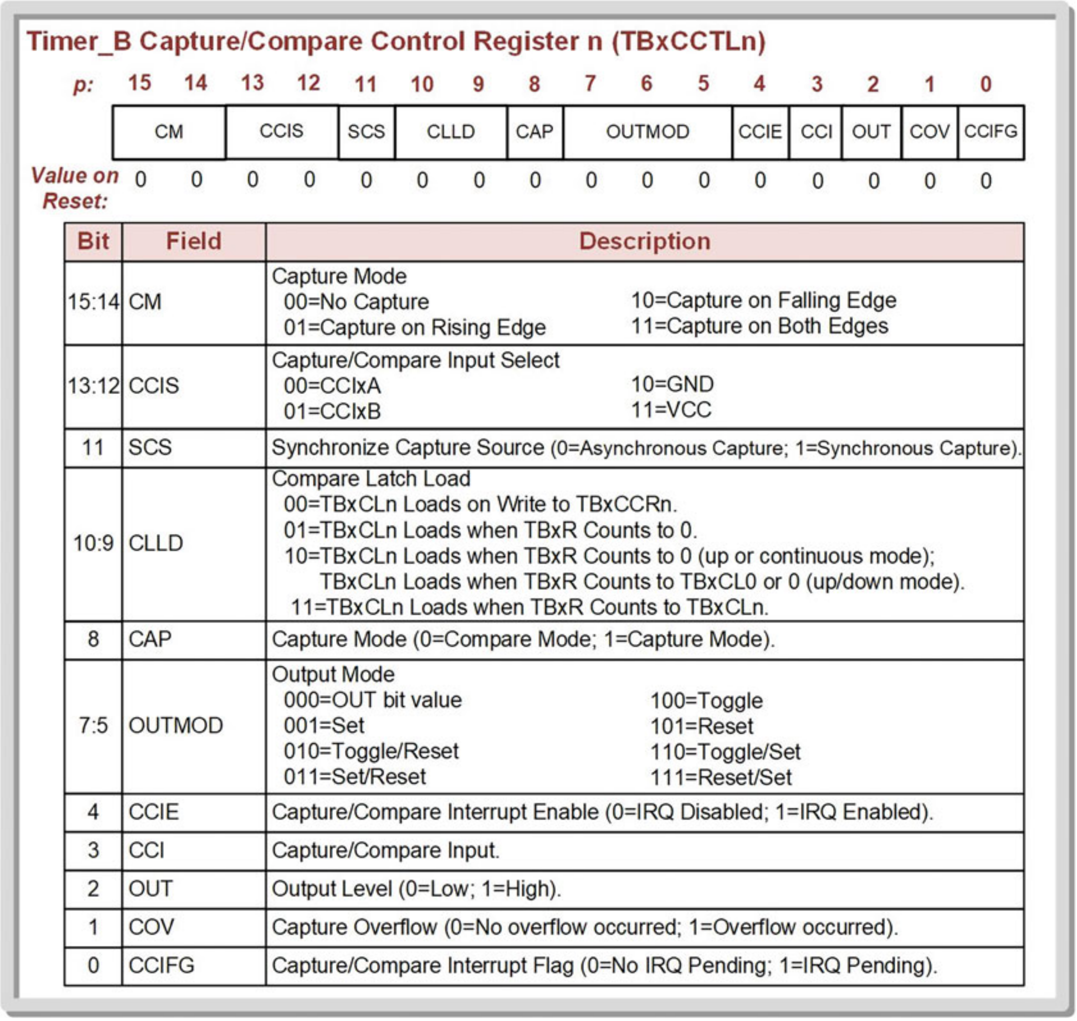
The control register for all the CCRs is TBxCCTLn where "x" is the timer (TB0, TB1, TB2 or TB3) and "n" is the capture/compare register number (i.e., TB0CCTL0,TB0CCTL1, TB0CCTL2, TB1CCTL0, etc.).

TB0, TB1, and TB2 each have three CCRs. TB3 has seven CCRs.

Fig. 12.5
Timer compare overview

Each timer contains two interrupt vector addresses for the capture/compare sources. The MSP430 provides a prioritization scheme where the lower CCR number has the highest priority within the vector (i.e., CCR1 has a higher priority than CCR2). The *Timer_B x Interrupt Vector Register* (TBxIV) holds a code that represents the highest CCR IRQ that has occurred when multiple IRQs are pending at the same time from the capture/compare system.

Each Timer_B CCR register is configured by its own *Timer B Capture/Compare Control Register* (TBxCCTLn). The notation for this register is that "x" stands for the timer (TB0, TB1, TB2, and TB3) and the "n" stands for the CCR number (TBxCCTL0, TBxCCTL1, and TBxCCTL2). Figure 12.6 shows the bit functionality of the TBxCCTLn registers.



Timer_B Capture/Compare Control Register n (TBxCCTLn)

p:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	CM		CCIS		SCS	CLLD		CAP	OUTMOD			CCIE	CCI	OUT	COV	CCIFG
Value on Reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	Field	Description
15:14	CM	Capture Mode 00=No Capture 10=Capture on Falling Edge 01=Capture on Rising Edge 11=Capture on Both Edges
13:12	CCIS	Capture/Compare Input Select 00=CCIxA 10=GND 01=CCIxB 11=VCC
11	SCS	Synchronize Capture Source (0=Asynchronous Capture; 1=Synchronous Capture).
10:9	CLLD	Compare Latch Load 00=TBxCLn Loads on Write to TBxCCRn. 01=TBxCLn Loads when TBxR Counts to 0. 10=TBxCLn Loads when TBxR Counts to 0 (up or continuous mode); TBxCLn Loads when TBxR Counts to TBxCL0 or 0 (up/down mode). 11=TBxCLn Loads when TBxR Counts to TBxCLn.
8	CAP	Capture Mode (0=Compare Mode; 1=Capture Mode).
7:5	OUTMOD	Output Mode 000=OUT bit value 100=Toggle 001=Set 101=Reset 010=Toggle/Reset 110=Toggle/Set 011=Set/Reset 111=Reset/Set
4	CCIE	Capture/Compare Interrupt Enable (0=IRQ Disabled; 1=IRQ Enabled).
3	CCI	Capture/Compare Input.
2	OUT	Output Level (0=Low; 1=High).
1	COV	Capture Overflow (0=No overflow occurred; 1=Overflow occurred).
0	CCIFG	Capture/Compare Interrupt Flag (0=No IRQ Pending; 1=IRQ Pending).

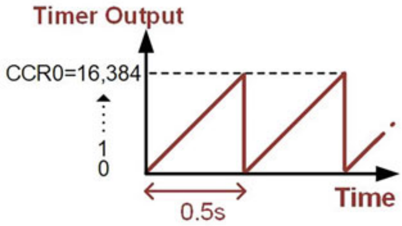
Fig. 12.6
Timer_B capture/compare control register (TBxCCTLn) details

CCR0 has special functionality when the timer is put into “up” mode in that it dictates the maximum value that the timer will count to before overflowing and continue counting at 0. This allows the overflow period of the timer to be configured with a much finer resolution than when using the standard timer overflow (TBIFG). A CCR0 overflow can set the overflow period within one period of the timer clock compared to TBIFG, which only has an overflow resolution of 2^n.

Let’s look at an example of using a timer compare to generate an event every 0.5 seconds. We will use ACLK as the timer source without any division. We need to put the timer into “up” mode to enable the compare functionality for CCR0. We then need to load CCR0 with the compare value that we want to use as the maximum value of the timer before it overflows and starts counting at 0. This is found using the $\Delta t = T \cdot N$ equation where, in this instance, we are solving for N. We plug in $\Delta t = 0.5$ s as the overflow period we are trying to achieve. We plug in $T = 1/(32{,}768$ kHz) since we are using ACLK without any dividers. Finally, we solve for N and get 16,384. This is the value we will put into CCR0 in order to set the maximum value for the timer. Once the timer reaches this value and overflows back to 0, CCIFG will be asserted. To enable an interrupt to be triggered by the CCIFG, we set CCIE and GIE. The interrupt service routine will simply toggle LED1 and clear CCIFG each time it is executed. The vector address we need to use is the one with TB0CCR0 CCIFG0 listed in the interrupt flag column for Timer0_B3, which is FFF8h with the literal .int43. Examples 12.13 and 12.14 show how to implement this timer compare design.

Example: Toggling LED1 Using a CCR0 Compare (Part 1)

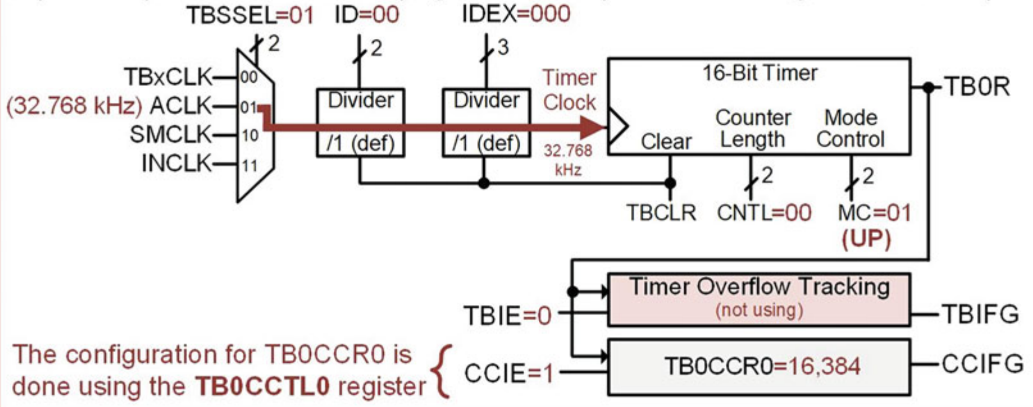
Let’s design a program that will toggle LED1 every 0.5 s using a timer compare. We will use **ACLK** without any dividers. We will use CCR0, which allows us to simply enter a new maximum value for TB0R and then generate an IRQ upon overflow. First, let’s calculate the value to put into CCR0 to achieve an overflow at 0.5 s.



$$\Delta t = T \cdot N$$

$$0.5 \text{ s} = (1/32.768\text{k}) \cdot N \quad \rightarrow N = 16{,}384$$

Next, we need to decide on the settings for TB0. We want the clock source to be ACLK with both dividers set to 1. We also want the counter length to be 16-bits and the counting mode to be **up** so that CCR0 is used as the maximum TB0R value. We then need to load CCR0 with 16,384 and assert CCIE. We can use the default values for the dividers, counter length, capture/compare mode to make our program more compact. The following is our timer setup:



The configuration for TB0CCR0 is done using the **TB0CCTL0** register

Example 12.13
Toggling LED1 using a CCR0 compare (part 1)

Example: Toggling LED1 Using a CCR0 Compare (part 2)

1) Create a new Empty Assembly-only CCS project titled: **Asm_Timer_Compare_CCR0**.

2) Type in the following code into the main.asm file where the comments say “Main loop here”.

```
init:
;-- Setup LED1
        bis.b   #BIT0, &P1DIR
        bic.b   #BIT0, &P1OUT
        bic.b   #LOCKLPM5, &PM5CTL0
```

- Setup LED1 to be an output: (P1DIR.0=1)
- Clear LED1 initially: (P1OUT.0=0)
- Clear LOCKLPM5 Bit.

```
;-- Setup Timer B0
        bis.w   #TBCLR, &TB0CTL
        bis.w   #TBSSEL__ACLK, &TB0CTL
        bis.w   #MC__UP, &TB0CTL

        mov.w   #16384, &TB0CCR0
        bis.w   #CCIE, &TB0CCTL0
        bic.w   #CCIFG, &TB0CCTL0

        bis.w   #GIE, SR
```

- Clear Timer & Dividers: (TBCLR=1)
- Select ACLK as Timer Source: (TBSSEL=01)
- Choose **UP** Counting: (MC=01)
- Initialize CCR0 to 16,384.
- Enable Capture/Compare IRQ: (CCIE=1)
- Clear Interrupt Flag: (CCIFG=0)
- Enable Maskable Interrupts: (GIE=1)

```
main:
        jmp     main
```

The main program doesn’t do anything but loop forever.

```
;-------------------------------
; Interrupt Service Routines
;-------------------------------
ISR_TB0_CCR0:
        xor.b   #BIT0, &P1OUT
        bic.w   #CCIFG, &TB0CCTL0
        reti
```

The ISR toggles LED1 and clears the CCIFG flag.

```
;-------------------------------
; Interrupt Vectors
;-------------------------------
      .sect   ".reset"
      .short  RESET

      .sect   ".int43"
      .short  ISR_TB0_CCR0
```

We initialize the vector table for Timer0_B3 using the CCS section name found in the linker file. Timer0_B3 has two vector addresses associated with it. We want the one that has “CCIFG0” listed in the interrupt flag column (.int43).

At this vector address, we want to download the starting address of the ISR (i.e., ISR_TB0_CCR0).

3) Debug your program and run it. You will see LED1 toggle on and off every 0.5 seconds.

Did it work? Did you see LED1 toggle every 0.5 seconds? If it didn’t, you should inspect the TB0 configuration settings in the Register Viewer.

Example 12.14
Toggling LED1 using a CCR0 compare (part 2)

CONCEPT CHECK

CC12.3 When would a timer compare produce the same period as an n-bit timer overflow?

A) When the value of the compare register is 2^n.

B) When both run off of the same clock frequency.

C) When both are in the same timer peripheral.

D) When both use interrupts to toggle a port.

12.4 Creating Pulse Width Modulated Signals Using Timer Compares

Timer compares allow multiple events to be generate as the timer increments through its counts. One very popular type of electrical signal that can be generated with timer compares is a *pulse width modulated* (PWM) signal. PWM signals are used for motor control, for dimming LEDs, and to communicate information. A PWM signal is a periodic signal where the amount of time that it is HIGH is called its *On Time*. We define the PWM *Duty Cycle* as the percentage of the period that it is high, which is found by simply dividing the On Time by the PWM Period. When using PWM signals, altering the duty cycle is how control of a separate system is achieved. For example, some motors will spin faster as the duty cycle increases and even reverse directions when the duty cycle drops below a certain value. Figure 12.7 shows the details of a PWM signal highlighting some of the key timing characteristics.

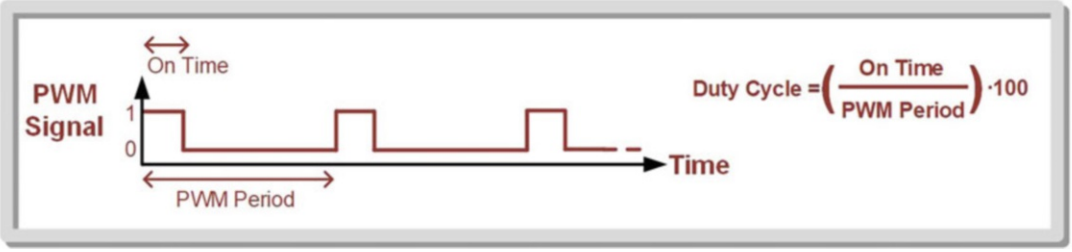


Fig. 12.7
Pulse width modulated (PWM) signal definition

A PWM can be created using the Timer_B system by setting up two compare registers. The first register is CCR0, which will hold the count value corresponding to the period of the PWM signal. The second register, CCR1, will hold the count value corresponding to the amount of on time that is desired. Upon startup, the PWM signal is initialized to a 1. When the first compare occurs on CCR1, the associated ISR will drive the PWM signal to a 0. When the second compare occurs on CCR0, the associated ISR will drive the PWM signal to a 1. This process repeats forever and creates the desired PWM signal with the desired duty cycle. Figure 12.8 shows the theory of how to use two timer compare interrupts to generate a PWM signal.

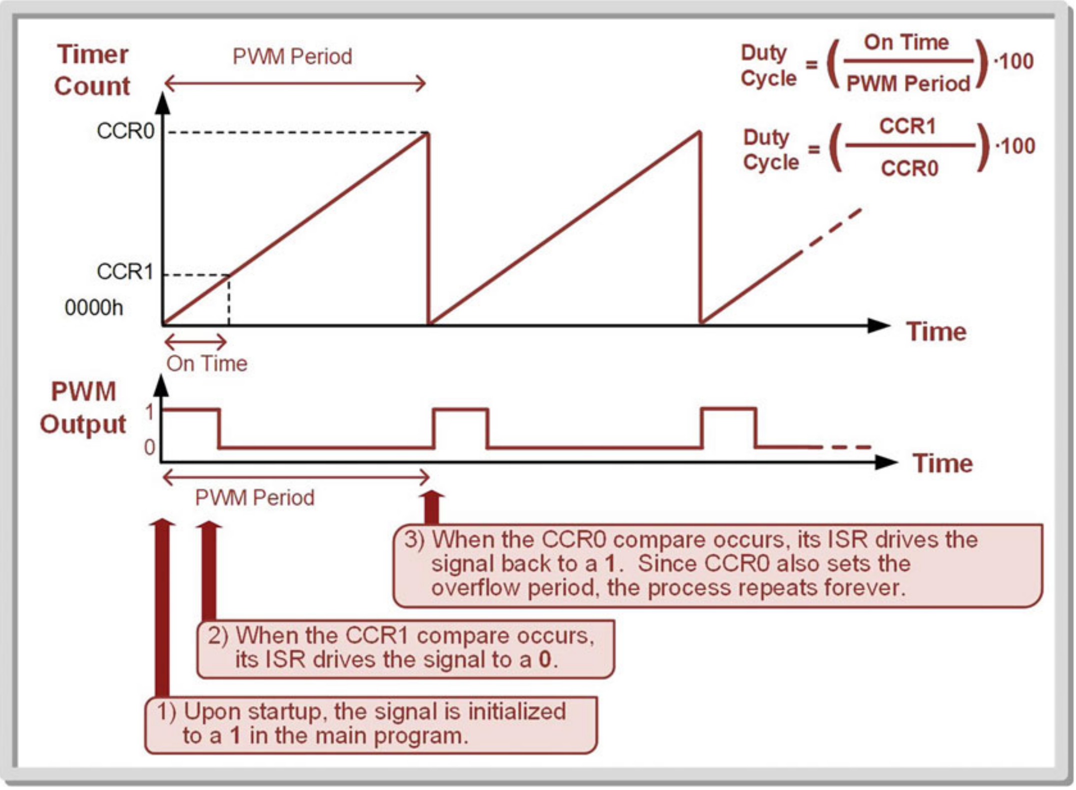

Timer Count
PWM Period
CCR0
CCR1
0000h
Time
On Time
Duty Cycle = (On Time / PWM Period)·100
Duty Cycle = (CCR1 / CCR0)·100
PWM Output
1
0
Time
PWM Period
3) When the CCR0 compare occurs, its ISR drives the signal back to a 1. Since CCR0 also sets the overflow period, the process repeats forever.
2) When the CCR1 compare occurs, its ISR drives the signal to a 0.
1) Upon startup, the signal is initialized to a 1 in the main program.

Fig. 12.8
Creating a PWM signal with multiple timer compares

Let's look at an example of creating a PWM signal on the MSP430FR2355. Let's drive LED1 with a PWM signal that will have a period of 1 second and a duty cycle of 5%. This will result in LED1 coming up for a very short amount of time each period. To accomplish this, let's use TB0 with a clock source of ACLK with no dividers. We will use CCR0 to set the period of the PWM signal to 1 second and CCR1 to set the duty cycle to 5%, or 50 ms. Examples 12.15 and 12.16 show how to implement this program.

Example: Flashing LED1 with a 5% Duty Cycle PWM Using Multiple Compares (Part 1)

Let's design a program that will drive LED1 with a PWM signal with a period of 1 second and a duty cycle of 5%. This will result in a momentary flash on the LED every second. 5% of 1 second is only 50 ms, but it is still long enough to see with the naked eye. We will use ACLK without dividers to create a timer clock of 32.768 kHz. We will use a CCR0 compare to set the period of the PWM to 1 second. We will use a CCR1 compare to set the duty cycle to 5%. We will initialize LED1 to a 1 upon startup. When the first compare on CCR1 occurs after 50 ms, we will drive LED1 to a 0. When the second compare on CCR0 occurs after 1 second, we will drive LED1 back to a 1. Since CCR0 also sets the overflow period, as long as we running in "up" mode, this will repeat forever. First, let's calculate the values to put into CCR0 to achieve a period of 1 second and the value to put into CCR1 to achieve a duty cycle of 5%=50ms..

CCR0

$\Delta t = T \cdot N$

$1\text{ s} = (1/32.768\text{k}) \cdot N$

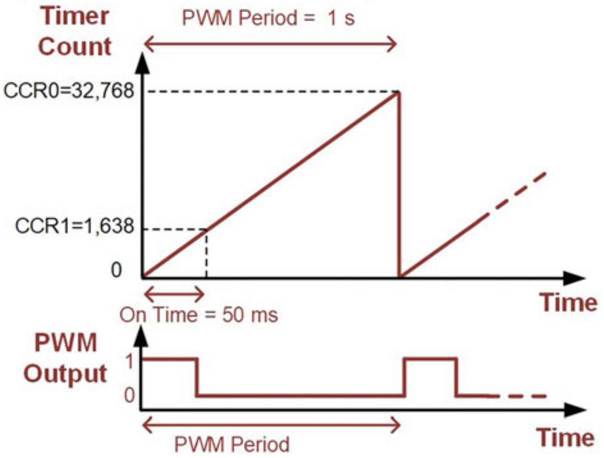
→ **N=32,768**

CCR1

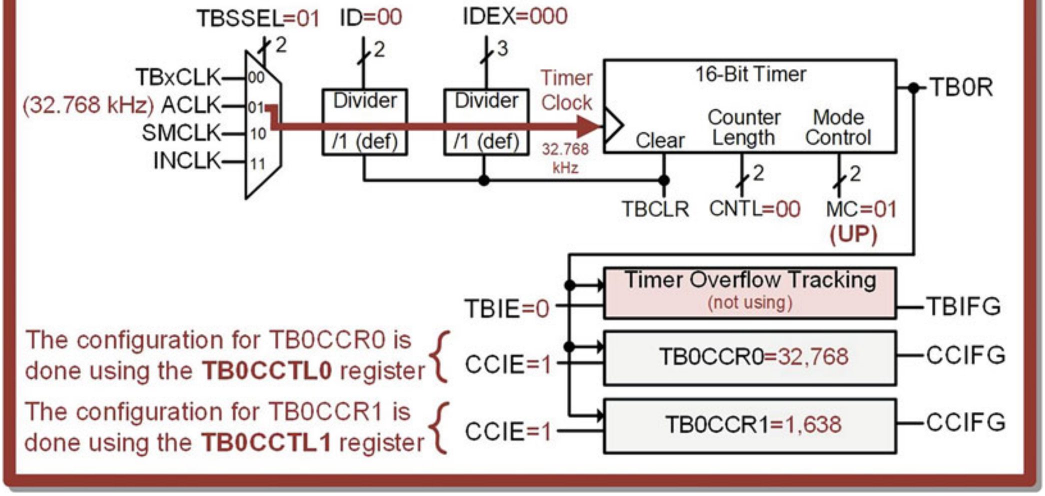
$\Delta t = T \cdot N$

$50\text{m} = (1/32.768\text{k}) \cdot N$

→ **N=1,638**



Next, we need to decide on the settings for TB0. We want the clock source to be ACLK with both dividers set to 1. We also want the counter length to be 16-bits and the counting mode to be **up** so that CCR0 is used as the maximum TB0R value. We then need to load CCR0 with 32,768 and CCR1 with 1,638. We also need to enable the interrupt for CCR0 and the interrupt for CCR1. These each have their own CCIE bit in their respective TB0CCTLn control registers. These two IRQs exist on separate vector addresses, so each can generate its own dedicated interrupt. The following is our timer setup:



The configuration for TB0CCR0 is done using the **TB0CCTL0** register

The configuration for TB0CCR1 is done using the **TB0CCTL1** register

Example 12.15
Flashing LED1 with a 5% duty cycle PWM using multiple compares (part 1)

Example: Flashing LED1 with a 5% Duty Cycle PWM Using Multiple Compares (Part 2)

1) Create a new Empty Assembly-only CCS project titled: **Asm_Timer_Compare_CCR1_n_CCR0_PWM.**

2) Type in the following code into the main.asm file where the comments say “Main loop here”.

```
init:
;-- Setup LED1
        bis.b   #BIT0, &P1DIR
        bis.b   #BIT0, &P1OUT
        bic.b   #LOCKLPM5, &PM5CTL0
```

- Setup LED1 to be an output: (P1DIR.0=1)
- Set LED1 initially: (P1OUT.0=1)
- Clear LOCKLPM5 Bit.

```
;-- Setup Timer B0
        bis.w   #TBCLR, &TB0CTL
        bis.w   #TBSSEL__ACLK, &TB0CTL
        bis.w   #MC__UP, &TB0CTL
```

- Clear Timer & Dividers: (TBCLR=1)
- Select ACLK as Timer Source: (TBSSEL=01)
- Choose **UP** Counting: (MC=01)

```
;-- Setup Compare Registers
        mov.w   #32768, &TB0CCR0
        mov.w   #1638,  &TB0CCR1
```

- Initialize CCR0 to 32,768.
- Initialize CCR1 to 1,638.

```
        bis.w   #CCIE, &TB0CCTL0
        bic.w   #CCIFG, &TB0CCTL0
```

- Enable TB0CCR0 Interrupt.
- Clear TB0CCR0 Flag.

```
        bis.w   #CCIE, &TB0CCTL1
        bic.w   #CCIFG, &TB0CCTL1
```

- Enable TB0CCR1 Interrupt.
- Clear TB0CCR1 Flag.

```
        bis.w   #GIE, SR
```

- Enable Maskable Interrupts.

```
main:
        jmp     main
```

The main loop doesn’t do anything but loop forever.

```
;---------------------------------
; Interrupt Service Routines
;---------------------------------
ISR_TB0_CCR1:
        bic.b   #BIT0, &P1OUT
        bic.w   #CCIFG, &TB0CCTL1
        reti
```

The CCR1 IRQ will trigger first. It needs to drive LED1 to a <u>0</u> and clear the CCR1 flag.

```
ISR_TB0_CCR0:
        bis.b   #BIT0, &P1OUT
        bic.w   #CCIFG, &TB0CCTL0
        reti
```

The CCR0 IRQ will trigger second . It needs to drive LED1 back to a <u>1</u> and clear the CCR0 flag.

```
;---------------------------------
; Interrupt Vectors
;---------------------------------
        .sect   ".reset"
        .short  RESET

        .sect   ".int43"
        .short  ISR_TB0_CCR0
```

The TB0CCR1 CCIFG1 interrupt vector is FFF6h, which uses the literal “.int43”.

```
        .sect   ".int42"
        .short  ISR_TB0_CCR1
```

The TB0CCR0 CCIFG0 interrupt vector is FFF8h, which uses the literal “.int42”.

3) Debug your program and run it. You will see LED1 flash briefly every 1 second.

? ⇒ Did it work? Did you see LED1 flash quickly every 1 second? If it didn’t, you should inspect the TB0 and TB1 configuration settings in the Register Viewer.

Example 12.16
Flashing LED1 with a 5% duty cycle PWM using multiple compares (part 2)

CONCEPT CHECK

CC12.4 What limits the smallest duty cycle time that can be obtained when using a timer to create the signal?

A) The period of the clock because it represents the smallest time amount in a counter (i.e., one count period).

B) The length of the counter.

C) How much you can divide the incoming clock.

D) The number of compare registers available in the timer system.

12.5 Timer Captures on the MSP430FR2355

A timer *capture* will store the current value of the timer into one of the capture/compare registers upon a triggering event. This function can be used to measure time between events, both externally or internally. The capture mode is selected with CAP $=$ 1 in the TBxCCTLn register. A triggering event can be an external signal or an internal system. A transition on the internal signal *Capture/Compare Input* (CCI) will cause a capture. The edge polarity on CCI that triggers the capture is dictated by the *Capture Mode* (CM) bits within TBxCCTLn and supports, rising edge, falling edge, or both edge sensitivity. The source for CCI can come from four different inputs: CCIxA, CCIxB, VCC, or GND. The source for CCI is dictated by the *Capture/Compare Input Select* (CCIS) bits within TBxCCTLn. The CCIxA and CCIxB signals are generic names for inputs that are connected differently on each MCU. For the MSP430FR2355, they are connected to the external timer clock inputs (TBxCLK), the internal comparators, and some of the compare register outputs. The device-specific data sheet lists out the exact connections for each CCIxA and CCIxB signals for each timer.

A capture event can also be triggered by software by manually creating an edge on CCI. This is done using the CCIS bits to manually create an edge using a transition between the VCC and GND inputs to the CCI select multiplexer.

CONCEPT CHECK

CC12.5 What information can you determine about an incoming signal using a timer capture?

A) Its duty cycle.

B) Its period.

C) Its frequency.

D) All of the above.

Summary

- ❖ A timer is a binary counter that runs independently of the CPU. This allows it to track the passage of time without requiring CPU execution cycles.
- ❖ A timer can trigger interrupts when it overflows or when it reaches a certain value held in a compare register.
- ❖ A timer can also store the current count value into a capture register when an event occurs.
- ❖ The time that it takes for the counter to increment by 1 is the period of the clock ($T = 1/f$).
- ❖ The amount of time that has elapsed is found by $\Delta t = T{\cdot}N$ where N is the number of counts.
- ❖ An overflow occurs when the timer goes from its maximum value back to 0. This amount of time is called the timer overflow period and is given by $T_{\text{overflow}} = T{\cdot}2^n$ where n is the number of bits in the counter.
- ❖ When tracking the passage of time between two external events and overflow occurs, the calculation of time must account for each overflow.
- ❖ The MSP430FR2355 contains a Timer_B system with four, 16-bit, independent binary counters.
- ❖ TB0, TB1, and TB2 each have three capture/compare registers, so they are often referred to as Timer_B3 timers. TB3 has seven capture/compare registers so it is often referred to as a Timer_B7 timer.
- ❖ Each of the four timers on the MSP430FR2355 has two dedicated interrupt vector addresses. Each vector address is shared amongst a number of flags for that timer.
- ❖ The Timer_B system on the MSP430FR2355 allows the clock source to be selected among four inputs: TBxCLK, ACLK, SMCLK, and INCLK. TBxCLK and INCLK come from external pins on the MCU. ACLK and SMCLK come from internal oscillators on the MCU.
- ❖ ACLK has a frequency of 32.768 kHz and SMCLK has a frequency of 1 MHz.
- ❖ The clock sources for the Timer_B counters can be divided down to achieve slower counting frequencies.
- ❖ The Timer_B counters can be configured to have different lengths (16-bit, 12-bit, 10-bit, and 8-bit).
- ❖ The Timer_B counters can be configured to count in different modes (halted, up, continuous, or up/down).
- ❖ The setup of the Timer_B counters and the overflow interrupts are configured using the TBxCTL and TBxEX0 registers. All four Timer_Bs (TB0, TB1, TB2, and TB3) have their own configuration registers (TB0CTL, TB1CTL, TB2CTL, TB3CTL, TB0EX0, TB1EX0, TB2EX0, and TB3EX0).
- ❖ A timer compare occurs when the timer value equals the value that was put into a capture/compare register.
- ❖ A timer compare can generate interrupts at timing intervals that have more precision than the standard timer overflow (TBxIFG) because it is based on an individual count value instead of a rollover at 2^n counts.
- ❖ The CCR0 capture/compare register has a special functionality when the counter is in "up" mode in that it sets the maximum value of the timer before it overflows. This can be used when a timer period is sought that is less than its maximum value of 2^n.
- ❖ Each capture/compare register can generate an interrupt; however, every timer only has two interrupt vectors, so multiple sources share each vector address.
- ❖ The TBxIV register will indicate the highest priority source on a shared interrupt vector.
- ❖ Each CCR is configured by its own Timer_B Capture/Compare Control Register n (TBxCCTLn).
- ❖ Multiple CCR events on the same timer can be used to create pulse width modulated (PWM) signals.
- ❖ A PWM signal has a period and a duty cycle. The period is the amount of time before the signal repeats. The duty cycle is the percentage of time that the signal is "on" (also called the on time).
- ❖ PWM signals are used to control motors, dim LEDs, and communicate information.
- ❖ PWM signals are created on the MSP430 by using a CCR0 event to dictate the period and a CCR1 event to dictate the duty cycle.
- ❖ A timer capture event will store the current value of the timer into a capture/compare register when an event occurs.
- ❖ Timer captures allow the time to be measured between events.
- ❖ A timer capture event is triggered by a transition on the CCI signal. CCI is driven by either an external pin, an internal system, VCC or GND. The internal signal connections for a capture differ between MCUs and even between timers. The specific connections are given in the device-specific data sheet.
- ❖ A capture can be created in software by manually creating an edge on the CCI signals by switching its source between GND and VCC (or vice versa depending on the desired polarity).

Exercise Problems

Section 12.1: Timer Overview

12.1.1 How long does it take to increment a timer's binary counter by 1 if the clock frequency is known? Give the equation.

12.1.2 What is the equation for how much time has elapsed between 0 and a count value assuming no overflow has occurred?

12.1.3 What is the equation for the timer overflow period T_{overflow}?

12.1.4 What is the equation for the difference in time between two values when overflow(s) occur between the two values?

12.1.5 If the timer clock is 500 kHz, what is the period of an individual count?

12.1.6 If the timer clock is 250 kHz, what is the period of an individual count?

12.1.7 If the timer clock is 16.384 kHz, what is the period of an individual count?

12.1.8 If the timer clock is 8.192 kHz, what is the period of an individual count?

12.1.9 If the timer clock is 1 MHz, how much time elapses between the two values 5555h and 9999h if no overflow has occurred?

12.1.10 If the timer clock is 500 kHz, how much time elapses between the two values 5555h and 9999h if no overflow has occurred?

12.1.11 If the timer clock is 250 kHz, how much time elapses between the two values 5555h and 9999h if no overflow has occurred?

12.1.12 If the timer clock is 32.768 kHz, how much time elapses between the two values 5555h and 9999h if no overflow has occurred?

12.1.13 If the timer clock is 16.384 kHz, how much time elapses between the two values 5555h and 9999h if no overflow has occurred?

12.1.14 If the timer clock is 8.192 kHz, how much time elapses between the two values 5555h and 9999h if no overflow has occurred?

12.1.15 If the timer clock is 500 kHz, what is the timer overflow period for a 16-bit counter?

12.1.16 If the timer clock is 500 kHz, what is the timer overflow period for an 8-bit counter?

12.1.17 If the timer clock is 250 kHz, what is the timer overflow period for a 16-bit counter?

12.1.18 If the timer clock is 250 kHz, what is the timer overflow period for an 8-bit counter?

12.1.19 If the timer clock is 16.384 kHz, what is the timer overflow period for a 16-bit counter?

12.1.20 If the timer clock is 16.384 kHz, what is the timer overflow period for an 8-bit counter?

12.1.21 If the timer clock is 8.192 kHz, what is the timer overflow period for a 16-bit counter?

12.1.22 If the timer clock is 8.192 kHz, what is the timer overflow period for an 8-bit counter?

12.1.23 If the timer clock is 500 kHz and the counter length is 16-bits, how much time elapses between Value1 = 7777h and Value2 = 2222h if 3 overflows also occur?

12.1.24 If the timer clock is 250 kHz and the counter length is 16-bits, how much time elapses between Value1 = 7777h and Value2 = 2222h if 3 overflows also occur?

12.1.25 If the timer clock is 16.384 kHz and the counter length is 16-bits, how much time elapses between Value1 = 7777h and Value2 = 2222h if 3 overflows also occur?

12.1.26 If the timer clock is 8.192 kHz and the counter length is 16-bits, how much time elapses between Value1 = 7777h and Value2 = 2222h if 3 overflows also occur?

Section 12.2: Timer Overflows on the MSP430FR2355

12.2.1 If you are using the Timer_B system and want to select ACLK as the clock source, what should you set TBSSEL to?

12.2.2 If you are using the Timer_B system and want to select SMCLK as the clock source, what should you set TBSSEL to?

12.2.3 If you are using the Timer_B system and want to divide the incoming clock by 2 using the first divider stage, what should you set ID to?

12.2.4 If you are using the Timer_B system and want to divide the incoming clock by 8 using the first divider stage, what should you set ID to?

12.2.5 If you are using the Timer_B system and want to divide the incoming clock by 2 using the second divider stage, what should you set IDEX to?

12.2.6 If you are using the Timer_B system and want to divide the incoming clock by 7 using the second divider stage, what should you set IDEX to?

12.2.7 If you are using the Timer_B system and want to configure the counter length to 8-bits, what should you set CNTL to?

12.2.8 If you are using the Timer_B system and want to configure the counter length to 10-bits, what should you set CNTL to?

12.2.9 If you are using the Timer_B system and want to configure the counter mode to "up," what should you set MC to?

12.2.10 If you are using the Timer_B system and want to configure the counter mode to "up/down," what should you set MC to?

12.2.11 What is the timer overflow period if you configure the Timer_B to use ACLK, divided by 4, with a counter length of 8?

12.2.12 What is the timer overflow period if you configure the Timer_B to use ACLK, divided by 8, with a counter length of 16?

12.2.13 What is the timer overflow period if you configure the Timer_B to use SMCLK, divided by 8, with a counter length of 8?

12.2.14 What is the timer overflow period if you configure the Timer_B to use ACLK, divided by 2, with a counter length of 16?

Section 12.3: Timer Compares on the MSP430FR2355

12.3.1 What mode does the counter need to be in to use a CCR0 event to set the maximum timer value before overflow?

12.3.2 What is the name of the configuration register that handles setting up the CCRs?

12.3.3 What is the name of the register that provides a unique code with the highest priority CCR that has occurred when multiple CCIFG have been asserted?

12.3.4 How many capture/compare registers does the TB0 system have?

12.3.5 How many capture/compare registers does the TB3 system have?

12.3.6 If you are using ACLK with no dividers as the timer clock with the timer running in 16-bit up mode, how long does it take for overflow if CCR0 = 1234h?

12.3.7 If you are using ACLK with no dividers as the timer clock with the timer running in 16-bit up mode, how long does it take for overflow if CCR0 = 5555h?

12.3.8 If you are using SMCLK with no dividers as the timer clock with the timer running in 16-bit up mode, how long does it take for overflow if CCR0 = 1234h?

12.3.9 If you are using SMCLK with no dividers as the timer clock with the timer running in 16-bit up mode, how long does it take for overflow if CCR0 = 5555h?

12.3.10 If you are using SMCLK with no dividers as the timer clock with the timer running in 16-bit up mode, how long does it take for overflow if CCR0 = 2000h?

12.3.11 If you are using SMCLK with no dividers as the timer clock with the timer running in 16-bit up mode, how long does it take for overflow if CCR0 = 3000h?

Section 12.4: Creating Pulse Width Modulated Signals Using Timer Compares

12.4.1 How long is a PWM signal HIGH for if the period is 500 ms and the duty cycle is 10%?

12.4.2 How long is a PWM signal LOW for if the period is 500 ms and the duty cycle is 10%?

12.4.3 How long is a PWM signal HIGH for if the period is 750 ms and the duty cycle is 10%?

12.4.4 How long is a PWM signal LOW for if the period is 750 ms and the duty cycle is 10%?

12.4.5 You are going to generate a PWM signal with a period of 700 ms and a duty cycle of 10%. You are using ACLK with no dividers as the timer clock with the timer running in 16-bit up mode. What value should you put into CCR0 to set the PWM *period*?

12.4.6 You are going to generate a PWM signal with a period of 700 ms and a duty cycle of 10%. You are using ACLK with no dividers as the timer clock with the timer running in 16-bit up mode. What value should you put into CCR1 to set the PWM *duty cycle*?

12.4.7 You are going to generate a PWM signal with a period of 200 μs and a duty cycle of 5%. You are using SMCLK with no dividers as the timer clock with the timer running in 16-bit up mode. What value should you put into CCR0 to set the PWM *period*?

12.4.8 You are going to generate a PWM signal with a period of 200 μs and a duty cycle of 5%. You are using SMCLK with no dividers as the timer clock with the timer running in 16-bit up mode. What value should you put into CCR1 to set the PWM *duty cycle*?

12.4.9 You are going to generate a PWM signal with a period of 500 μs and a duty cycle of 10%. You are using SMCLK with no dividers as the timer clock with the timer running in 16-bit up mode. What value should you put into CCR0 to set the PWM *period*?

12.4.10 You are going to generate a PWM signal with a period of 500 μs and a duty cycle of 10%. You are using SMCLK with no dividers as the timer clock with the timer running in 16-bit up mode. What value should you put into CCR1 to set the PWM *duty cycle*?

Section 12.5: Timer Captures on the MSP430FR2355

12.5.1 What bit in the TBxCCTLn register is used to put the CCR system into capture mode?

12.5.2 What bits in the TBxCCTLn register are used to control the edge polarity that triggers a capture?

12.5.3 What bits in the TBxCCTLn register are used to select the source for the capture triggering signal CCI?

12.5.4 How can a capture event be generated by software?

12.5.5 If the source for the capture triggering signal CCI differ between MCU and even between timers, how can I find the specific connections for a device?

Chapter 13: Switching to the C Language

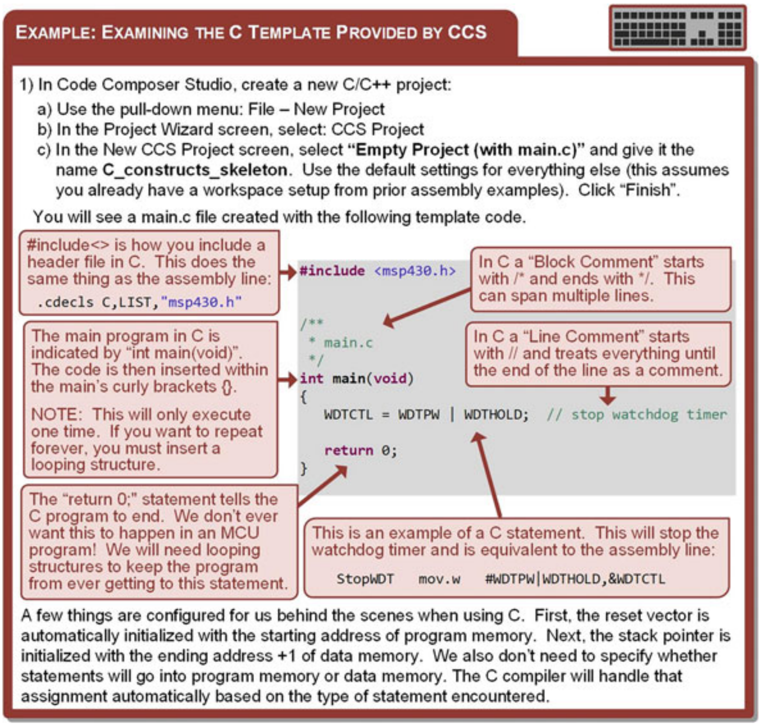
This chapter starts looking at programming the MSP430FR2355 in C [1–3,11]. C is one of the most common languages to program embedded computers. This chapter assumes that the reader has had some exposure to programming using a higher-level language, so not every construct of the C language will be covered; instead, only the functionality of the MSP430FR2355 that has already been covered will be presented. The goal of this chapter is that after completion, the reader can do everything that has been presented thus far in this book in assembly, but using C.

Learning Outcomes—After completing this chapter, you will be able to:

13.1 Implement basic programming techniques in C.
13.2 Implement programs in C that use the digital I/O system of the MSP430FR2355.
13.3 Implement programs in C that use interrupts on the MSP430FR2355.
13.4 Implement programs in C that use the timer system on the MSP430FR2355.

13.1 Basics of C Programming on the MSP430

Let's start by looking at the template of a C program on the MSP430FR2355. CCS automatically provides a main.c template for new projects. Follow Example 13.1 to start a new C project.



EXAMPLE: EXAMINING THE C TEMPLATE PROVIDED BY CCS

1) In Code Composer Studio, create a new C/C++ project:
   a) Use the pull-down menu: File – New Project
   b) In the Project Wizard screen, select: CCS Project
   c) In the New CCS Project screen, select **"Empty Project (with main.c)"** and give it the name **C_constructs_skeleton**. Use the default settings for everything else (this assumes you already have a workspace setup from prior assembly examples). Click "Finish".

You will see a main.c file created with the following template code.

```
#include <msp430.h>

/**
 * main.c
 */
int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer

    return 0;
}
```

#include<> is how you include a header file in C. This does the same thing as the assembly line:

```
.cdecls C,LIST,"msp430.h"
```

In C a "Block Comment" starts with /* and ends with */. This can span multiple lines.

The main program in C is indicated by "int main(void)". The code is then inserted within the main's curly brackets {}.

NOTE: This will only execute one time. If you want to repeat forever, you must insert a looping structure.

In C a "Line Comment" starts with // and treats everything until the end of the line as a comment.

The "return 0;" statement tells the C program to end. We don't ever want this to happen in an MCU program! We will need looping structures to keep the program from ever getting to this statement.

This is an example of a C statement. This will stop the watchdog timer and is equivalent to the assembly line:

```
StopWDT    mov.w    #WDTPW|WDTHOLD,&WDTCTL
```

A few things are configured for us behind the scenes when using C. First, the reset vector is automatically initialized with the starting address of program memory. Next, the stack pointer is initialized with the ending address +1 of data memory. We also don't need to specify whether statements will go into program memory or data memory. The C compiler will handle that assignment automatically based on the type of statement encountered.

Example 13.1
Examining the CCS C template provided by CCS

B. J. LaMeres, *Embedded Systems Design using the MSP430FR2355 LaunchPad™*,
https://doi.org/10.1007/978-3-031-20888-1_13

There are a few important things to note in the template shown in Example 13.1. First, the program file that is created is now called main.c instead of main.asm. Next, the msp430.h header file with all of the bit masks and address name literals is included using the `#include<msp430.h>` statement. This is the standard way to include header files in C. The main program in C is indicated by the `int main (void)` statement with the program residing within curly brackets afterward. This is how the main program is defined in all C files. This can be thought of as a routine call that is named `main` that will return an integer (e.g., the reason for the `int` part of the statement). The `(void)` portion of the routine signifies that there are no variables passed into the routine. The last statement in the main loop is `return 0;`, which means that when it completes, it will return the integer 0 to the calling program. In an embedded program, there is no concept of the programming ending. As such, we never want the return 0 statement to be executed. This means we will always need a looping structure within the main program curly brackets. Also shown is an example statement that disables the watchdog timer.

Example 13.1 also shows the two ways that comments are entered in C. The first is called a *block comment* and begins with `/*` and ends with `*/`. Anything between the starting and ending syntax will be treated as a comment. Block comments can span multiple lines. The second type of comment in C is called a *line comment* and beings with `//`. Anything after the starting syntax to the end of the line will be treated as a comment.

Also, of interest in Example 13.1 is what is not explicitly coded. We do not need to explicitly define the reset vector as it is done automatically for us outside of the main.c functionality. The C linker file contains information to load the starting address of program memory into the reset vector address. Additionally, we do not need to initialize the stack pointer because it is also done for us. The linker file handles initializing the SP register to the last address of data memory +1. We also don't need to define whether lines of code represent information that will go into program memory as opcodes and operands or into data memory as numbers. The C compiler will know whether each line is to be interpreted as an instruction or storage allocation. This gives the compiler the ability to use whatever storage type is most optimal for the program. For example, the standard C syntax to define an integer such as int `i=0` could result in either the use of a CPU register, the stack, or a location in data memory to hold the integer i. When using a high-level language such as C, we give up the lower-level control of the hardware in exchange for abstraction. The abstraction allows our programs to focus more on accomplishing tasks as opposed to interfacing with the registers and memory at the bit level as in assembly.

13.1.1 While() Loops in C

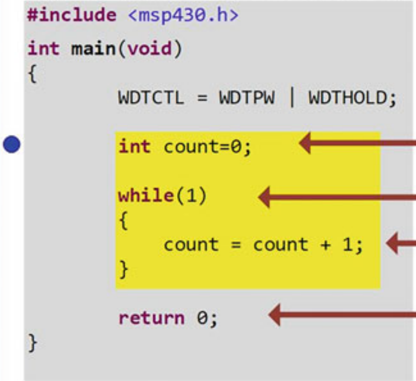
Let's now look at some of the common operations that are used when programming an MCU. First, let's look at a while() loop. A while() loop will execute as long as the Boolean condition provided within its parenthesis is true. A while(1) condition will create an infinite loop because the condition 1 = true. This is how we implement looping forever as we did with the `jmp main` syntax in assembly. Let's look at an example of a while() loop that loops forever while incrementing a variable named *count*. Example 13.2 shows an example of creating this while() loop functionality in C. One important item to note when running C programs on the MSP430 is that the compiler will attempt to optimize the program by default. The C compiler is very efficient at recognizing code that doesn't access the outside of the MCU and can be omitted. For our C examples, we don't want the compiler to optimize out any code because we want to observe the program operation step-by-step as it was written. So, some of the C examples in this book should be compiled with optimization turned *off*. Another interesting note about the code in Example 13.2 is where the variable "count" is stored. The compiler will automatically decide the most efficient place to store a variable. It has the option of storing the variable in a CPU register, in data memory, or on the stack. In Example 13.2, the compiler decided to store count on the stack. This can be determined by looking at the address location of count in the Variable Viewer. As a programmer, we don't need to explicitly define the location of the storage allocation; instead, we just program at a higher-level of abstraction and leave it to the compiler to decide the most efficient location for variable storage.

Example: While() Loops in C

Let's create a while() loop in C that loops forever by using "1" as its Boolean condition. Let's declare a variable called "count" and increment it each time through the loop.

1) In CCS, create a new C/C++ Empty Project (with main.c) titled: **C_While_Loops**.

2) Type in the following code into the main.c file after the statement to stop the watchdog timer.



```
#include <msp430.h>
int main(void)
{
        WDTCTL = WDTPW | WDTHOLD;

        int count=0;

        while(1)
        {
            count = count + 1;
        }

        return 0;
}
```

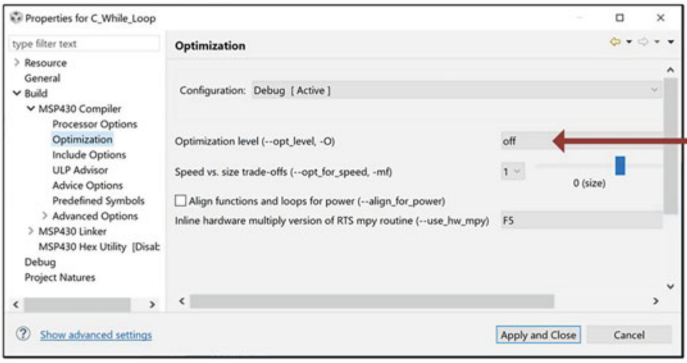
This line will create a variable of type integer called "count" and initialize it to 0.

The while(1) condition will create an infinite loop.

This statement will increment count by one each time through the loop.

Since we use a while(1) infinite loop structure, our program will never reach the return statement and run forever.

3) **TURN OFF OPTIMIZATION!!!** The C compiler is very efficient at removing code from your program that doesn't do anything . If you debug the above program, it will remove the code you entered because it doesn't interface with anything outside of the MCU. For our examples, we don't want anything removed so that we can observe the step-by-step execution of our programs. So we need to turn optimization off using the pull-down menu: Project → Properties. In the dialog that appears, click on "Optimization" and then set the "Optimization level" to "off".



Properties for C_While_Loop

Optimization

Configuration: Debug [Active]

Optimization level (--opt_level, -O) off

Speed vs. size trade-offs (--opt_for_speed, -mf) 1 0 (size)

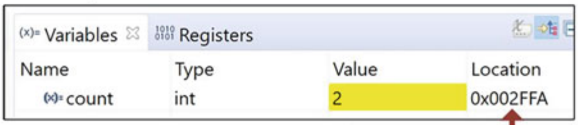
□ Align functions and loops for power (--align_for_power)

Inline hardware multiply version of RTS mpy routine (--use_hw_mpy) F5

Set to "off"

4) Save and debug your program. Set a breakpoint before the `int count=0;` line. Run to the breakpoint.

5) Click on the "Variables" tab next to the Register Viewer tab. This will allow you to see the values of any variables you declared.



Name	Type	Value	Location
(x)= count	int	2	0x002FFA

Notice the compiler decided to store count on the stack.

6) Now step your program and observe the behavior of the while(1) loop and the value of count incrementing each time through the loop.

? Did it work? Did you see count increment as you stepped through the while(1) loop? If you don't see count in the Variables tab, double-check that you turned off optimization.

Example 13.2
While() loops in C

13.1.2 For() Loops in C

Now let's look at an example of a for() loop in C. A for() loop allows us to easily manage the number of times the loop will execute by using a loop variable. We can specify the loop variable's starting value, its end value, and how to increment/decrement it each time through the loop. The loop variable can be used to perform operations on other variables or as an index for addresses. Example 13.3 shows an example of a for() loop that will execute ten times using a loop variable named *i*. Each time through the loop it will assign the value of i to another variable named *count*. Note that to keep the program from ending, we need to nest the for() loop inside of a while(1) loop.

Example: For() Loops in C

Let's create a for() loop in C that will execute 10 times. Each time through the loop, it will assign its loop variable "i" to another variable called "count".

1) In CCS, create a new C/C++ Empty Project (with main.c) titled: **C_For_Loops**.

2) Type in the following code into the main.c file after the statement to stop the watchdog timer.

```
#include <msp430.h>
int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;

    int i=0;
    int count=0;

    while(1)
    {
        for(i=0; i<10; i=i+1)
        {
            count = i;
        }
    }

    return 0;
}
```

These lines declare two variables of type integer. "i" will be used as the for() loop variable and "count" will be assigned i each time through the loop.

The while(1) loop will execute forever and keep the MCU running.

The for() loop will execute 10 times as i is incremented from 0 to 9. Once it reaches 10, it will exit. Each time through the loop, it will assign i to count.

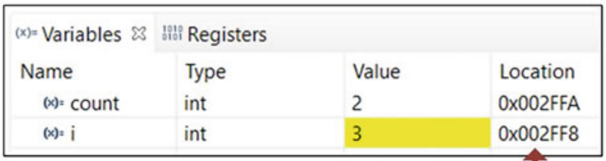
Once the for() loop completes, the while(1) loop will execute again, thus continually running the for() loop. Each time the for() loop starts, the loop variable i will be initialized to 0.

3) **TURN OFF OPTIMIZATION!!!** Each time a new project is created, CCS will set the default compiler setting to optimize your program. You need to turn it off each time.

4) Save and debug your program. Set a breakpoint before the `int i=0;` line. Run to the breakpoint.

5) Click on the "Variables" tab and observe i and count.

6) Now step your program and observe the behavior of the for() loop and the values of i and count each time through the loop.



Name	Type	Value	Location
count	int	2	0x002FFA
i	int	3	0x002FF8

Notice the compiler decided to store both variables on the stack.

Did it work? Did you see the for() loop execute 10 times? Were you able to see the loop variable i increment each time through the loop?

Example 13.3
For() loops in C

13.1.3 If/Else Statements in C

Now let’s look at implementing an if/else statement in C. Example 13.4 shows a program that will use an if/else statement to set a variable named “it_is_TWO” when another variable “i” is equal to 2. In order to cycle through different values of i, a for() loop is used.

Example: If/Else Statements in C

Let’s create an if/else statement in C that will check the value of a variable named “i”. If this variable ever equals two (2), the statement will assert another variable named “it_is_TWO”. For any other values of “i”, the variable “it_is_TWO” will be cleared. We will put this statement within a for() loop so that we can automatically cycle through different values of “i”.

1) In CCS, create a new C/C++ Empty Project (with main.c) titled: **C_If_Else**.

2) Type in the following code into the main.c file after the statement to stop the watchdog timer.

```
#include <msp430.h>
int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;

    int i=0;
    int it_is_TWO=0;

    while(1)
    {

        for(i=0; i<5; i=i+1)
        {
            if(i == 2)
              {
                it_is_TWO=1;
              }
            else
              {
                it_is_TWO=0;
              }
        }
    }
    return 0;
}
```

These lines declare two variables of type integer. “i” will be used as the for() loop variable and “is_it_TWO” will be asserted when i=2 using an if/else statement.

The while(1) loop will execute forever and keep the MCU running.

The for() loop will execute 5 times as i is incremented from 0 to 4 so that we can automatically cycle through different values to test the if/else statement.

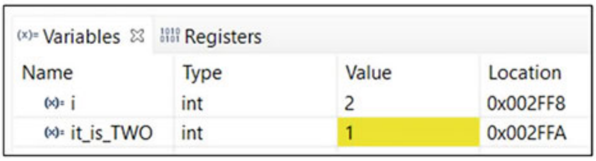
The if/else statement will set it_is_TWO=1 if i=2. Otherwise, it will assign it_is_TWO=0.

3) TURN OFF OPTIMIZATION!!!

4) Save and debug your program. Set a breakpoint before the for() loop. Run to the breakpoint.

5) Step your program and observe “i” and “it_is_TWO” in the Variables Viewer.

Did it work? Did you see the if/else statement make different assignments based on the value of i?



Name	Type	Value	Location
(x)= i	int	2	0x002FF8
(x)= it_is_TWO	int	1	0x002FFA

Example 13.4
If/else statements in C

13.1.4 Switch/Case Statements in C

Now let's look at implementing a switch/case statement in C. Example 13.5 shows a program that will use a switch/case statement to set variables named "it_is_ONE" and "it_is_TWO" when another variable "i" is equal to 1 or 2, respectively. In order to cycle through different values of i, a for() loop is used.

EXAMPLE: SWITCH/CASE STATEMENTS IN C

Let's create a switch/case statement in C that will check the value of a variable named "i". If this variable ever equals one (1), the statement will assert another variable named "it_is_ONE". If this variable ever equals two (2), the statement will assert another variable named "it_is_TWO". For any other values of "i", the variables "it_is_ONE" and "it_is_TWO" will be cleared. We will put this statement within a for() loop so that we can automatically cycle through different values of "i".

1) In CCS, create a new C/C++ Empty Project (with main.c) titled: **C_Switch_Case**.

2) Type in the following code into the main.c file after the statement to stop the watchdog timer.

```
#include <msp430.h>
int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;

    int i=0;
    int it_is_ONE=0;
    int it_is_TWO=0;

    while(1)
    {

      for(i=0; i<5; i=i+1)
      {
        switch(i)
        {
         case 1:  it_is_ONE = 1;
                  it_is_TWO = 0;
                  break;
         case 2:  it_is_ONE = 0;
                  it_is_TWO = 1;
                  break;
         default: it_is_ONE = 0;
                  it_is_TWO = 0;
                  break;
        }
      }
    }

    return 0;
}
```

Variable declarations for "i", "it_is_ONE", and "it_is_TWO".

The while(1) loop will execute forever and keep the MCU running.

The for() loop will increment "i" from 0 to 4 so that we can automatically cycle through different values to test the switch/case statement.

The switch/case statement will assert "it_is_ONE" if i=1;

The switch/case statement will assert "it_is_TWO" if i=2;

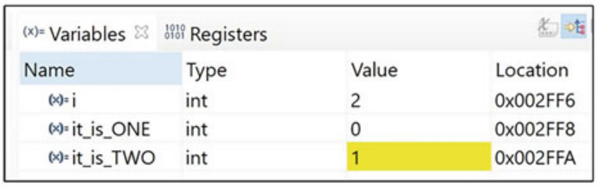
For any other value of "i", the switch/case statement will clear both "it_is_ONE" and "it_is_TWO".

3) **TURN OFF OPTIMIZATION!!!**

4) Save and debug your program. Set a breakpoint before the for() loop. Run to the breakpoint.

5) Step your program and observe "i", "it_is_ONE", and "it_is_TWO" in the Variable Viewer.

Did it work? Did you see the switch/case statement make different assignments based on the value of i?



Name	Type	Value	Location
(x)= i	int	2	0x002FF6
(x)= it_is_ONE	int	0	0x002FF8
(x)= it_is_TWO	int	1	0x002FFA

Example 13.5
Switch/case statements in C

13.1.5 Arithmetic Operators in C

Let's look at some of the basic arithmetic operations provided in C. These are addition (+), subtraction (−), increment (++), and decrement (−−). Follow Example 13.6 to see how these operations work on the MCU.

Example: Arithmetic Operations in C

Let's look at some of the basic arithmetic operations provided in C: addition, subtraction, increment, and decrement. Note that increment and decrement have shorthand syntax that can be optionally used.

1) In CCS, create a new C/C++ Empty Project (with main.c) titled: **C_Arithmetic**.

2) Type in the following code into the main.c file after the statement to stop the watchdog timer.

```
#include <msp430.h>
int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;

    int a = 2;
    int b = 3;
    int c = 4;
    int d = 5;

    while(1)
    {
     b = a + b;
     d = c - d;

     b = b + 1;
     b++;

     d = d - 1;
     d--;
    }

    return 0;
}
```

Variable declaration and initialization.

Addition and subtraction operators.

Increment operations. The b++ is a shorthand way to indicate an increment.

Decrement operations. The d-- is a shorthand way to indicate a decrement.

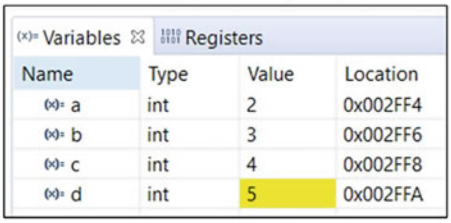
Did it work? Did you get the results that you expected?

3) TURN OFF OPTIMIZATION!!!

4) Save and debug your program. Set a breakpoint before the while() loop. Run to the breakpoint.

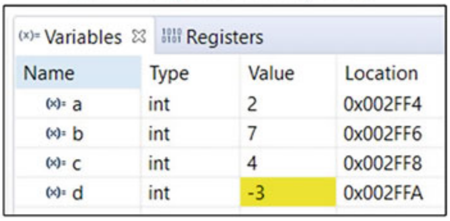
5) Step your program and observe the variables in the Variable Viewer.

The variable values before entering the while() loop.



Name	Type	Value	Location
(x)= a	int	2	0x002FF4
(x)= b	int	3	0x002FF6
(x)= c	int	4	0x002FF8
(x)= d	int	5	0x002FFA

The variable values after the first time through the while() loop.

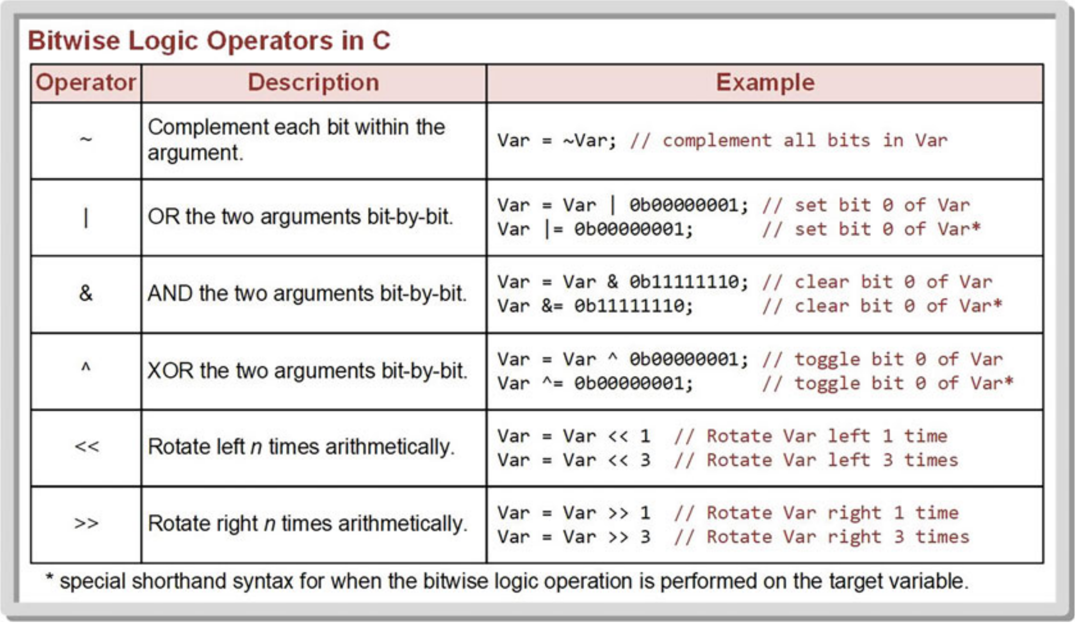


Name	Type	Value	Location
(x)= a	int	2	0x002FF4
(x)= b	int	7	0x002FF6
(x)= c	int	4	0x002FF8
(x)= d	int	-3	0x002FFA

Example 13.6
Arithmetic operations in C

13.1.6 Bitwise Logic Operators in C

Let's look at bit manipulation operations in C. These types of instructions allow us to set, clear, or toggle bits within a variable or register. These operations are critical to setting up the sub-systems on an MCU. In assembly, these were accomplished using the `bis` (bit set) and `bic` (bit clear) instructions. Table 13.1 lists some of the common bit manipulations we will be using in C.



Bitwise Logic Operators in C

Operator	Description	Example
~	Complement each bit within the argument.	`Var = ~Var; // complement all bits in Var`

|	OR the two arguments bit-by-bit.	`Var = Var | 0b00000001; // set bit 0 of Var`
		`Var |= 0b00000001;       // set bit 0 of Var*`

&	AND the two arguments bit-by-bit.	`Var = Var & 0b11111110; // clear bit 0 of Var`
		`Var &= 0b11111110;       // clear bit 0 of Var*`

^	XOR the two arguments bit-by-bit.	`Var = Var ^ 0b00000001; // toggle bit 0 of Var`
		`Var ^= 0b00000001;       // toggle bit 0 of Var*`

<<	Rotate left *n* times arithmetically.	`Var = Var << 1  // Rotate Var left 1 time`
		`Var = Var << 3  // Rotate Var left 3 times`

>>	Rotate right *n* times arithmetically.	`Var = Var >> 1  // Rotate Var right 1 time`
		`Var = Var >> 3  // Rotate Var right 3 times`

* special shorthand syntax for when the bitwise logic operation is performed on the target variable.

Table 13.1
Bitwise logic operators in C

Follow Example 13.7 to see how these operators work on the MSP430FR2355. Also shown in Example 13.7 is how to initialize variables using either binary or hexadecimal format.

Example: Bitwise Logic Operations in C

Let's look at some of the bitwise logic operations provided in C: complement, OR, AND, and XOR. Note that OR, AND, and XOR have shorthand syntax that is commonly used.

1) In CCS, create a new C/C++ Empty Project (with main.c) titled: **C_Bitwise_Logic**.

2) Type in the following code into the main.c file after the statement to stop the watchdog timer.

```c
#include <msp430.h>
int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;

    int e = 0b1111111111110000;
    int f = 0x0001;

    while(1)
    {
     e = ~e;
     e = e | BIT7;
     e = e & ~BIT0;
     e = e ^ BIT4;

     e |= BIT6;
     e &= ~BIT1;
     e ^= BIT3;

     f = f<<1;
     f = f<<2;
     f = f>>1;
    }

    return 0;
}
```

Variable declaration and initialization. Notice that "e" is initialized in binary and "f" is initialized in hexadecimal.

Bitwise logical operations
- complement all bits in e
- set bit 7 using a bitwise OR with 0b10000000
- clear bit 0 using a bitwise AND with 0b11111110 (notice the "~")
- toggle bit 4 using a bitwise XOR with 0b00010000

Bitwise logical operation shorthand
- set bit 6 using a bitwise OR with 0b01000000
- clear bit 1 using a bitwise AND with 0b11111101 (notice the "~")
- toggle bit 3 using a bitwise XOR with 0b00001000

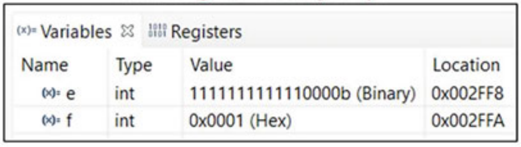
Rotate left and right operations. The arrows give the direction (<< = left, >> = right) and the number indicates how many rotates to do.

3) **TURN OFF OPTIMIZATION!!!**

4) Save and debug your program. Set a breakpoint before the while() loop. Run to the breakpoint.

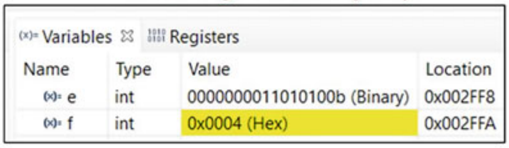
5) Step your program and observe the variables in the Variable Viewer.

The variable values before entering the while() loop.



Name	Type	Value	Location
e	int	1111111111110000b (Binary)	0x002FF8
f	int	0x0001 (Hex)	0x002FFA

The variable values after the first time through the while() loop.



Name	Type	Value	Location
e	int	0000000011010100b (Binary)	0x002FF8
f	int	0x0004 (Hex)	0x002FFA

Did it work? Did you get the results that you expected?

Example 13.7
Bitwise logic operations in C

Concept Check

CC13.1 Does the different bit manipulation syntax supported in C result in different instructions being used?

A) No. The compiler will compile into the same instructions.

B) Yes. We need to be careful using the shorthand syntax because it might leave out a few critical instructions that make the program error prone.

13.2 Digital I/O in C

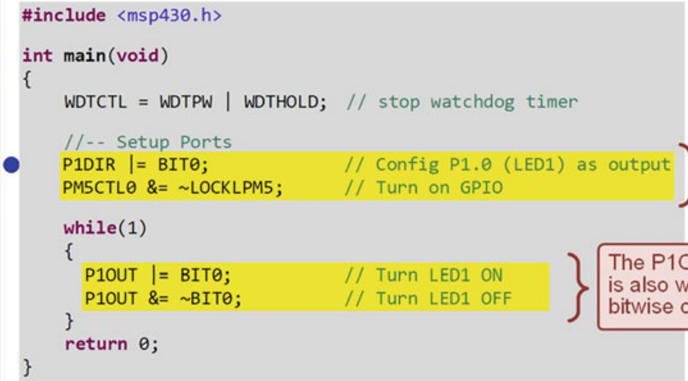
Now let's look at programming the digital I/O system on the MSP430FR2355 in the C language. All of the steps that we had to do in assembly to configure the I/O system still need to be done in C. When using a port as an output, we need to configure its direction to an output (PxDIR = 1) and then disable the I/O low power mode by clearing the LOCKLPM5 bit in PM5CTL0. We use bitwise logic operations to do this configuration. Once the I/O system is enabled, we can simply write logic levels to the port (PxOUT). Follow Example 13.8 to see how to use the digital outputs of the MSP430FR2355. Note that this example is intended to be stepped so that you can observe the execution of each statement.

Example: Using a Digital Output to Drive LED1 in C

Let's write a simple program to turn LED1 on and off as we step the program.

1) In CCS, create a new C/C++ Empty Project (with main.c) titled: **C_Dig_IO_Outputs_n_LEDs**.

2) Type in the following code into the main.c file after the statement to stop the watchdog timer.



```
#include <msp430.h>

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

    //-- Setup Ports
    P1DIR |= BIT0;              // Config P1.0 (LED1) as output
    PM5CTL0 &= ~LOCKLPM5;       // Turn on GPIO

    while(1)
    {
      P1OUT |= BIT0;            // Turn LED1 ON
      P1OUT &= ~BIT0;           // Turn LED1 OFF
    }
    return 0;
}
```

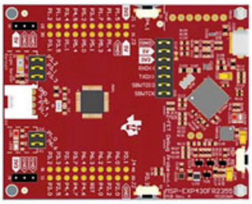
The direction register (P1DIR) and low-power mode are configured using bitwise operations.

The P1OUT register is also written using bitwise operations.

3) Save and debug your program. Set a breakpoint before the statement to set P1DIR. (**P1DIR |= BIT0**).

4) Run your program to the breakpoint. Step your program to observe its operation.

LED1 will turn on and off as you step. If you run continuously the LED will look dim as it turns on and off rapidly.



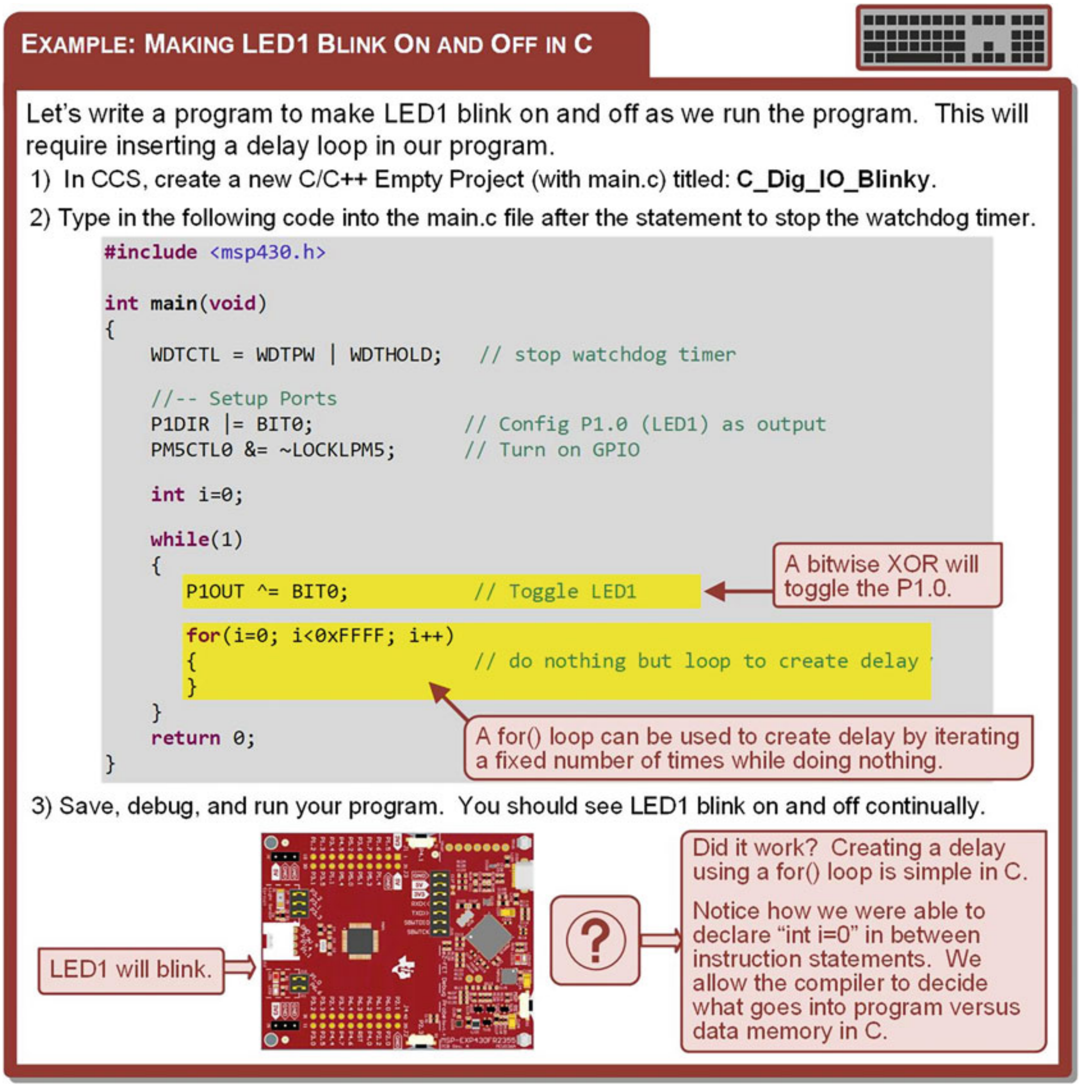
?

Did it work? Notice that the steps to use the digital I/O were the same as what we had to do in assembly.

Notice that we didn't need to turn optimization off this time because we are accessing the outside world.

Example 13.8
Using a digital output to drive LED1 in C

Let's look at how we can incorporate a delay loop in order to slow down how fast an output port is written to. Follow Example 13.9 to see how a for() loop can be used to create delay. This program will blink LED1 on and off continuously about two times per second.



EXAMPLE: MAKING LED1 BLINK ON AND OFF IN C

Let's write a program to make LED1 blink on and off as we run the program. This will require inserting a delay loop in our program.

1) In CCS, create a new C/C++ Empty Project (with main.c) titled: **C_Dig_IO_Blinky**.

2) Type in the following code into the main.c file after the statement to stop the watchdog timer.

```
#include <msp430.h>

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

    //-- Setup Ports
    P1DIR |= BIT0;              // Config P1.0 (LED1) as output
    PM5CTL0 &= ~LOCKLPM5;       // Turn on GPIO

    int i=0;

    while(1)
    {
        P1OUT ^= BIT0;          // Toggle LED1

        for(i=0; i<0xFFFF; i++)
        {                       // do nothing but loop to create delay
        }
    }
    return 0;
}
```

A bitwise XOR will toggle the P1.0.

A for() loop can be used to create delay by iterating a fixed number of times while doing nothing.

3) Save, debug, and run your program. You should see LED1 blink on and off continually.

LED1 will blink.

Did it work? Creating a delay using a for() loop is simple in C.

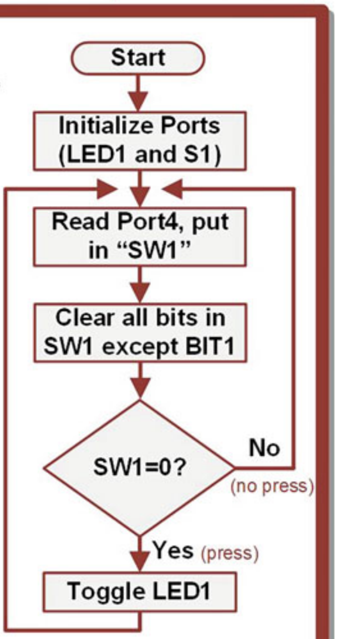
Notice how we were able to declare "int i=0" in between instruction statements. We allow the compiler to decide what goes into program versus data memory in C.

Example 13.9
Making LED1 blink on and off in C

Now let's look at using a port as an input. Again, all of the steps that were covered when using a port input in assembly need to be done in C. These include setting the port direction to an input (PxDIR = 0), enabling a pull-up/down resistor (PxREN = 1) if applicable, and setting the polarity of the resistor (PxOUT). Once configured, the input can be read from (PxIN). Let's examine how we can use a while (1) loop and an if/else statement to continually check the value of an input port. In this example, we will poll S1 on the MSPFR2355 board. Recall that S1 is a HIGH when not pressed and a LOW when pressed. We can read the input by simply making an assignment statement between PxIN to an internal variable (we'll call the internal variable "SW1"). One additional step that must be performed when assigning the port to a variable is to clear the unused bits in the SW1 variable. In the case of using S1 (P4.1), we only care about bit 1. Once we assign P4IN to SW1, we can clear out all other bits in the variable using a bitwise AND with 0b00000010. Once this is done, SW1 can be used to check if the button was pressed. If the button is *not* pressed, then SW1 will equal 0b00000010. If the button *is* pressed, then SW1 will equal 0b00000000. We can then use SW1 in an if/else statement by checking whether it is 0 or *not* 0 (i.e., `if (SW == 0)`). Follow Example 13.10 to see how an input can be polled in C.

EXAMPLE: POLLING THE INPUT S1 WITH DELAY IN C

Let's write a program that will poll S1. When S1 is not pressed (S1=1), LED1 will be off. When it is pressed (S1=0), LED1 will turn on. To make the MCU more responsive to presses, we will insert a delay after a press. The flow chart provided gives the functionality of our program.



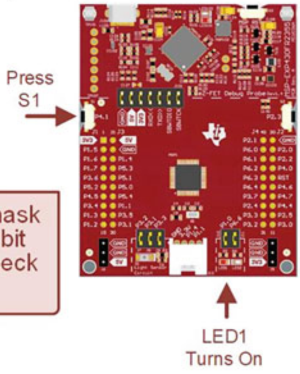
1) In CCS, create a new C/C++ Empty Project (with main.c) titled: **C_Dig_IO_Inputs_n_Polling_S1**.

2) Type in the following code into the main.c file after the statement to stop the watchdog timer.

```c
#include <msp430.h>

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

    //-- Setup Ports
    P1DIR |= BIT0;          // Set P1.0=LED1 direction = out
    P1OUT &= ~BIT0;         // Clear P1.0 (LED1) to start

    P4DIR &= ~BIT1;         // Clear P4.1 (S1) direction = in
    P4REN |= BIT1;          // Enable pull up/down resistor
    P4OUT |= BIT1;          // Make resistor a pull up

    PM5CTL0 &= ~LOCKLPM5; // Turn on GPIO

    int i;
    int SW1;

    while(1)
    {
        SW1 = P4IN;         // Read Port4, put in SW1
        SW1 &= BIT1;        // Clear bits in SW1 except BIT1

        if (SW1 == 0)
        {
            P1OUT ^= BIT0;  // Toggle LED1
        }

        for(i=0; i<10000; i++)
        {                   // Delay loop
        }
    }
    return 0;
}
```

Read Port4, mask all bits except bit 1, and then check its value.

3) Save, debug, and run your program. You should see LED1 turn on when you press S1.



Did it work? If it didn't, view the register settings for Port4 and your variable values for SW1.

Example 13.10
Polling the input S1 with delay in C

CONCEPT CHECK

CC13.2 Which C construct covered used in digital I/O is the closest to a 1-to-1 mapping to an assembly language instruction?

A) A while(1) loop to create an infinite loop.

B) An if/else statement within a while() loop to poll an input.

C) A for() loop to create delay.

D) A bit manipulation statement to set or clear bits in a configuration register.

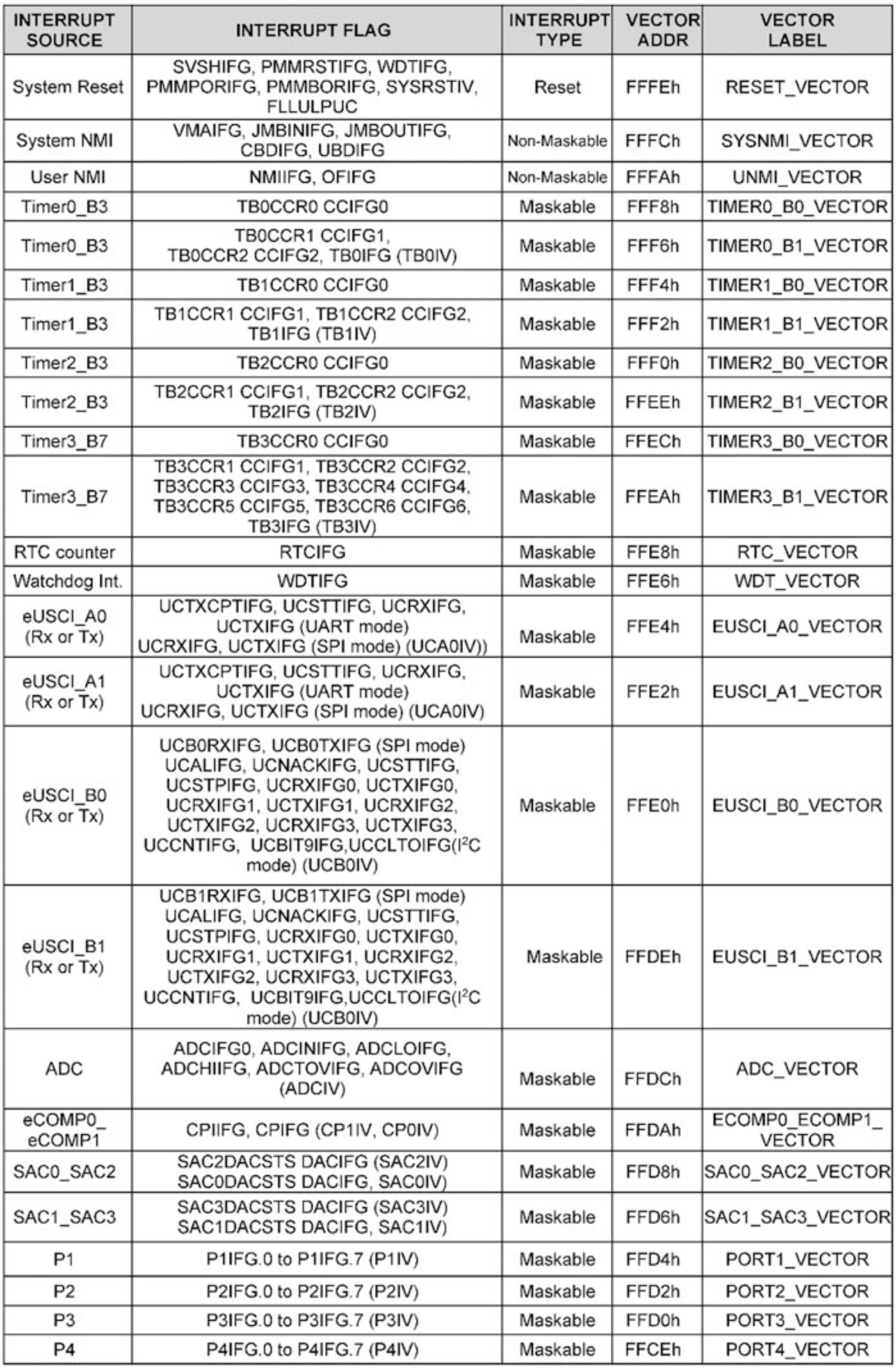
13.3 Interrupts in C

Now let's look at how to program interrupts in C on the MSP430FR2355. We'll start with a port interrupt since that was how we started learning interrupts in assembly. Let's review the steps that we need to take to use a maskable port interrupt:

1. Configure the peripheral for the desired functionality.
2. Clear the peripheral's interrupt flag (PxIFG).
3. Assert the local interrupt enable (PxIE) for the peripheral.
4. Assert the global interrupt enable (GIE) in the status register.
5. Write ISR with an address label to mark starting location and the `reti` instruction to denote its end. Remember that the ISR must clear the peripherals local interrupt flag (PxIFG) so that when the ISR completes, the peripheral doesn't inadvertently trigger another IRQ.
6. Initialize the vector address for the peripheral using the ISR address label and directives.

Let's specifically look at setting up an IRQ on P4.1 since this is connected to S1 on the MSP430FR2355 LaunchPad^{TM} board. Steps 1 through 3 in the above list are accomplished using bitwise logical operations just as was shown in the prior section on digital I/O. For step 1 we need to set the direction of the port to an input (P4DIR), enable the pull-up/down resistor (P4REN), and configure the resistor's polarity (P4OUT). There is one additional statement that is needed when configuring the port peripheral for an interrupt, which is setting the IRQ sensitivity (P4IES). For step 2, we use a bitwise logic operation to clear the interrupt flag (P4IFG). For step 3, we use a bitwise logic operation to enable the IRQ (P4IE). For step 4, the CCS environment provides a unique function to enable maskable interrupts. We can use the function **`__enable_interrupt()`** in our program to set the GIE bit in the status register. This function implements the functionality of `bis.w #GIE, SR`.

For steps 5 and 6, the CCS environment provides a concise way to label our interrupt service routine, initialize the vector table and denote that our routine is an ISR and should be compiled to use `reti`. We use the **#pragma** directive to specify the interrupt vector we are initializing. A #pragma directive allows us to provide additional information to the compiler beyond what is described in our C statements. In CCS we use the syntax *#pragma vector=<VECTOR_LABEL>* to indicate the vector address to initialize. The VECTOR_LABEL in C is different from the labels we used in assembly. Table 13.2 gives the VECTOR_LABELs for the MSP430FR2355 in C that are provided in the CCS. For our P4.1 example, we use the syntax: #pragma vector=PORT4_VECTOR. This directive line tells the compiler that the routine that is listed next in our main.c will be the ISR. It then takes the starting address of this routine and inserts it into the vector table.



INTERRUPT SOURCE	INTERRUPT FLAG	INTERRUPT TYPE	VECTOR ADDR	VECTOR LABEL
System Reset	SVSHIFG, PMMRSTIFG, WDTIFG, PMMPORIFG, PMMBORIFG, SYSRSTIV, FLLULPUC	Reset	FFFEh	RESET_VECTOR
System NMI	VMAIFG, JMBINIFG, JMBOUTIFG, CBDIFG, UBDIFG	Non-Maskable	FFFCh	SYSNMI_VECTOR
User NMI	NMIIFG, OFIFG	Non-Maskable	FFFAh	UNMI_VECTOR
Timer0_B3	TB0CCR0 CCIFG0	Maskable	FFF8h	TIMER0_B0_VECTOR
Timer0_B3	TB0CCR1 CCIFG1, TB0CCR2 CCIFG2, TB0IFG (TB0IV)	Maskable	FFF6h	TIMER0_B1_VECTOR
Timer1_B3	TB1CCR0 CCIFG0	Maskable	FFF4h	TIMER1_B0_VECTOR
Timer1_B3	TB1CCR1 CCIFG1, TB1CCR2 CCIFG2, TB1IFG (TB1IV)	Maskable	FFF2h	TIMER1_B1_VECTOR
Timer2_B3	TB2CCR0 CCIFG0	Maskable	FFF0h	TIMER2_B0_VECTOR
Timer2_B3	TB2CCR1 CCIFG1, TB2CCR2 CCIFG2, TB2IFG (TB2IV)	Maskable	FFEEh	TIMER2_B1_VECTOR
Timer3_B7	TB3CCR0 CCIFG0	Maskable	FFECh	TIMER3_B0_VECTOR
Timer3_B7	TB3CCR1 CCIFG1, TB3CCR2 CCIFG2, TB3CCR3 CCIFG3, TB3CCR4 CCIFG4, TB3CCR5 CCIFG5, TB3CCR6 CCIFG6, TB3IFG (TB3IV)	Maskable	FFEAh	TIMER3_B1_VECTOR
RTC counter	RTCIFG	Maskable	FFE8h	RTC_VECTOR
Watchdog Int.	WDTIFG	Maskable	FFE6h	WDT_VECTOR
eUSCI_A0 (Rx or Tx)	UCTXCPTIFG, UCSTTIFG, UCRXIFG, UCTXIFG (UART mode) UCRXIFG, UCTXIFG (SPI mode) (UCA0IV))	Maskable	FFE4h	EUSCI_A0_VECTOR
eUSCI_A1 (Rx or Tx)	UCTXCPTIFG, UCSTTIFG, UCRXIFG, UCTXIFG (UART mode) UCRXIFG, UCTXIFG (SPI mode) (UCA0IV)	Maskable	FFE2h	EUSCI_A1_VECTOR
eUSCI_B0 (Rx or Tx)	UCB0RXIFG, UCB0TXIFG (SPI mode) UCALIFG, UCNACKIFG, UCSTTIFG, UCSTPIFG, UCRXIFG0, UCTXIFG0, UCRXIFG1, UCTXIFG1, UCRXIFG2, UCTXIFG2, UCRXIFG3, UCTXIFG3, UCCNTIFG, UCBIT9IFG,UCCLTOIFG(I^{2}C mode) (UCB0IV)	Maskable	FFE0h	EUSCI_B0_VECTOR
eUSCI_B1 (Rx or Tx)	UCB1RXIFG, UCB1TXIFG (SPI mode) UCALIFG, UCNACKIFG, UCSTTIFG, UCSTPIFG, UCRXIFG0, UCTXIFG0, UCRXIFG1, UCTXIFG1, UCRXIFG2, UCTXIFG2, UCRXIFG3, UCTXIFG3, UCCNTIFG, UCBIT9IFG,UCCLTOIFG(I^{2}C mode) (UCB0IV)	Maskable	FFDEh	EUSCI_B1_VECTOR
ADC	ADCIFG0, ADCINIFG, ADCLOIFG, ADCHIIFG, ADCTOVIFG, ADCOVIFG (ADCIV)	Maskable	FFDCh	ADC_VECTOR
eCOMP0_ eCOMP1	CPIIFG, CPIFG (CP1IV, CP0IV)	Maskable	FFDAh	ECOMP0_ECOMP1_ VECTOR
SAC0_SAC2	SAC2DACSTS DACIFG (SAC2IV) SAC0DACSTS DACIFG, SAC0IV)	Maskable	FFD8h	SAC0_SAC2_VECTOR
SAC1_SAC3	SAC3DACSTS DACIFG (SAC3IV) SAC1DACSTS DACIFG, SAC1IV)	Maskable	FFD6h	SAC1_SAC3_VECTOR
P1	P1IFG.0 to P1IFG.7 (P1IV)	Maskable	FFD4h	PORT1_VECTOR
P2	P2IFG.0 to P2IFG.7 (P2IV)	Maskable	FFD2h	PORT2_VECTOR
P3	P3IFG.0 to P3IFG.7 (P3IV)	Maskable	FFD0h	PORT3_VECTOR
P4	P4IFG.0 to P4IFG.7 (P4IV)	Maskable	FFCEh	PORT4_VECTOR

Table 13.2
MSP430FR2355 interrupt vector table (for C)

Next, we need to write our ISR. As mentioned in the last paragraph, it is critical that our routine comes immediately after the #pragma vector=<VECTOR_LABEL> directive. CCS provides the function `__interrupt` to denote that the routine is an ISR and not a subroutine. This is critical because our ISR needs to be compiled so that it uses `reti` at the end instead of `ret` so that the appropriate steps are followed to return from the interrupt. Just as in assembly, our service routine needs to clear the IRQ flag each time it executes to avoid continually triggering.

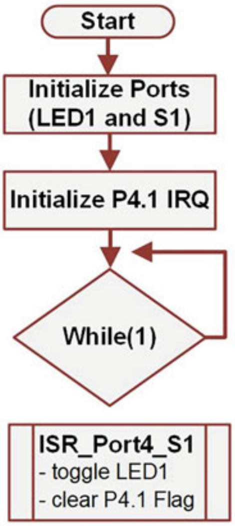
Example 13.11 shows the code to implement a P4.1 interrupt that, when triggered, will toggle LED1. Follow this example to see how to program a port interrupt in C on the MSP430FR2355.

Example: Using a Port Interrupt on S1 to Toggle LED1 in C

Let's design a program that will toggle LED1 each time S1 is pressed using a port interrupt. This approach will be much more efficient than polling.

1) In CCS, create a new C/C++ Empty Project (with main.c) titled: **C_IRQs_Port4_S1**.

2) Type in the following code into the main.c file after the statement to stop the watchdog timer.



```
#include <msp430.h>

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

    //-- Setup Ports
    P1DIR |= BIT0;          // Config P1.0 (LED1) as output
    P1OUT &= ~BIT0;         // Clear P1.0 (LED1) to start

    P4DIR &= ~BIT1;         // Config P4.1 (S1) as input
    P4REN |= BIT1;          // Enable resistor
    P4OUT |= BIT1;          // Make pull up resistor
    P4IES |= BIT1;          // Config IRQ Sensitivity H-to-L

    PM5CTL0 &= ~LOCKLPM5;  // Turn on GPIO

    //-- Setup IRQ
    P4IFG &= ~BIT1;         // Clear Port4.1 IRQ Flag
    P4IE |= BIT1;           // Enable Port4.1 IRQ
    __enable_interrupt();  // Enable Maskable IRQs

     //-- Main Loop
    while(1){}              // Loop forever

    return 0;
}
//-- Interrupt Service Routines --------
#pragma vector = PORT4_VECTOR
__interrupt void ISR_Port4_S1(void)
{
    P1OUT ^= BIT0;
    P4IFG &= ~BIT1;
}
```

Configuring the IRQ sensitivity is done when setting up the port.

Setup and enable IRQ.

The main program just loops forever.

The "#pragma vector" indicates that the function that follows is the ISR for the vector listed and its address should be inserted into the vector table.

The __interrupt keyword specifies that the following function is to be treated as an ISR (i.e., return using `reti`).

3) Save, debug, and run your program.

? Did it work? Does LED1 toggle each time you press S1? If not, check the Port4 configuration registers' values to ensure you setup the ports and IRQ correctly.

Example 13.11
Using a port interrupt on S1 to toggle LED1 in C

Concept Check

CC13.3 Why are the CCS interrupt vector names different between C and assembly?

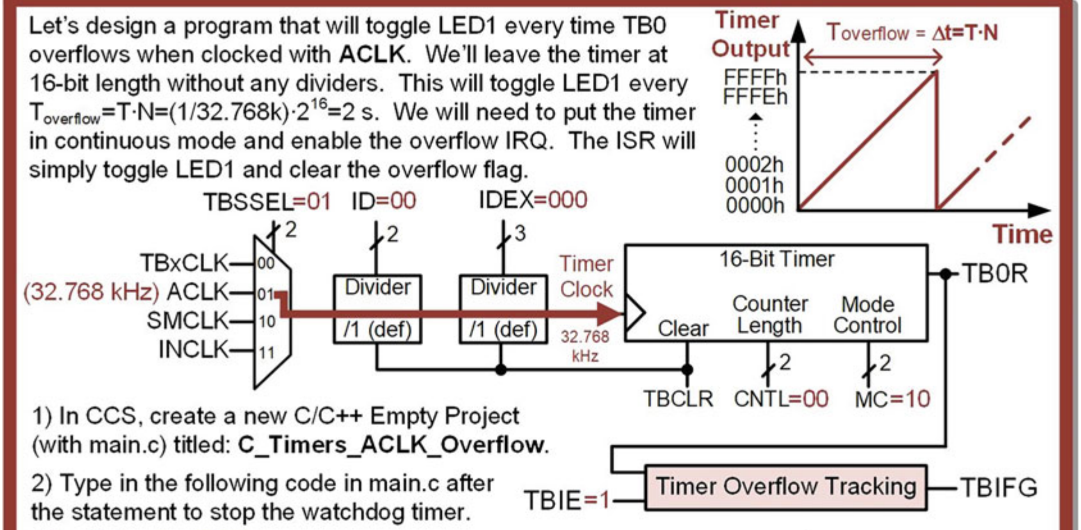
A) I have no idea! My guess is the assembly vector table names were created first. Then later somebody else came up with the names in C that were supposed to make more sense. It was probably too late to change the older names since they are already in use. If you know, please email me!

13.4 Timers in C

Now let's look at how to program the timer system on the MSP430FR2355 in C. Just as in assembly, this involves setting up the timer system, enabling timer IRQs, and then creating an ISR that will accomplish the desired task. Follow Example 13.12 to toggle LED1 every time there is a timer overflow. Example 13.12 uses ACLK (32.768 kHz) as the timer source.

Example: Toggling LED1 on TB0 Overflow Using ACLK in C

Let's design a program that will toggle LED1 every time TB0 overflows when clocked with **ACLK**. We'll leave the timer at 16-bit length without any dividers. This will toggle LED1 every $T_{overflow}=T\cdot N=(1/32.768k)\cdot 2^{16}=2$ s. We will need to put the timer in continuous mode and enable the overflow IRQ. The ISR will simply toggle LED1 and clear the overflow flag.



1) In CCS, create a new C/C++ Empty Project (with main.c) titled: **C_Timers_ACLK_Overflow.**

2) Type in the following code in main.c after the statement to stop the watchdog timer.

```c
#include <msp430.h>

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

    //-- Setup Ports
    P1DIR |= BIT0;               // Config P1.0 (LED1) as output
    P1OUT &= ~BIT0;              // Clear P1.0 (LED1) to start
    PM5CTL0 &= ~LOCKLPM5;        // Turn on GPIO

    //-- Setup Timer
    TB0CTL  |= TBCLR;            // Clear timer and dividers
    TB0CTL  |= TBSSEL__ACLK;     // Source=ACLK
    TB0CTL  |= MC__CONTINUOUS;   // Mode=Continuous

    //-- Setup Timer Overflow IRQ
    TB0CTL &= ~TBIFG;            // Clear TB0 flag
    TB0CTL |= TBIE;              // Enable TB0 Overflow IRQ
    __enable_interrupt();        // Enable Maskable IRQs

     //-- Main Loop
    while(1){}                   // Loop forever

    return 0;
}

//-- Interrupt Service Routines -------------
#pragma vector = TIMER0_B1_VECTOR
__interrupt void ISR_TB0_Overflow(void)
{
    P1OUT ^= BIT0;        // Toggle LED1
    TB0CTL &= ~TBIFG;     // Clear TB0 flag
}
```

Configure Timer

Enable timer overflow interrupt.

The "TIMER0_B1_VECTOR" is the vector for TB0IFG.

3) Debug your program and run it.

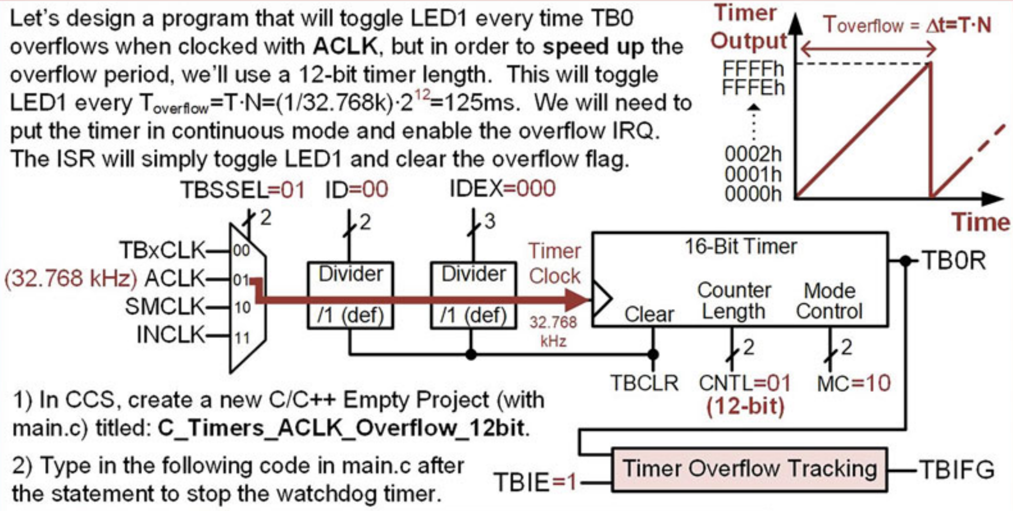
? ⇒ Did it work? Did you see LED1 toggle every 2 seconds? If not, check the settings in the TB0CTL register.

Example 13.12
Toggling LED1 on TB0 overflow using ACLK in C

Next, let's look at how to speed up the overflow period by changing the timer length. Follow Example 13.13 to see how to change the timer length to 12-bits while still using ACLK. This example will again toggle LED1 every time there is a timer overflow.

Example: Toggling LED1 on TB0 Overflow Using ACLK and a 12-bit Counter Configuration in C

Let's design a program that will toggle LED1 every time TB0 overflows when clocked with **ACLK**, but in order to **speed up** the overflow period, we'll use a 12-bit timer length. This will toggle LED1 every $T_{overflow}=T \cdot N=(1/32.768k) \cdot 2^{12}=125ms$. We will need to put the timer in continuous mode and enable the overflow IRQ. The ISR will simply toggle LED1 and clear the overflow flag.



1) In CCS, create a new C/C++ Empty Project (with main.c) titled: **C_Timers_ACLK_Overflow_12bit.**

2) Type in the following code in main.c after the statement to stop the watchdog timer.

```
#include <msp430.h>

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

    //-- Setup Ports
    P1DIR |= BIT0;                // Config P1.0 (LED1) as output
    P1OUT &= ~BIT0;               // Clear P1.0 (LED1) to start
    PM5CTL0 &= ~LOCKLPM5;         // Turn on GPIO

    //-- Setup Timer
    TB0CTL |= TBCLR;              // Clear timer and dividers
    TB0CTL |= TBSSEL__ACLK;       // Source=ACLK
    TB0CTL |= MC__CONTINUOUS;     // Mode=Continuous
    TB0CTL |= CNTL_1;             // Length=12-bit

    //-- Setup Timer Overflow IRQ
    TB0CTL &= ~TBIFG;             // Clear TB0 flag
    TB0CTL |= TBIE;               // Enable TB0 Overflow IRQ
    __enable_interrupt();         // Enable Maskable IRQs

     //-- Main Loop
    while(1){}                    // Loop forever

    return 0;
}

//-- Interrupt Service Routines --------------------
#pragma vector = TIMER0_B1_VECTOR
__interrupt void ISR_TB0_Overflow(void)
{
    P1OUT ^= BIT0;        // Toggle LED1
    TB0CTL &= ~TBIFG;     // Clear TB0 flag
}
```

12-bit length is set by inserting one additional statement.

3) Debug your program and run it.

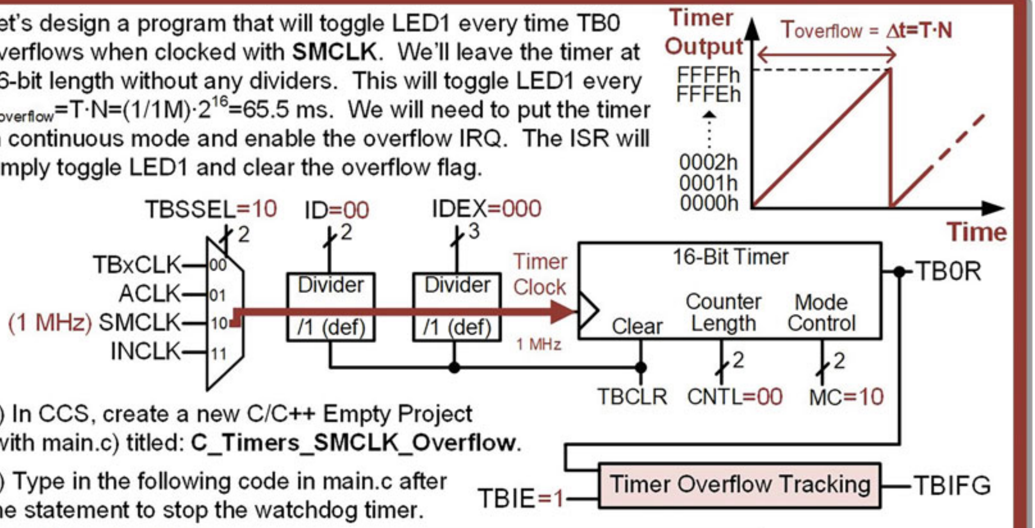
Did it work? Did you see LED1 toggle every 125ms? This will turn on/off 4 times per second.

Example 13.13
Toggling LED1 on TB0 overflow using ACLK and a 12-bit counter configuration in C

Next, let's look at how to change the timer clock source. Follow Example 13.14 to see how to change the timer clock to SMCLK (1 MHz) and toggle LED1 every time there is an overflow.

Example: Toggling LED1 on TB0 Overflow Using SMCLK in C

Let's design a program that will toggle LED1 every time TB0 overflows when clocked with **SMCLK**. We'll leave the timer at 16-bit length without any dividers. This will toggle LED1 every $T_{overflow}$=T·N=(1/1M)·2^{16}=65.5 ms. We will need to put the timer in continuous mode and enable the overflow IRQ. The ISR will simply toggle LED1 and clear the overflow flag.



1) In CCS, create a new C/C++ Empty Project (with main.c) titled: **C_Timers_SMCLK_Overflow.**

2) Type in the following code in main.c after the statement to stop the watchdog timer.

```
#include <msp430.h>

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

    //-- Setup Ports
    P1DIR |= BIT0;                // Config P1.0 (LED1) as output
    P1OUT &= ~BIT0;               // Clear P1.0 (LED1) to start
    PM5CTL0 &= ~LOCKLPM5;         // Turn on GPIO

    //-- Setup Timer
    TB0CTL |= TBCLR;              // Clear timer and dividers
    TB0CTL |= TBSSEL__SMCLK;      // Source=SMCLK
    TB0CTL |= MC__CONTINUOUS;     // Mode=Continuous

    //-- Setup Timer Overflow IRQ
    TB0CTL &= ~TBIFG;             // Clear TB0 flag
    TB0CTL |= TBIE;               // Enable TB0 Overflow IRQ
    __enable_interrupt();         // Enable Maskable IRQs

     //-- Main Loop
    while(1){}                    // Loop forever

    return 0;
}

//-- Interrupt Service Routines --------------------
#pragma vector = TIMER0_B1_VECTOR
__interrupt void ISR_TB0_Overflow(void)
{
    P1OUT ^= BIT0;        // Toggle LED1
    TB0CTL &= ~TBIFG;     // Clear TB0 flag
}
```

SMCLK can be selected by setting TBSSEL=10.

3) Debug your program and run it.

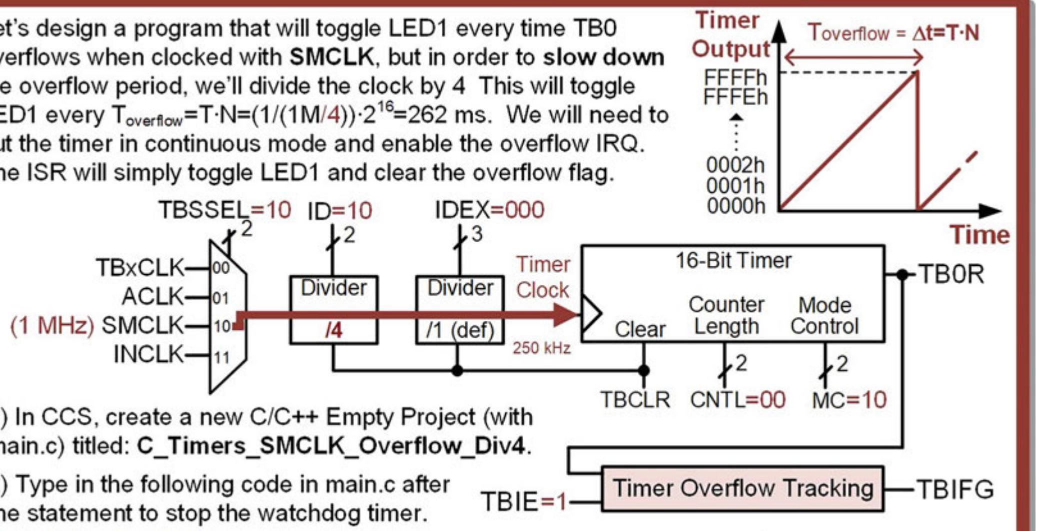
Did it work? Did you see LED1 toggle every 65.5ms? This will turn on/off 8 times per second.

Example 13.14
Toggling LED1 on TB0 overflow using SMCLK in C

Next, let's look at how to slow down the overflow period by dividing the clock frequency. Follow Example 13.15 to see how to divide the SMCLK source by 4 to produce a timer clock of 250 kHz. This example will again toggle LED1 every time there is a timer overflow.

Example: Toggling LED1 on TB0 Overflow Using SMCLK and a Divide-by-4 Clock Configuration in C

Let's design a program that will toggle LED1 every time TB0 overflows when clocked with **SMCLK**, but in order to **slow down** the overflow period, we'll divide the clock by 4 This will toggle LED1 every $T_{overflow}$=T·N=(1/(1M/4))·2^{16}=262 ms. We will need to put the timer in continuous mode and enable the overflow IRQ. The ISR will simply toggle LED1 and clear the overflow flag.



1) In CCS, create a new C/C++ Empty Project (with main.c) titled: **C_Timers_SMCLK_Overflow_Div4**.

2) Type in the following code in main.c after the statement to stop the watchdog timer.

```
#include <msp430.h>

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

    //-- Setup Ports
    P1DIR |= BIT0;                  // Config P1.0 (LED1) as output
    P1OUT &= ~BIT0;                 // Clear P1.0 (LED1) to start
    PM5CTL0 &= ~LOCKLPM5;           // Turn on GPIO

    //-- Setup Timer
    TB0CTL |= TBCLR;                // Clear timer and dividers
    TB0CTL |= TBSSEL__SMCLK;        // Source=SMCLK
    TB0CTL |= MC__CONTINUOUS;       // Mode=Continuous
    TB0CTL |= ID__4;                // Divide-by-4

    //-- Setup Timer Overflow IRQ
    TB0CTL &= ~TBIFG;               // Clear TB0 flag
    TB0CTL |= TBIE;                 // Enable TB0 Overflow IRQ
    __enable_interrupt();           // Enable Maskable IRQs

     //-- Main Loop
    while(1){}                      // Loop forever

    return 0;
}
//-- Interrupt Service Routines --------------------
#pragma vector = TIMER0_B1_VECTOR
__interrupt void ISR_TB0_Overflow(void)
{
    P1OUT ^= BIT0;        // Toggle LED1
    TB0CTL &= ~TBIFG;     // Clear TB0 flag
}
```

Divide-by-4 is set by inserting one additional statement to configure ID=10.

3) Debug your program and run it.

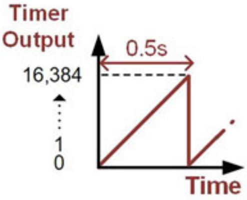
Did it work? Did you see LED1 toggle every 262ms? This will turn on/off 2 times per second.

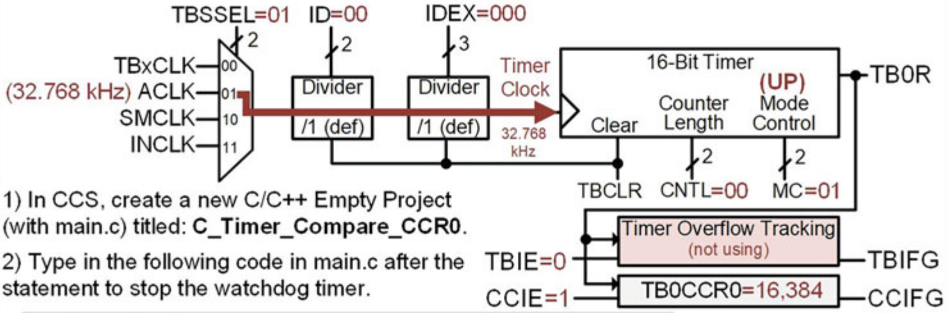
Example 13.15
Toggling LED1 on TB0 overflow using SMCLK and a divide-by-4 configuration in C

Next, let's look at programming timer compares in C. Follow Example 13.16 to see how to toggle LED1 every 0.5 second using a CCR0 IRQ. Recall that to use CCR0 IRQs, the timer must be in UP mode. Also note that CCR0 IRQs use a different vector address than the TB0IFG IRQ in the past examples.

Example: Toggling LED1 Using a CCR0 Compare in C

Let's design a program that will toggle LED1 every 0.5 s using a timer compare. We will use **ACLK** without any dividers. We will use CCR0, which allows us to simply enter a new maximum value for TB0R and then generate an IRQ upon overflow. First we calculate the value to put into CCR0 using: $T_{overflow}=T\cdot N=0.5s=(1/(32.768k))\cdot N$. This gives us a value of N=16,384 to achieve an overflow of 0.5s. We need to put the timer in UP mode to enable CCR0 IRQs. We then enable the CCR0 IRQ and create an ISR to toggle LED1 every time it is executed. The following is the timer setup:





1) In CCS, create a new C/C++ Empty Project (with main.c) titled: **C_Timer_Compare_CCR0**.

2) Type in the following code in main.c after the statement to stop the watchdog timer.

```
#include <msp430.h>

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

    //-- Setup Ports
    P1DIR |= BIT0;           // Config P1.0 (LED1) as output
    P1OUT &= ~BIT0;          // Clear P1.0 (LED1) to start
    PM5CTL0 &= ~LOCKLPM5;    // Turn on GPIO

    //-- Setup Timer
    TB0CTL |= TBCLR;         // Clear timer and dividers
    TB0CTL |= TBSSEL__ACLK;  // Source=ACLK
    TB0CTL |= MC__UP;        // Mode=UP
    TB0CCR0 = 16384;         // CCR0=16384

    //-- Setup Timer Compare IRQ
    TB0CCTL0 &= ~CCIFG;      // Clear CCR0 Flag
    TB0CCTL0 |= CCIE;        // Enable TB0 CCR0 Overflow IRQ
    __enable_interrupt();    // Enable Maskable IRQs

     //-- Main Loop
    while(1){}               // Loop forever

    return 0;
}

//-- Interrupt Service Routines ---------------
#pragma vector = TIMER0_B0_VECTOR
__interrupt void ISR_TB0_CCR0(void)
{
    P1OUT ^= BIT0;           // Toggle LED1
    TB0CCTL0 &= ~CCIFG;      // Clear CCR0 Flag
}
```

Recall that the timer must be in UP mode to use the CCR0 overflow IRQ.

TB0CCR0 holds the compare value.

The CCR0 interrupt is configured in the TB0CCTL0 register.

The vector for CCIFG0 is "TIMER0_B0_VECTOR". This is different that what we used for TB0IFG.

3) Debug your program and run it.

? Did it work? Did you see LED1 toggle every 0.5s?

Example 13.16
Toggling LED1 using a CCR0 compare in C

Next, let's look at generating a PWM signal using two separate timer compares. Follow Examples 13.17 and 13.18 to see how to configure a PWM signal on the MSP430FR2355 in C. This example drives LED1 with a 5% duty cycle signal with a period of 1 second. This provides a momentary flash on LED1. The period of the PWM signal is dictated by a compare on CCR0, and the duty cycle is dictated by a separate compare on CCR1.

Example: Flashing LED1 with a 5% Duty Cycle PWM Using Multiple Compares in C (part 1)

Let's design a program that will drive LED1 with a PWM signal with a period of 1 second and a duty cycle of 5%. This will result in a momentary flash on the LED every second. 5% of 1 second is only 50 ms, but it is still long enough to see with the naked eye. We will use ACLK without dividers to create a timer clock of 32.768 kHz. We will use a CCR0 compare to set the period of the PWM to 1 second. We will use a CCR1 compare to set the duty cycle to 5%. We will initialize LED1 to a 1 upon startup. When the first compare on CCR1 occurs after 50 ms, we will drive LED1 to a 0. When the second compare on CCR0 occurs after 1 second, we will drive LED1 back to a 1. Since CCR0 also sets the overflow period, as long as we running in "up" mode, this will repeat forever. First, let's calculate the values to put into CCR0 to achieve a period of 1 second and the value to put into CCR1 to achieve a duty cycle of 5%=50ms.

CCR0

$\Delta t = T\cdot N$

$1\text{ s} = (1/32.768\text{k})\cdot N$

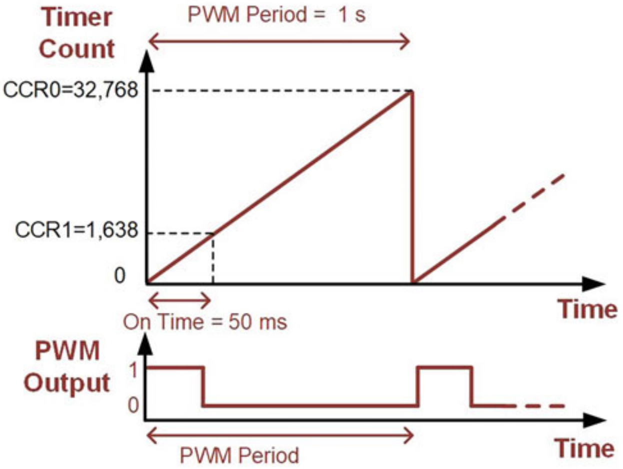
→ N=32,768

CCR1

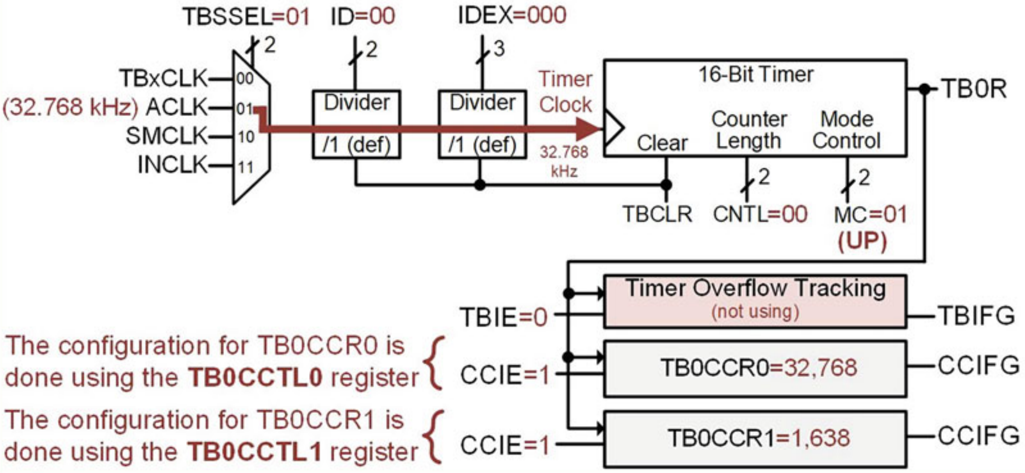
$\Delta t = T\cdot N$

$50\text{m} = (1/32.768\text{k})\cdot N$

→ N=1,638



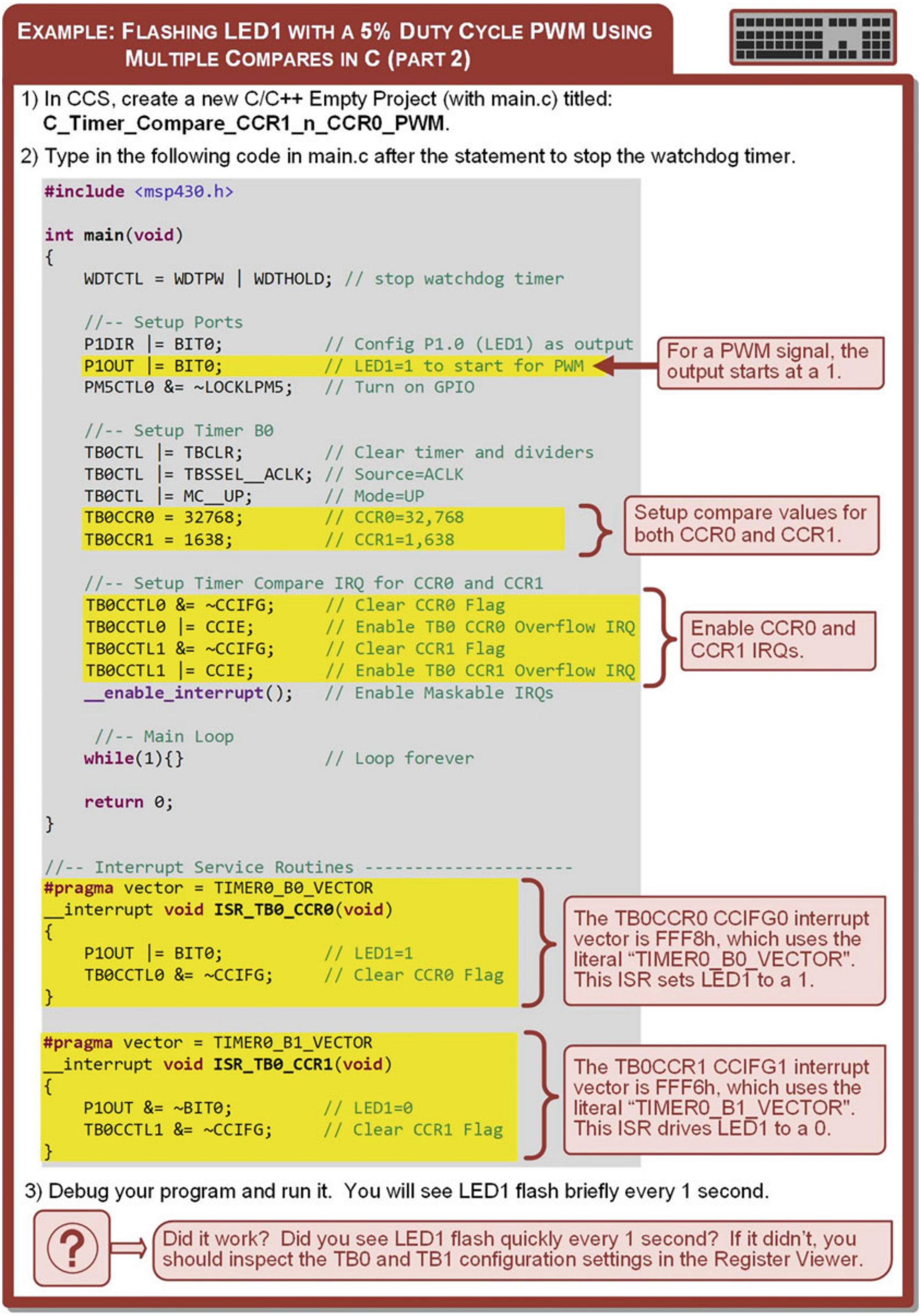
The following is our timer setup:



The configuration for TB0CCR0 is done using the **TB0CCTL0** register

The configuration for TB0CCR1 is done using the **TB0CCTL1** register

Example 13.17
Flashing LED1 with a 5% duty cycle PWM using multiple compares in C (part 1)



Example: Flashing LED1 with a 5% Duty Cycle PWM Using Multiple Compares in C (part 2)

1) In CCS, create a new C/C++ Empty Project (with main.c) titled:
C_Timer_Compare_CCR1_n_CCR0_PWM.

2) Type in the following code in main.c after the statement to stop the watchdog timer.

```
#include <msp430.h>

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

    //-- Setup Ports
    P1DIR |= BIT0;            // Config P1.0 (LED1) as output
    P1OUT |= BIT0;            // LED1=1 to start for PWM
    PM5CTL0 &= ~LOCKLPM5;     // Turn on GPIO

    //-- Setup Timer B0
    TB0CTL |= TBCLR;          // Clear timer and dividers
    TB0CTL |= TBSSEL__ACLK;   // Source=ACLK
    TB0CTL |= MC__UP;         // Mode=UP
    TB0CCR0 = 32768;          // CCR0=32,768
    TB0CCR1 = 1638;           // CCR1=1,638

    //-- Setup Timer Compare IRQ for CCR0 and CCR1
    TB0CCTL0 &= ~CCIFG;       // Clear CCR0 Flag
    TB0CCTL0 |= CCIE;         // Enable TB0 CCR0 Overflow IRQ
    TB0CCTL1 &= ~CCIFG;       // Clear CCR1 Flag
    TB0CCTL1 |= CCIE;         // Enable TB0 CCR1 Overflow IRQ
    __enable_interrupt();     // Enable Maskable IRQs

     //-- Main Loop
    while(1){}                // Loop forever

    return 0;
}

//-- Interrupt Service Routines --------------------
#pragma vector = TIMER0_B0_VECTOR
__interrupt void ISR_TB0_CCR0(void)
{
    P1OUT |= BIT0;            // LED1=1
    TB0CCTL0 &= ~CCIFG;       // Clear CCR0 Flag
}

#pragma vector = TIMER0_B1_VECTOR
__interrupt void ISR_TB0_CCR1(void)
{
    P1OUT &= ~BIT0;           // LED1=0
    TB0CCTL1 &= ~CCIFG;       // Clear CCR1 Flag
}
```

For a PWM signal, the output starts at a 1.

Setup compare values for both CCR0 and CCR1.

Enable CCR0 and CCR1 IRQs.

The TB0CCR0 CCIFG0 interrupt vector is FFF8h, which uses the literal “TIMER0_B0_VECTOR”. This ISR sets LED1 to a 1.

The TB0CCR1 CCIFG1 interrupt vector is FFF6h, which uses the literal “TIMER0_B1_VECTOR”. This ISR drives LED1 to a 0.

3) Debug your program and run it. You will see LED1 flash briefly every 1 second.

? Did it work? Did you see LED1 flash quickly every 1 second? If it didn’t, you should inspect the TB0 and TB1 configuration settings in the Register Viewer.

Example 13.18
Flashing LED1 with a 5% duty cycle PWM using multiple compares in C (part 2)

Concept Check

CC13.4 Programming the timer peripheral in C seems very similar to doing it in assembly due to all of the statements to configure the register settings. Is there any benefit to doing it in C?

A) No. It is about the same.

B) Yes. Programming in C still allows the compiler/optimizer to choose the most effective instructions to use to implement the specified functionality.

Summary

- ❖ The CCS design environment provides a main.c template for us when creating a new project. This provides statements to include the msp43.h header file and stop the watchdog timer.
- ❖ When creating a new CCS project (with main.c), the reset vector and stack pointer are automatically initialized for us outside of the main.c file.
- ❖ The literal names for the MSP430 registers and bit fields can be used directly in C statements.
- ❖ Block comments start with /* and end with */ and can span multiple lines.
- ❖ Line comments start with // and treat everything until the end of the line as a comment.
- ❖ MCU programs should never end. This means we need to insert looping structures in our main program to avoid the program ever reaching the `return 0;` statement.
- ❖ The CCS compiler contains optimization that will remove code that doesn't access the outside word or other peripherals. If we want to prevent optimization in order to inspect the step-by-step operation of a program, we need to turn it off.
- ❖ A while(1) loop can be used to create an infinite loop in our main program.
- ❖ A for() loop is used to create a structure that executes a fixed number of times. This can be used to create a delay loop or to provide an index for accessing other data structures.
- ❖ The if/else and switch/case statements provide a way to create conditionally executed statements.
- ❖ C provides arithmetic instructions for addition, subtraction, increment, and decrement.
- ❖ Increment and decrement operations can be coded with the shorthand *Var*++ or *Var*−−.
- ❖ C provides bitwise logic operations for complement (~), AND (&), OR (|), XOR (^), rotate arithmetically left (<<), and rotate arithmetically right (>>).
- ❖ Bitwise logic operations are useful for setting, clearing, and toggling bits within variables or registers.
- ❖ For bitwise logic operations where one of the two arguments is also the destination of the assignment, C provides a shorthand syntax for AND (&=), OR (|=), and XOR (^=).
- ❖ When using the digital I/O system of the MSP430FR2355, all of the configuration steps that were covered in assembly need to also be done when using C. These steps include altering all necessary configuration registers including PxDIR, PxREN, and PxOUT.
- ❖ When reading from an input port, we can make a simple assignment from the port input (PxIN) to an internal variable. We then clear out all bits that are not being viewed using a bitwise AND operation. This leaves our internal variable either a 0 or not 0 according to the value of the input port bit. This allows us to compare the variable to 0 in subsequent if/else statements to make assignments based on the input bit value.
- ❖ When using interrupts on the MSP430FR2355, all of the steps covered in assembly must be done when using C. These include setting up the peripherals, clearing the peripheral's interrupt flag, enabling the peripherals interrupt, asserting the GIE bit in the status register, writing an ISR, and initializing the vector table with the starting address of the ISR.
- ❖ For maskable interrupts on the MSP430FR2355, the `__enable_`

interrupt() function will set the GIE bit in the SR.

- A #pragma vector=<VECTOR_LABEL> directive is used to initialize the vector table with the address of the routine that immediately follows it.
- The VECTOR_LABELs when coding in C on the MSP430FR2355 are different from the labels used in assembly.
- The __interrupt() function tells CCS that the routine that follows is to be treated as an ISR and to use reti to be used at its end instead of ret.
- When programming timers on the MSP430FR2355 in C, all of the steps to configure the timers that were covered in assembly must also be done in C.
- Programming peripherals in C is easier to read due to the intuitive nature of the signal assignments and the use of descriptive literal names that come from the msp430.h header file.

Exercise Problems

Section 13.1: Basics of C Programming on the MSP430

13.1.1 What items are automatically inserted into the main.c file when creating a new CCS project for the MSP430FR2355?

13.1.2 What two steps are handled automatically by the CCS environment for us, but not explicitly inserted in the main.c filie?

13.1.3 How is a block comment coded in C?

13.1.4 How is a line comment coded in C?

13.1.5 What is the functionality of a while(1) loop?

13.1.6 Give the result of the following bitwise logic operation:

```
Var |= 0b00000010;
```

13.1.7 Give the result of the following bitwise logic operation:

```
Var &= 0b00000010;
```

13.1.8 Give the result of the following bitwise logic operation:

```
Var &= ~0b00000010;
```

13.1.9 Give the result of the following bitwise logic operation:

```
Var ^= 0b00000010;
```

13.1.10 Give the result of the following bitwise logic operation:

```
Var = Var << 2;
```

13.1.11 Give the result of the following bitwise logic operation:

```
Var = Var >> 3;
```

Section 13.2: Digital I/O in C

13.2.1 What does the following bitwise logic operation do regarding the digital I/O system on the MSP430FR2355?

```
P2DIR |= BIT6;
```

13.2.2 What does the following bitwise logic operation do regarding the digital I/O system on the MSP430FR2355?

```
P2DIR &= ~BIT6;
```

13.2.3 What does the following bitwise logic operation do regarding the digital I/O system on the MSP430FR2355?

```
P2REN |= BIT6;
```

13.2.4 What is the polarity of the pull-up/down resistor that results from the following bitwise logic operations on the MSP430FR2355?

```
P2REN |= BIT6;
P2OUT |= BIT6;
```

13.2.5 What is the polarity of the pull-up/down resistor that results from the following bitwise logic operations on the MSP430FR2355?

```
P2REN |= BIT6;
P2OUT &= ~BIT6;
```

Section 13.3: Interrupts in C

13.3.1 What is the vector label that we need to use when initializing the vector address in C for a port 2 interrupt?

13.3.2 What is the vector label that we need to use when initializing the vector address in C for a port 3 interrupt?

13.3.3 What does the __enable_interrupt() function do?

13.3.4 What does the #pragma vector=<VECTOR_LABEL> do?

13.3.5 What does the __interrupt() function do?

Section 13.4: Timers in C

13.4.1 What does the following bitwise logic operation do regarding the timer system on the MSP430FR2355?

```
TB0CTL |= CNTL_2;
```

13.4.2 What does the following bitwise logic operation do regarding the timer system on the MSP430FR2355?

```
TB0CTL |= ID_2;
```

13.4.3 What is the vector label that we need to use when initializing the vector table in C for a TB1 overflow (TB1IFG) interrupt?

13.4.4 What is the vector label that we need to use when initializing the vector table in C for a TB2 capture on CCR0?

13.4.5 What is the vector label that we need to use when initializing the vector table in C for a TB3 capture on CCR1?

Chapter 14: Serial Communication in C

This chapter looks at the *serial communication* capability of the MSP430 [1–3,12]. Serial communication uses a single wire to send information between a transmitter (Tx) and a receiver (Rx) as a sequence of bits. This is the opposite of how we can send information as a full word using the MCU ports, which is considered *parallel communication*. Serial communication allows information to be transmitted using less pins compared to parallel communication, but at a slower overall information transmission rate due to only sending one bit at a time as opposed to a full word. The MSP430 contains dedicated peri/pherals to handle serial communication called the *enhanced universal serial communication interfaces* (eUSCI). The MSP430FR2355 contains four separate eUSCI peripherals called eUSCI_A0, eUSCI_A1, eUSCI_B0, and eUSCI_B1. The MSP430FR2355 supports three serial communication modes including: (1) *universal asynchronous receiver/transmitter* (UART); (2) *serial peripheral interface* (SPI); and (3) *inter-integrated circuit* (I2C) protocol. The eUSCI_A0 and eUSCI_A1 peripherals can be configured to support either UART or SPI. The eUSCI_B0 and eUSCI_B1 peripherals can be configured to support either SPI or I2C. This chapter gives an overview of the UART, SPI, and I2C protocols and presents how to use these communication modes on the eUSCIs within the MSP430FR2355 in C.

Learning Outcomes—After completing this chapter, you will be able to:

14.1 Design programs in C that use the MSP430FR2355's UART communication peripheral.
14.2 Design programs in C that use the MSP430FR2355's SPI communication peripheral.
14.3 Design programs in C that use the MSP430FR2355's I2C communication peripheral.

14.1 Universal Asynchronous Receiver/Transmitter (UART)

14.1.1 The UART Standard

The universal asynchronous receiver/transmitter is an approach to serial communication between two devices in which neither device shares a common clock. This approach saves pins on each device by only using lines for the data; however, since a clock is not shared, a UART requires an approach to ensure that bits are not lost in the transmission due to the asynchronous nature of the communication. Thus, a UART has a predefined protocol for how data is framed, how data packets start and stop, and how the receiver oversamples the incoming data to recover the information sent. A UART was one of the most popular communication standards in early computers for applications that required cables in which cost was an issue and the least number of wires was desired. Examples of early UART applications were computer mice, keyboards, and printers.

The first step in setting up a UART is to decide on a common data rate between the Tx and Rx. In a UART, this is called a *baud rate* (BR). The baud rate is the fastest rate at which the data line changes states. On the MSP430, the only two logic states allowed are HIGH and LOW. This means that the baud rate also represents the fastest rate that a bit, or symbol, can be transmitted. The baud rate and bit period (T_B) are related by $BR = 1/T_B$. The baud rate must be manually set in both the Tx and Rx prior to data transmission. A UART uses a baud rate that comes from a list of commonly accepted values. These typically range from 9600 to 115,200 baud. The maximum rate of a UART stems from its asynchronous nature as it becomes more and more difficult for the receiver to recover the transmitted data as the rates get higher and the timing differences between each device's own clock becomes more noticeable. Figure 14.1 gives an overview of the baud rate and bit period relationship in a UART.

B. J. LaMeres, *Embedded Systems Design using the MSP430FR2355 LaunchPad™*,
https://doi.org/10.1007/978-3-031-20888-1_14

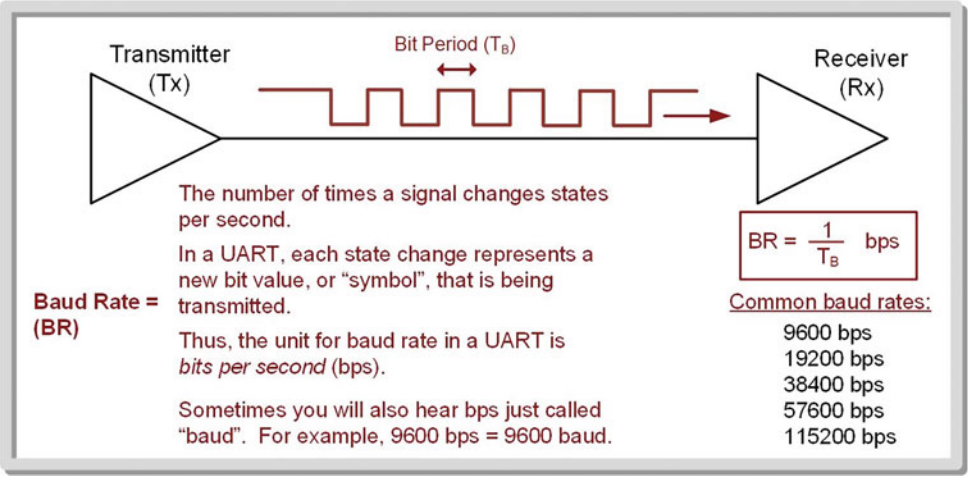

Transmitter
(Tx)
Bit Period (T_B)
Receiver
(Rx)
Baud Rate =
(BR)
The number of times a signal changes states per second.
In a UART, each state change represents a new bit value, or "symbol", that is being transmitted.
Thus, the unit for baud rate in a UART is *bits per second* (bps).
Sometimes you will also hear bps just called "baud". For example, 9600 bps = 9600 baud.
BR = $\frac{1}{T_B}$ bps
Common baud rates:
9600 bps
19200 bps
38400 bps
57600 bps
115200 bps

Fig. 14.1
UART baud rate

When two devices are connected together to transfer information, it is called a *link*. A UART can have different physical link configurations. A *simplex* link provides unidirectional data communication between a Tx on one device and an Rx on a second device using a single wire. A *full-duplex* link provides bidirectional communication between Tx/Rx pairs on both devices using two wires. This allows information to flow between the devices at the same time but requires two wires. A *half-duplex* link also contains Tx/Rx pairs on both devices but only uses a single wire. This reduces the number of wires in the physical link but requires that the two devices share the wire and cannot transmit at the same time. A half-duplex link requires additional handshaking steps to agree upon who is going to transmit at any given time. Figure 14.2 gives an overview of the possible link configurations. Note that while not shown, the Tx and Rx must share the same ground.

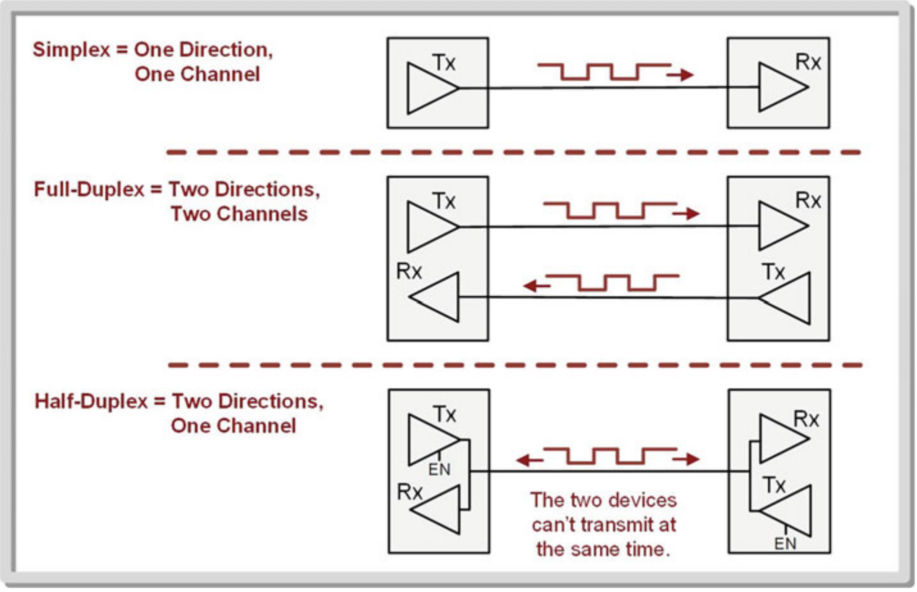

Simplex = One Direction,
One Channel
Tx
Rx
Full-Duplex = Two Directions,
Two Channels
Tx
Rx
Rx
Tx
Half-Duplex = Two Directions,
One Channel
Tx
EN
Rx
Rx
Tx
EN
The two devices can't transmit at the same time.

Fig. 14.2
Simplex vs. duplex link configurations

Framing is the term used to describe how the bits are arranged in the UART serial bit sequence. When a UART link is not transmitting, it is held at a logic HIGH. This idle logic level stems from early systems in which they wanted a way to make sure that the transmitting link was still operational. Having the Tx drive a HIGH while not transmitting tells the Rx that it is still powered on. To start a data transmission, the Tx drives the line LOW and holds it there for one T_B. This initial symbol is called the *start bit*. After that, the data word is transmitted bit by bit, typically starting with the LSB. After the bits have been transmitted, the Tx drives the line HIGH for one T_B to represent the end of the sequence. This last symbol is called the *stop bit*. Figure 14.3 shows the framing for a typical 8-bit UART transmission.

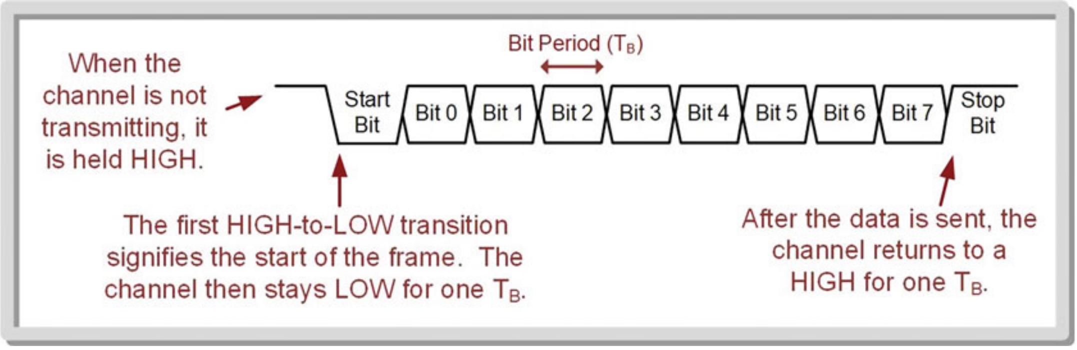

When the channel is not transmitting, it is held HIGH.
Bit Period (T_B)
Start Bit
Bit 0
Bit 1
Bit 2
Bit 3
Bit 4
Bit 5
Bit 6
Bit 7
Stop Bit
The first HIGH-to-LOW transition signifies the start of the frame. The channel then stays LOW for one T_B.
After the data is sent, the channel returns to a HIGH for one T_B.

Fig. 14.3
UART framing

There are many different framing variations beyond what is shown in Fig. 14.3. The most common options that are configurable on the MSP430 are swapping the order in which the bits are sent (i.e., sending MSB first, LSB last), changing the data size between 7 bits and 8 bits, adding an address bit (AD), adding a *parity* bit (PA), and adding a second stop bit. These framing options are shown in Fig. 14.4.

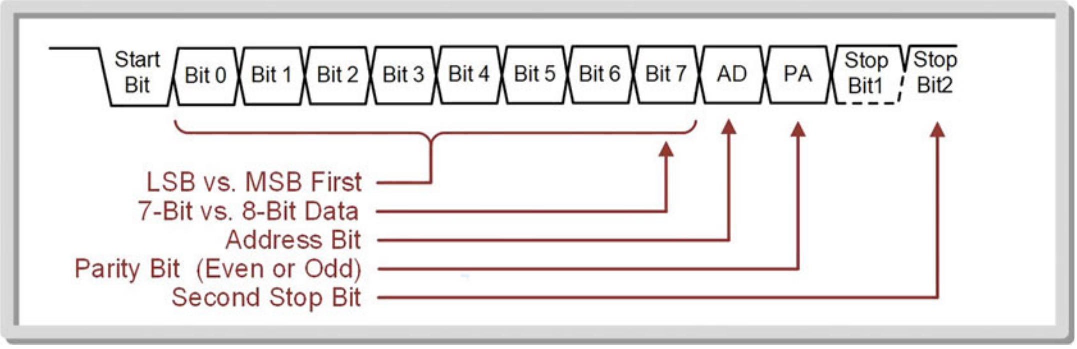


Fig. 14.4
UART framing options on the MSP430

When using a transmit address, two frames are sent, one for data and one for address. The AD bit signifies whether the next frame is the address or data. The addition of a second stop bit can be used to provide extra time between packets when running at higher baud rates in order to make the link more reliable.

The *parity* bit is used to detect transmission errors. There are two types of parity, even and odd. In *even parity*, the transmitter counts the number of 1's in the data word, and if the count is odd, it asserts the parity bit. This results in the total number of bits that are 1 between the data word and parity bit being even. The receiver can count the number of bits that were received, and if the sum is not an even number, it knows the frame was corrupted and can request the data to be sent again. In *odd parity*, this logic is reversed. The transmitter counts the number of 1's in the data word, and if the count is even, it asserts the parity bit. This results in the total number of bits that are 1's between the data word and parity bit being odd.

In a UART, the Rx can determine the incoming data without a synchronous clock by oversampling the data frame. The term *sample* refers to the rate at which data is stored by the receiver. In serial communication, a shift register based on D-flip-flops is used on both the Tx and Rx sides. The D-flip-flops in the Tx shift register are clocked at the baud rate frequency. This results in a new symbol being transmitted every Tx clock period, or T_B. The D-flip-flops in the Rx shift register are clocked at a rate that is higher than the Tx baud rate. The most common Rx oversampling ratio is 16× of the Tx clock. The Rx contains a counter that is also clocked off of the oversampling clock that counts the number of Rx clock cycles from the time a start bit is observed. The Rx knows that each bit period of the transmitted data frame is approximately 16 cycles of the Rx clock. This means that the start bit will take 16 cycles of the Rx clock, and then 8 cycles later, and the middle of bit 0 will be present in the shift register. Thereafter, every 16 Rx clock cycles will result in the center of the next bit in the frame being present in the shift register. After 136 Rx clock cycles, the receiver has sampled the entire data frame, and the shift register values can be transferred to an internal register and the sampling is complete. Figure 14.5 shows a graphical depiction of this sampling scheme.

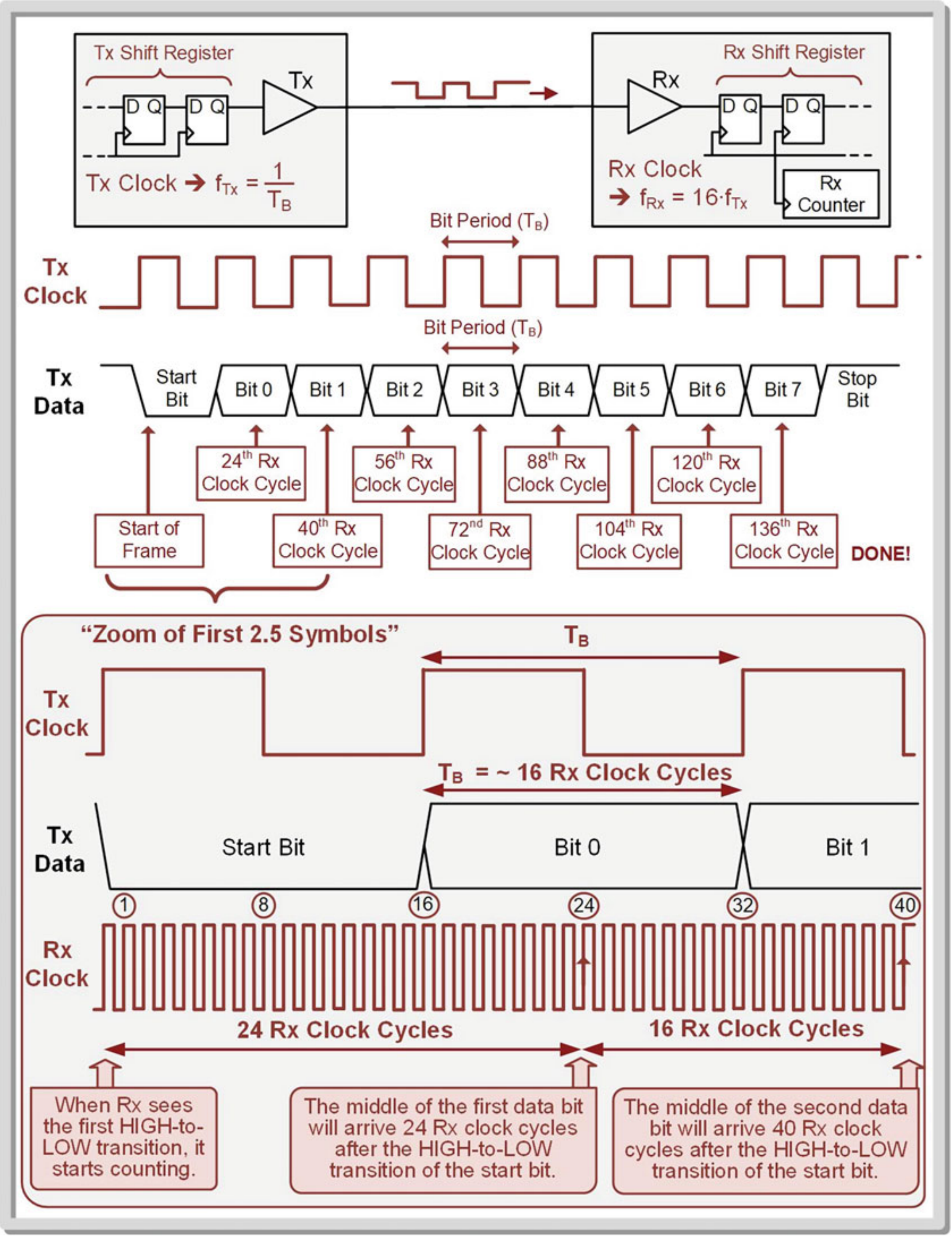
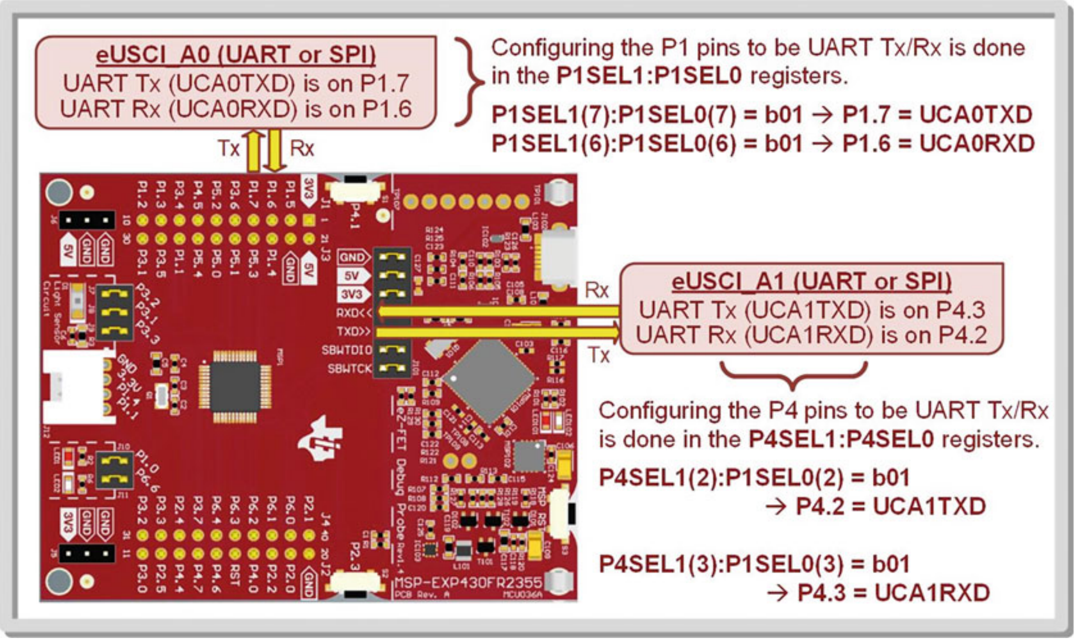


Fig. 14.5
UART clocking scheme (16× oversampling example)

The drawback of this approach is that the clock frequency used to generate the baud rate on the Tx will never be exactly the same as the clock frequency used to generate the baud rate oversampling ratio on the Rx. This error results in the possibility of the last bit of the frame being shifted either too far left or right of its expected position when the sampling is complete and the received data that is transferred to the Rx buffer being incorrect. This is the reason that a stop bit is necessary. The stop bit allows the receiver to *catch up* and then start over its sampling upon a new start bit. The clock error between the Tx and Rx is inherent in asynchronous communication and is a limiting factor in how fast the serial communication link can run.

Everything that has been covered about the UART assumes arbitrary logic levels of HIGH and LOW. When logic levels are assigned to the UART logic values, the system becomes a *communication standard*. A standard allows separate devices to communicate because both the logic levels and packet structure are known. The term transistor-to-transistor logic (TTL) is used to describe a UART that transmits a HIGH as the power supply (V_{CC}) of the device and a LOW as the GND of the device. Typically, we state the V_{CC} value of the device when we describe the standard (i.e., +3.4v TTL or +5v TTL). The MSP430FR2355 uses +3.4v TTL. The other most common UART standard is called the *recommended standard 232* (RS-232). In RS-232, a logic HIGH is represented as a voltage between −3v and −15v, and a logic LOW is represented as a voltage between +3 and +15v. These voltage ranges in RS-232 effectively invert the entire data frame. RS-232 standards are implemented by placing a transceiver chip on the UART lines of the MCU that translates the TTL levels to RS-232.

The MSP430FR2355 contains two eUSCIs that support UART. These are eUSCI_A0 and eUSCI_A1. These two eUSCI peripherals are configurable to either support UART or SPI. Each eUSCI has Tx and Rx signals that share pins with ports on the MCU. The eUSCI_A0 Tx/Rx pins share with port 1, bits 7 and 6, respectively. The eUSCI_A1 Tx/Rx pins share with port 4, bits 3 and 2, respectively. Figure 14.6 shows the location of the two UARTs on the MSP430FR2355 LaunchPad™ board.



eUSCI_A0 (UART or SPI)
UART Tx (UCA0TXD) is on P1.7
UART Rx (UCA0RXD) is on P1.6

Configuring the P1 pins to be UART Tx/Rx is done in the **P1SEL1:P1SEL0** registers.

P1SEL1(7):P1SEL0(7) = b01 → P1.7 = UCA0TXD
P1SEL1(6):P1SEL0(6) = b01 → P1.6 = UCA0RXD

eUSCI_A1 (UART or SPI)
UART Tx (UCA1TXD) is on P4.3
UART Rx (UCA1RXD) is on P4.2

Configuring the P4 pins to be UART Tx/Rx is done in the **P4SEL1:P4SEL0** registers.

P4SEL1(2):P1SEL0(2) = b01
→ P4.2 = UCA1TXD

P4SEL1(3):P1SEL0(3) = b01
→ P4.3 = UCA1RXD

Fig. 14.6
UART pins on the MSP430FR2355

14.1.2 UART Transmit on the MSP430FR2355

Let's start learning about UARTs by looking at the Tx capabilities of the MSP430FR2355. Recall that the MSP430FR2355 contains two eUSCIs that support UARTs, eUSCI_A0 and eUSCI_A1. These two peripherals are completely separate from each other and can support two independent UART links. Figure 14.7 shows a general block diagram of the UART Tx component of the eUSCI_Ax peripherals on the MSP430FR2355.

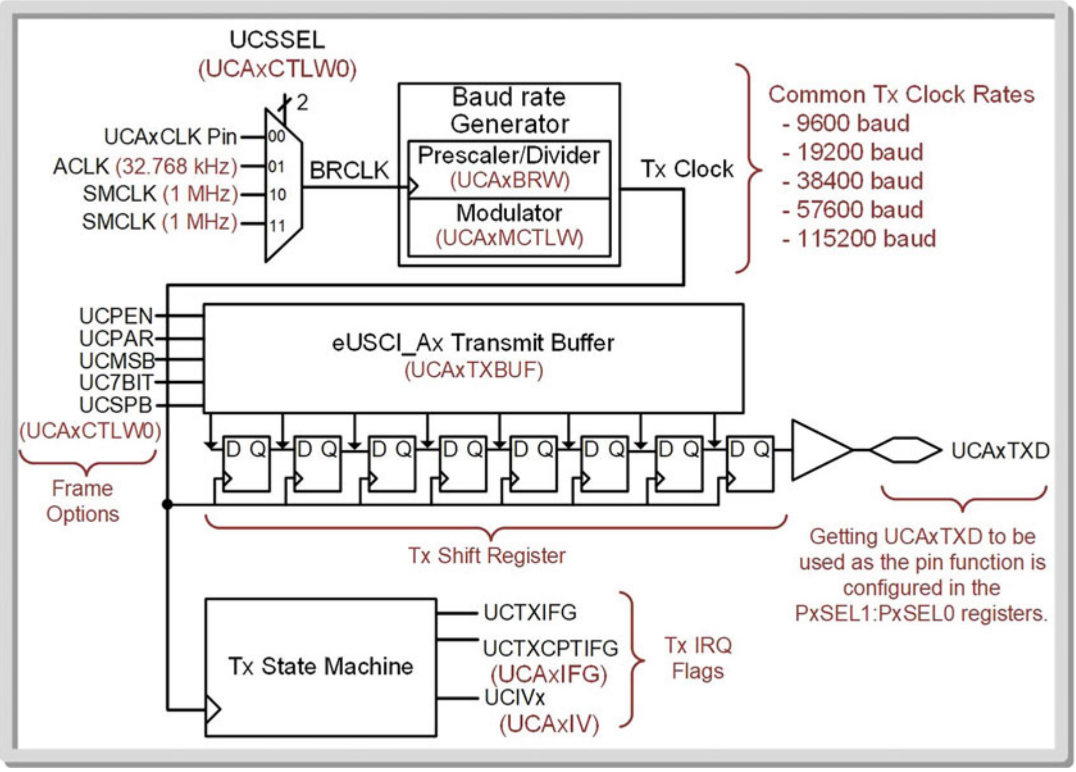


Fig. 14.7
UART Tx general block diagram of the eUSCI_Ax peripheral on the MSP430FR2355

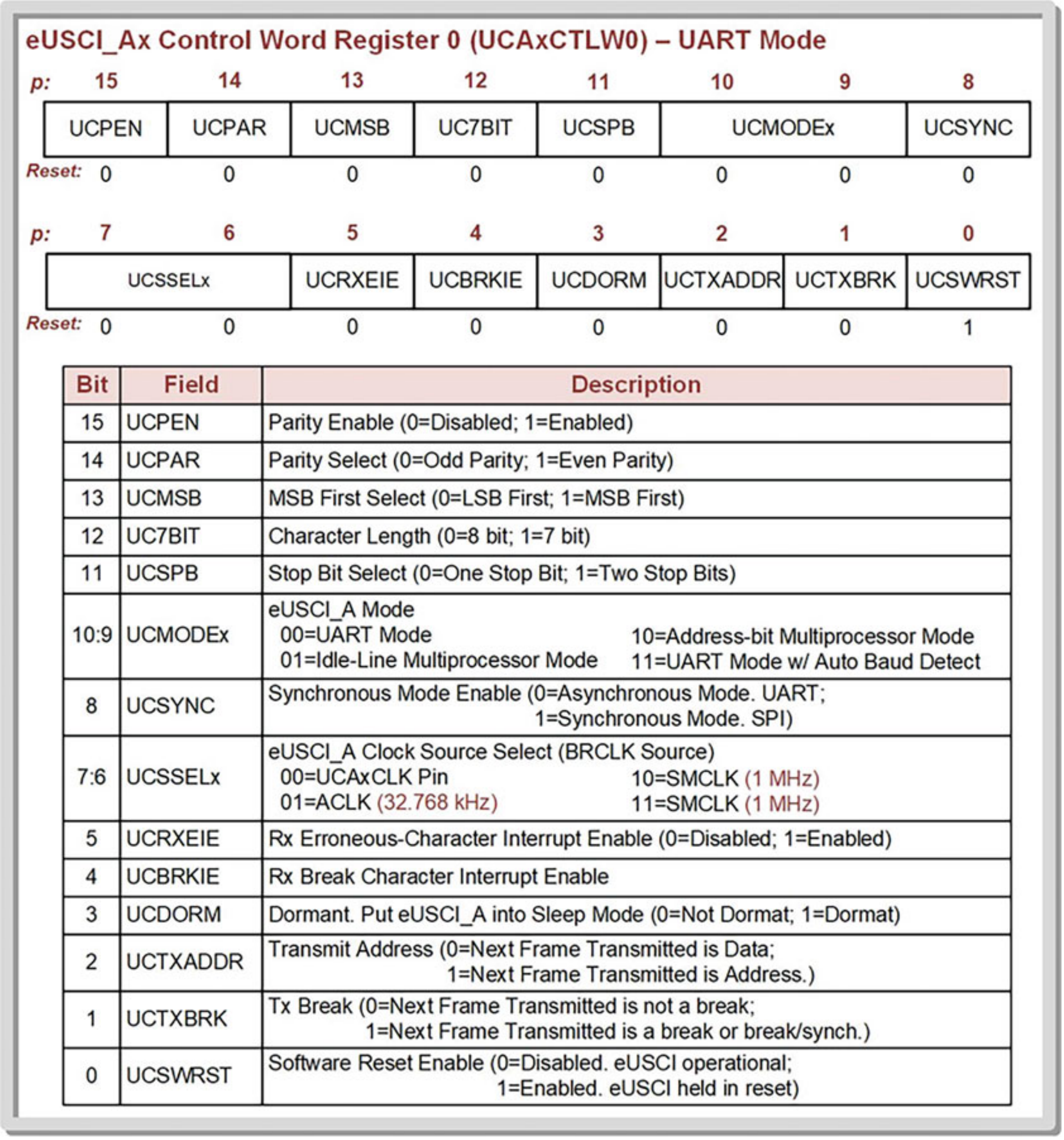
The basic concept of operation of the UART system is to first configure its baud rate and frame characteristics. We then store the data to be transmitted into a Tx buffer register, and a shift register will automatically send the data out over the Tx pin in a serial pattern. The UART can also produce a variety of interrupts such as *data received*, *transmit complete*, and many others used for detecting transmission errors. The UARTs are configured using a variety of registers. We will only cover the registers necessary to configure a basic UART. To see the more advanced UART features, refer to the MSP430 user's guide. More complex features are described in the MSP430 user's guide. The complete list of UART configuration registers is as follows:

- *eUSCI_Ax control word 0* (UCAxCTLW0)
- *eUSCI_Ax control word 1* (UCAxCTLW1)
- *eUSCI_Ax baud rate control word* (UCAxBRW)
- *eUSCI_Ax modulation control word* (UCAxMCTLW)
- *eUSCI_Ax status* (UCAxSTATW)
- *eUSCI_Ax receive buffer* (UCAxRXBUF)
- *eUSCI_Ax transmit buffer* (UCAxTXBUF)
- *eUSCI_Ax auto baud rate control* (UCAxABCTL)
- *eUSCI_Ax IrDA control* (UCAxIRCTL)
- *eUSCI_Ax interrupt enable* (UCAxIE)
- *eUSCI_Ax interrupt flag* (UCAxIFG)
- *eUSCI_Ax interrupt vector* (UCAxIV)

When configuring the UART peripheral, the recommended order of steps from the MSP430 user's guide is:

1. Set the UCSWRST bit in the UCAxCTLW0 configuration register to put the eUSCI_Ax peripheral into reset. Note that the default value for UCSWRST = 1 on system power-on or system reset, so the system is initially in software reset.
2. Initialize all eUSCI_Ax configuration registers.
3. Configure ports.
4. Clear UCSWRST to take the eUSCI_Ax peripheral out of reset.
5. Enable interrupts (optional) using the UCRXIE or UCTXIE bits in the UCAxIE configuration register.

To begin, we want to hold the eUSCI_Ax in reset while we configure it in order to avoid erroneous data from being transmitted. The eUSCI_Ax has a software reset that is enabled using the UCSWRST bit in the eUSCI_Ax control word 0 (UCAxCTLW0) register. Upon reset, UCSWRST = 1, so eUSCI_Ax is in software reset by default; however, it is good practice to explicitly set this bit in order to ensure the system is disabled. Figure 14.8 gives the details of UCAxCTLW0 register where the UCSWRST bit resides.



eUSCI_Ax Control Word Register 0 (UCAxCTLW0) – UART Mode

p:	15	14	13	12	11	10	9	8
	UCPEN	UCPAR	UCMSB	UC7BIT	UCSPB	UCMODEx		UCSYNC
Reset:	0	0	0	0	0	0	0	0

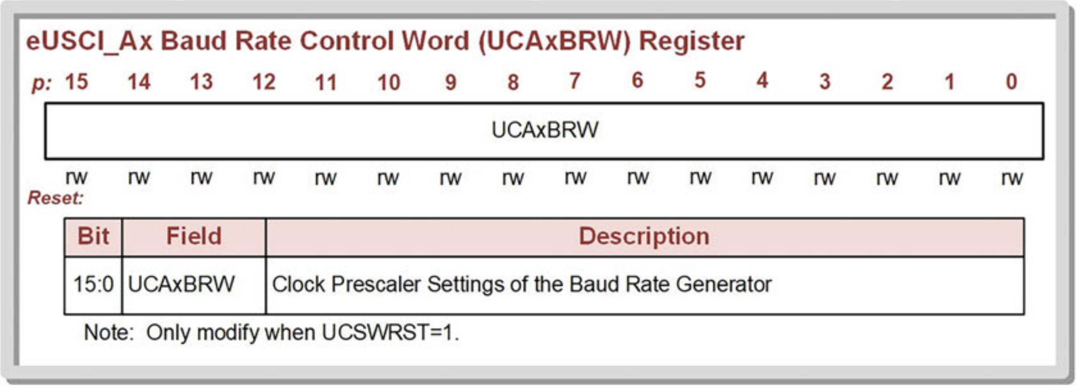
p:	7	6	5	4	3	2	1	0
	UCSSELx		UCRXEIE	UCBRKIE	UCDORM	UCTXADDR	UCTXBRK	UCSWRST
Reset:	0	0	0	0	0	0	0	1

Bit	Field	Description
15	UCPEN	Parity Enable (0=Disabled; 1=Enabled)
14	UCPAR	Parity Select (0=Odd Parity; 1=Even Parity)
13	UCMSB	MSB First Select (0=LSB First; 1=MSB First)
12	UC7BIT	Character Length (0=8 bit; 1=7 bit)
11	UCSPB	Stop Bit Select (0=One Stop Bit; 1=Two Stop Bits)
10:9	UCMODEx	eUSCI_A Mode 00=UART Mode 01=Idle-Line Multiprocessor Mode 10=Address-bit Multiprocessor Mode 11=UART Mode w/ Auto Baud Detect
8	UCSYNC	Synchronous Mode Enable (0=Asynchronous Mode. UART; 1=Synchronous Mode. SPI)
7:6	UCSSELx	eUSCI_A Clock Source Select (BRCLK Source) 00=UCAxCLK Pin 01=ACLK (32.768 kHz) 10=SMCLK (1 MHz) 11=SMCLK (1 MHz)
5	UCRXEIE	Rx Erroneous-Character Interrupt Enable (0=Disabled; 1=Enabled)
4	UCBRKIE	Rx Break Character Interrupt Enable
3	UCDORM	Dormant. Put eUSCI_A into Sleep Mode (0=Not Dormat; 1=Dormat)
2	UCTXADDR	Transmit Address (0=Next Frame Transmitted is Data; 1=Next Frame Transmitted is Address.)
1	UCTXBRK	Tx Break (0=Next Frame Transmitted is not a break; 1=Next Frame Transmitted is a break or break/synch.)
0	UCSWRST	Software Reset Enable (0=Disabled. eUSCI operational; 1=Enabled. eUSCI held in reset)

Fig. 14.8
eUSCI_Ax control word register 0 (UCAxCTLW0): UART mode

The next configuration step is selecting the clock for the eUSCI_Ax peripheral (i.e., BRCLK). This is done using the UCSSELx bits. On the MSP430FR2355, the default clock source is the external UCAxCLK pin on the MCU. We are given the choice of selecting either ACLK (UCSSELx = 01) or SMCLK (UCSSELx = 10 or 11) as internal clock sources for the eUSCI_Ax. On the MSP430FR2355 LaunchPad™ board, we will always use either ACLK or SMCLK for the source.

The next critical setting for the UART is its baud rate. The eUSCI_Ax peripheral has a baud rate generation circuit that can produce standard baud rates from nonstandard clock sources such as SMCLK and ACLK. The baud rate is set using two configuration registers, the eUSCI_Ax baud rate control word (UCAxBRW), and eUSCI_Ax modulation control word (UCAxMCTLW) registers. Figure 14.9 shows the details of UCAxBRW. Figure 14.10 shows the details of UCAxMCTLW.



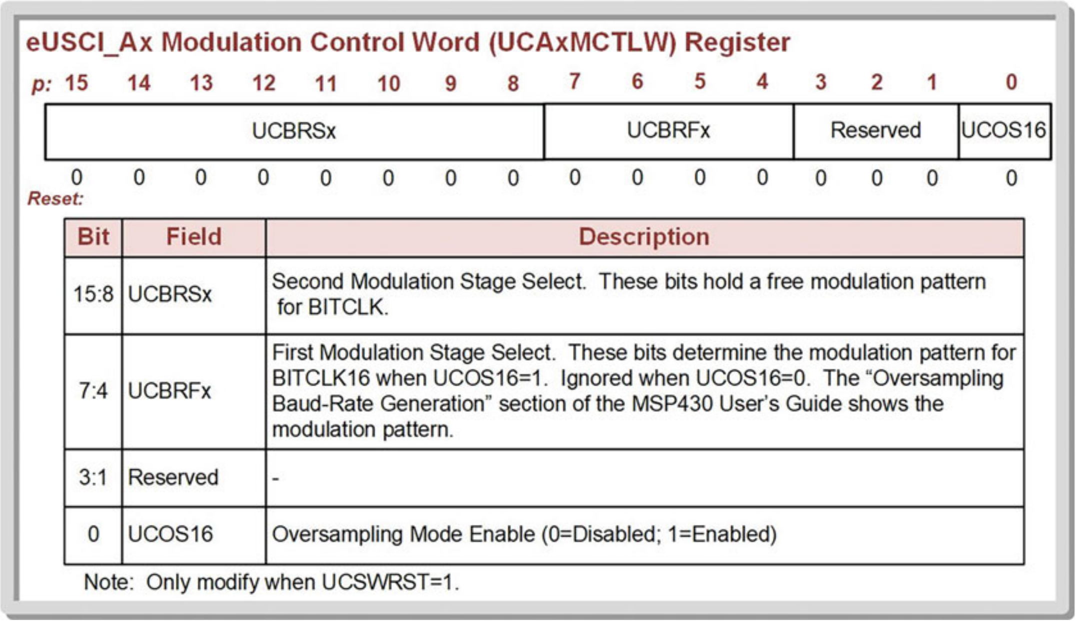
eUSCI_Ax Baud Rate Control Word (UCAxBRW) Register

p:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	UCAxBRW															
Reset:	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit	Field	Description
15:0	UCAxBRW	Clock Prescaler Settings of the Baud Rate Generator

Note: Only modify when UCSWRST=1.

Fig. 14.9
eUSCI_Ax baud rate control word (UCAxBRW) register



eUSCI_Ax Modulation Control Word (UCAxMCTLW) Register

p:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	UCBRSx								UCBRFx				Reserved			UCOS16
Reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

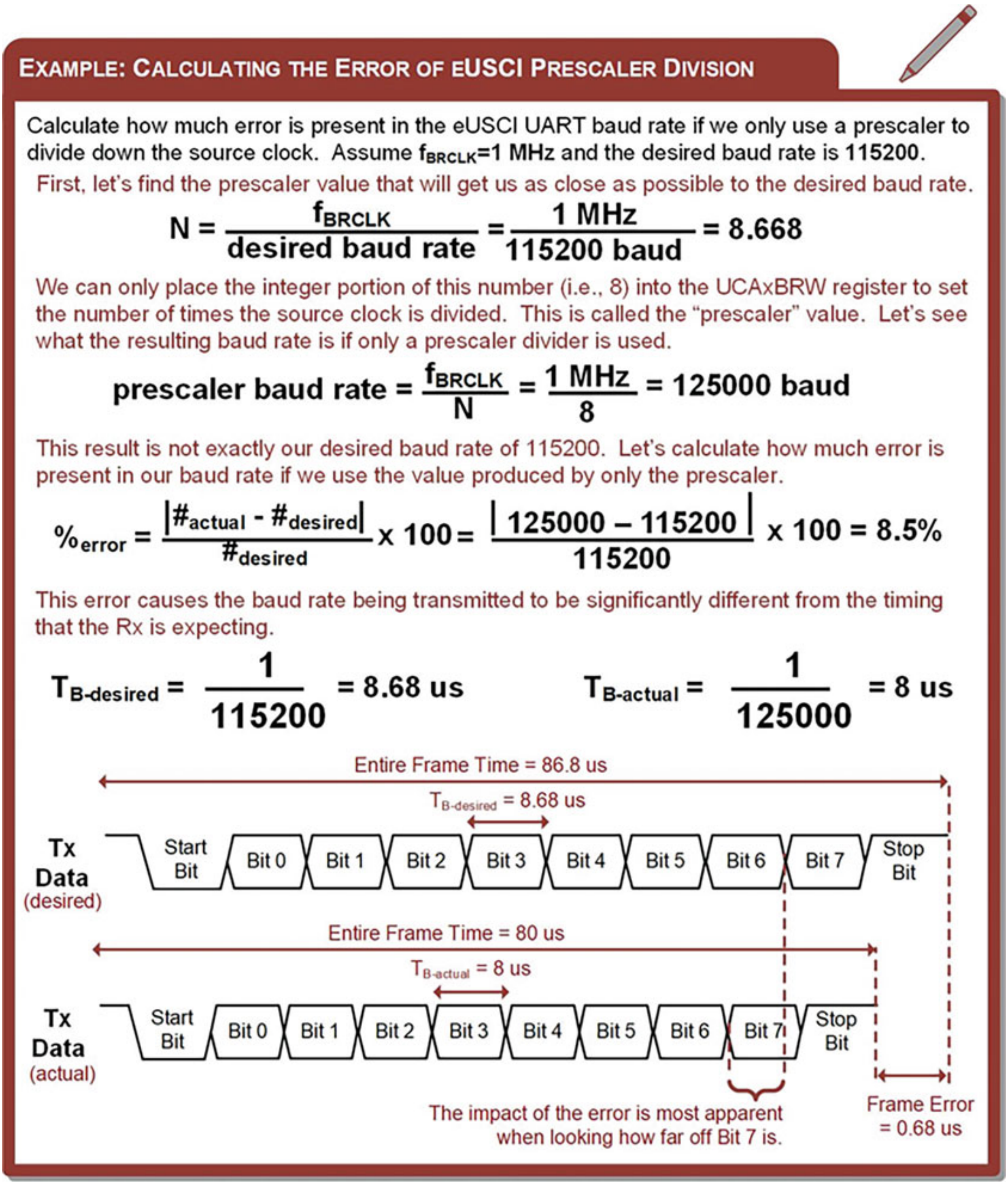
Bit	Field	Description
15:8	UCBRSx	Second Modulation Stage Select. These bits hold a free modulation pattern for BITCLK.
7:4	UCBRFx	First Modulation Stage Select. These bits determine the modulation pattern for BITCLK16 when UCOS16=1. Ignored when UCOS16=0. The "Oversampling Baud-Rate Generation" section of the MSP430 User's Guide shows the modulation pattern.
3:1	Reserved	-
0	UCOS16	Oversampling Mode Enable (0=Disabled; 1=Enabled)

Note: Only modify when UCSWRST=1.

Fig. 14.10
eUSCI_Ax modulation control word (UCAxMCTLW) register

The MSP430 baud rate generator has two modes of operation: *low frequency baud rate generation* and *oversampling baud rate generation*. The low frequency baud rate mode is used for baud rates that are less than or equal to 1/3 of BRCLK while the oversampling baud rate generation is used for higher ratios. The UCSO16 bit in the UCAxMCTLW register controls the modulation mode with UCSO16 = 0 being for low frequency (default) and UCSO16 = 1 being for oversampling mode.

In low frequency mode (UCSO16 = 0), the prescaler sets the initial division of BRCLK to get close to the desired baud rate, and then the modulation stages are used to eliminate the rest of the timing error. The first step in configuring the low frequency baud rate generator is to determine the prescaler value to be placed into UCAxBRW. In low power mode, the prescaler value sets the number of times the source clock (BRCLK) will be divided down to get close to the desired baud rate. The UCAxBRW register holds the integer portion of the quotient of the division of the eUSCI_Ax clock source (f_{BRCLK}) and the desired baud rate ($N = f_{BRCLK}$/baud_rate). This division often results in a non-integer value for N, which can lead to significant timing error and prevent the receiver from successfully recovering the transmitted data. To illustrate this timing error, consider the error calculations in Example 14.1 in which a desired rate of 115,200 baud is generated using only an integer prescaler division.



EXAMPLE: CALCULATING THE ERROR OF EUSCI PRESCALER DIVISION

Calculate how much error is present in the eUSCI UART baud rate if we only use a prescaler to divide down the source clock. Assume f_{BRCLK}=1 MHz and the desired baud rate is 115200.

First, let's find the prescaler value that will get us as close as possible to the desired baud rate.

$$N = \frac{f_{BRCLK}}{\text{desired baud rate}} = \frac{1\text{ MHz}}{115200\text{ baud}} = 8.668$$

We can only place the integer portion of this number (i.e., 8) into the UCAxBRW register to set the number of times the source clock is divided. This is called the "prescaler" value. Let's see what the resulting baud rate is if only a prescaler divider is used.

$$\text{prescaler baud rate} = \frac{f_{BRCLK}}{N} = \frac{1\text{ MHz}}{8} = 125000\text{ baud}$$

This result is not exactly our desired baud rate of 115200. Let's calculate how much error is present in our baud rate if we use the value produced by only the prescaler.

$$\%_{error} = \frac{|\#_{actual} - \#_{desired}|}{\#_{desired}} \times 100 = \frac{|125000 - 115200|}{115200} \times 100 = 8.5\%$$

This error causes the baud rate being transmitted to be significantly different from the timing that the Rx is expecting.

$$T_{B\text{-desired}} = \frac{1}{115200} = 8.68\text{ us} \qquad T_{B\text{-actual}} = \frac{1}{125000} = 8\text{ us}$$

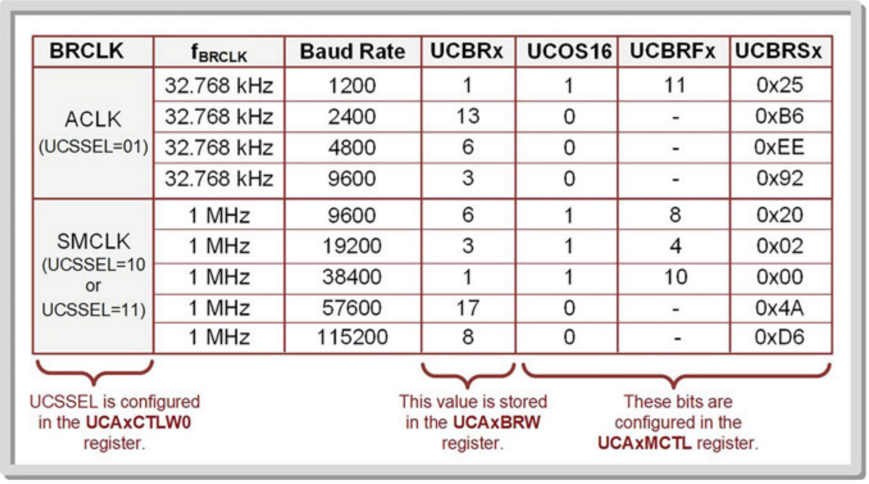
Entire Frame Time = 86.8 us
$T_{B\text{-desired}}$ = 8.68 us
Tx Data (desired)
Start Bit
Bit 0
Bit 1
Bit 2
Bit 3
Bit 4
Bit 5
Bit 6
Bit 7
Stop Bit
Entire Frame Time = 80 us
$T_{B\text{-actual}}$ = 8 us
Tx Data (actual)
Start Bit
Bit 0
Bit 1
Bit 2
Bit 3
Bit 4
Bit 5
Bit 6
Bit 7
Stop Bit
The impact of the error is most apparent when looking how far off Bit 7 is.
Frame Error = 0.68 us

Example 14.1
Calculating the error of eUSCI prescaler division

In order to compensate for the timing error resulting from the integer clock division in the prescaler, *modulation* stages are included on the MSP430. The MSP430's modulation circuits dynamically add and subtract BRCLK periods to the baud rate clock that drives the shift register. By adding small amounts of time from the BRCLK clock period to the baud rate clock, the timing error can be reduced. In low frequency mode, the baud rate generator uses the second modulation stage to create the desired baud rate. The second modulation stage is configured using the UCBRSx bits within the UCAxMCTLW register.

In oversampling baud rate mode (UCSO16 = 1), the prescaler register UCAxBRW holds the integer portion of *N* divided by 16 (i.e., INT(N/16) where *N* = (f_{BRCLK}/baud_rate)). Again, this division often results in a non-integer value for *N*/16, which can lead to timing error. In oversampling baud rate mode, both the first and second modulation stages are used to reduce this timing error. The first modulation stage is configured using the UCBRFx bits within the UCAxMCTLW register.

To simplify the settings for the first and second modulation stages, the MSP430 user's guide provides a table of settings that can be used for UCBRx, UCBRFx, and UCBRSx to achieve common baud rates based on typical BRCLK frequencies. On the MSP430FR2355 LaunchPad™ board, the BRCLK can only take on values of 32.768 kHz (ACLK) or 1 MHz (SMCLK) internally, so only a subset of the values provided in the user's guide are applicable to the LaunchPad™ board. Table 14.1 gives the relevant prescaler and modulation settings from the MSP430 user's guide that can be used on the LaunchPad™ board to achieve common baud rates.



BRCLK	f_{BRCLK}	Baud Rate	UCBRx	UCOS16	UCBRFx	UCBRSx
ACLK (UCSSEL=01)	32.768 kHz	1200	1	1	11	0x25
	32.768 kHz	2400	13	0	-	0xB6
	32.768 kHz	4800	6	0	-	0xEE
	32.768 kHz	9600	3	0	-	0x92
SMCLK (UCSSEL=10 or UCSSEL=11)	1 MHz	9600	6	1	8	0x20
	1 MHz	19200	3	1	4	0x02
	1 MHz	38400	1	1	10	0x00
	1 MHz	57600	17	0	-	0x4A
	1 MHz	115200	8	0	-	0xD6

UCSSEL is configured in the **UCAxCTLW0** register.

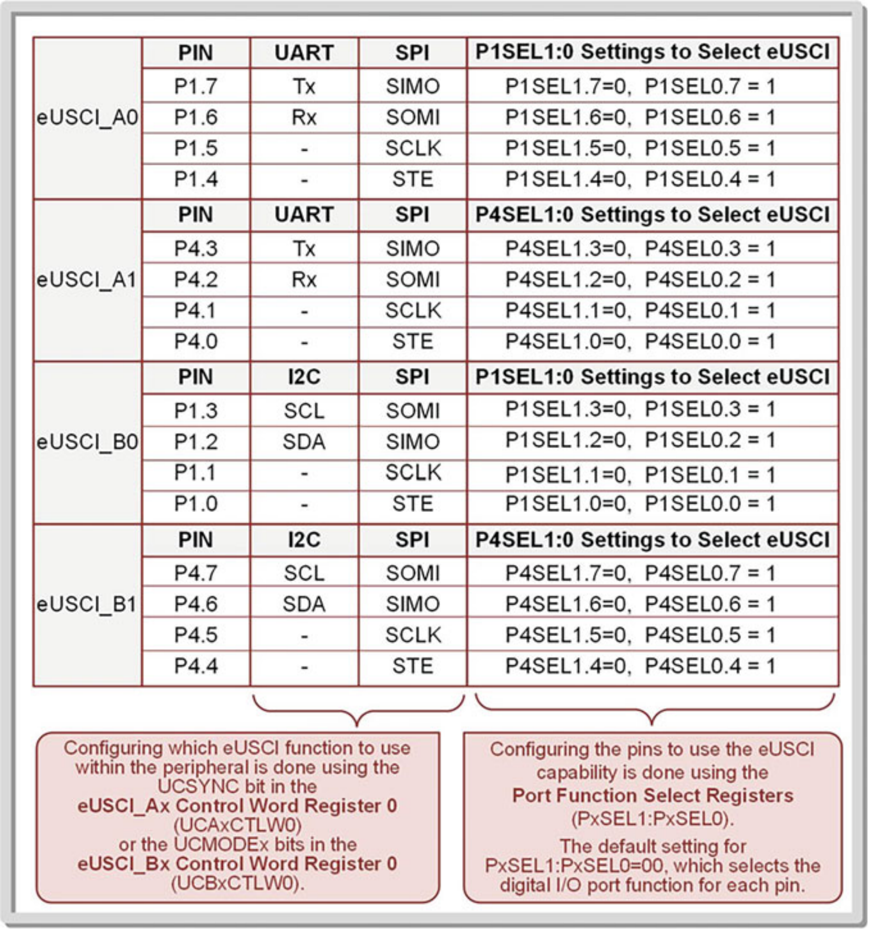
This value is stored in the **UCAxBRW** register.

These bits are configured in the **UCAxMCTL** register.

Table 14.1
Prescaler and modulation settings to configure the UART baud rate

Many of the UART framing options are also configured in UCAxCTLW0; however, many of the default settings for UCAxCTLW0 upon reset can be used to configure a typical UART frame. The UCSYNC value dictates whether the eUSCI_Ax peripheral is to be used as a UART (UCSYNC = 0 = asynchronous) or as a SPI (UCSYNC = 1 = synchronous). Since the default for UCSYNC = 0 upon reset, the eUSCI_Ax peripherals are in UART mode automatically. Additionally, the UCMODEx bits dictate whether the UART is in regular, in auto baud rate detect mode, or in one of two SPI modes. Since the default for UCMODEx = 00, upon reset, the eUSCI_Ax peripherals are also in regular UART mode automatically. Also, the reset values for UCTXBRK, UCTXADDR, UCDORM, UCBRKIE, UCRXEIE, UCSPB, UC7BIT, UCMSB, UCPAR, and UCPEN are all 0. If these settings are left at their default values, then the eUSCI_Ax peripheral comes up configured as a UART with typical frame settings.

At this point, we have covered all of the eUSCI_Ax settings to configure the UART. The next step is to configure the ports on the MSP430FR2355 to use the UART Tx and Rx functionality instead of the default digital I/O port functions. This is accomplished using the PxSEL1 and PxSEL0 configuration registers. These registers were mentioned in Table 9.1 when covering the digital I/O system. Recall that the default setting for the pins on the MSP430FR2355 is always the ports. Configuring the pin as a UART instead of a port will be the first time we have had to manually configure the PxSEL1 and PxSEL0 registers. Note that the function options for each pin on the MSP430FR2355 is unique, so there is not a consistent pattern to the mappings. The exhaustive list of function mappings is only given in the MSP430FR2355 device specific data sheet. Table 14.2 provides the detailed eUSCI mappings for the MSP430FR2355 MCU showing only the pins that have serial communication capability.



	PIN	UART	SPI	P1SEL1:0 Settings to Select eUSCI
eUSCI_A0	P1.7	Tx	SIMO	P1SEL1.7=0, P1SEL0.7 = 1
	P1.6	Rx	SOMI	P1SEL1.6=0, P1SEL0.6 = 1
	P1.5	-	SCLK	P1SEL1.5=0, P1SEL0.5 = 1
	P1.4	-	STE	P1SEL1.4=0, P1SEL0.4 = 1
	PIN	**UART**	**SPI**	**P4SEL1:0 Settings to Select eUSCI**
eUSCI_A1	P4.3	Tx	SIMO	P4SEL1.3=0, P4SEL0.3 = 1
	P4.2	Rx	SOMI	P4SEL1.2=0, P4SEL0.2 = 1
	P4.1	-	SCLK	P4SEL1.1=0, P4SEL0.1 = 1
	P4.0	-	STE	P4SEL1.0=0, P4SEL0.0 = 1
	PIN	**I2C**	**SPI**	**P1SEL1:0 Settings to Select eUSCI**
eUSCI_B0	P1.3	SCL	SOMI	P1SEL1.3=0, P1SEL0.3 = 1
	P1.2	SDA	SIMO	P1SEL1.2=0, P1SEL0.2 = 1
	P1.1	-	SCLK	P1SEL1.1=0, P1SEL0.1 = 1
	P1.0	-	STE	P1SEL1.0=0, P1SEL0.0 = 1
	PIN	**I2C**	**SPI**	**P4SEL1:0 Settings to Select eUSCI**
eUSCI_B1	P4.7	SCL	SOMI	P4SEL1.7=0, P4SEL0.7 = 1
	P4.6	SDA	SIMO	P4SEL1.6=0, P4SEL0.6 = 1
	P4.5	-	SCLK	P4SEL1.5=0, P4SEL0.5 = 1
	P4.4	-	STE	P4SEL1.4=0, P4SEL0.4 = 1

Configuring which eUSCI function to use within the peripheral is done using the UCSYNC bit in the **eUSCI_Ax Control Word Register 0** (UCAxCTLW0) or the UCMODEx bits in the **eUSCI_Bx Control Word Register 0** (UCBxCTLW0).

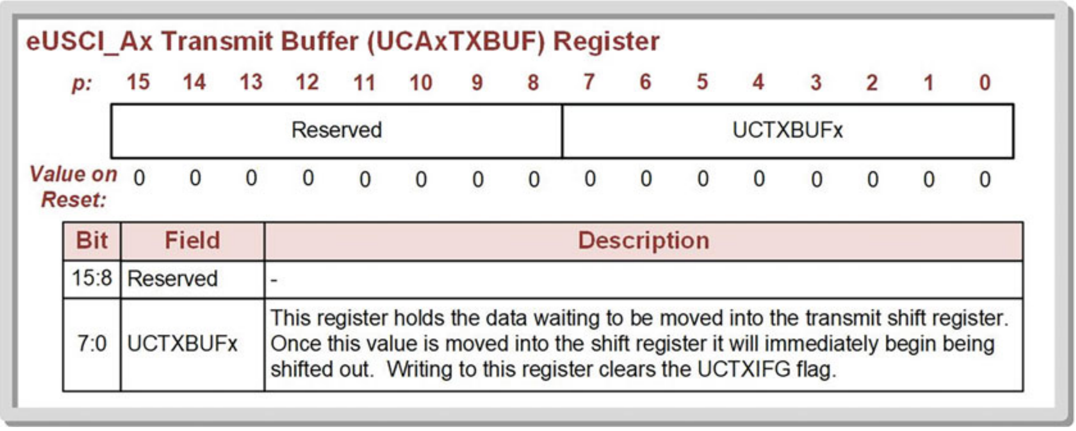
Configuring the pins to use the eUSCI capability is done using the **Port Function Select Registers** (PxSEL1:PxSEL0). The default setting for PxSEL1:PxSEL0=00, which selects the digital I/O port function for each pin.

Table 14.2
eUSCI pin mappings for the MSP430FR2355 MCU

At this point, the UART and the ports have been configured. The next step is to take the eUSCI_Ax peripheral out of software reset by clearing the UCSWRST bit in the UCAxCTLW0 register.

Once taken out of software reset, the eUSCI_Ax Tx module is enabled and will wait in an idle state until information is ready to be transmitted. While in the idle state, the baud rate generator is ready but not clocked nor producing any clocks. A Tx is initiated by writing a byte to the eUSCI_Ax Transmit Buffer (UCAxTXBUF). When this occurs, the data in UCAxTXBUF is moved into the Tx shift register and the baud rate generator begins producing clocks. The UCTXIFG flag provides the status of the transmission. When UCTXIFG = 0, data is being shifted out and new data should not be written to the Tx buffer. When

UCTXIFG = 1, new data can be written to the Tx buffer. Figure 14.11 gives the details of the UCAxTXBUF register.



eUSCI_Ax Transmit Buffer (UCAxTXBUF) Register

p:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Reserved								UCTXBUFx							
Value on Reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

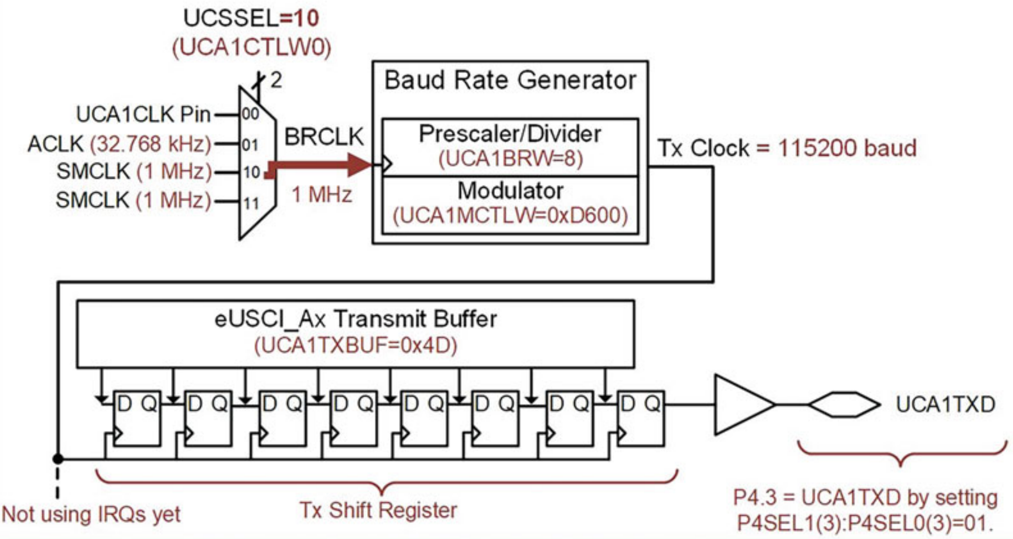
Bit	Field	Description
15:8	Reserved	-
7:0	UCTXBUFx	This register holds the data waiting to be moved into the transmit shift register. Once this value is moved into the shift register it will immediately begin being shifted out. Writing to this register clears the UCTXIFG flag.

Fig. 14.11
eUSCI_Ax transmit buffer (UCAxTXBUF) register

Let's start programming the UART on the LaunchPad™ board by first looking at sending out an 8-bit value that is continually stored to the transmit buffer. Follow the design in Examples 14.2, 14.3, and 14.4 to gain experience with using the MSP430FR2355 UART Tx.

EXAMPLE: TRANSMITTING A BYTE OVER UART AT 115200 BAUD (PART 1)

Let's configure the eUSCI_A1 peripheral as a UART and transmit the value **0x4D** at a rate of 115200 baud. The Tx output for eUSCI_A1 is on P4.3. We will use SMCLK as the source clock (BRCLK) and the normal UART framing options (8-bit, LSB first, no parity, no address, no extra stop bit). The following is the eUSCI_A1 setup for this example.

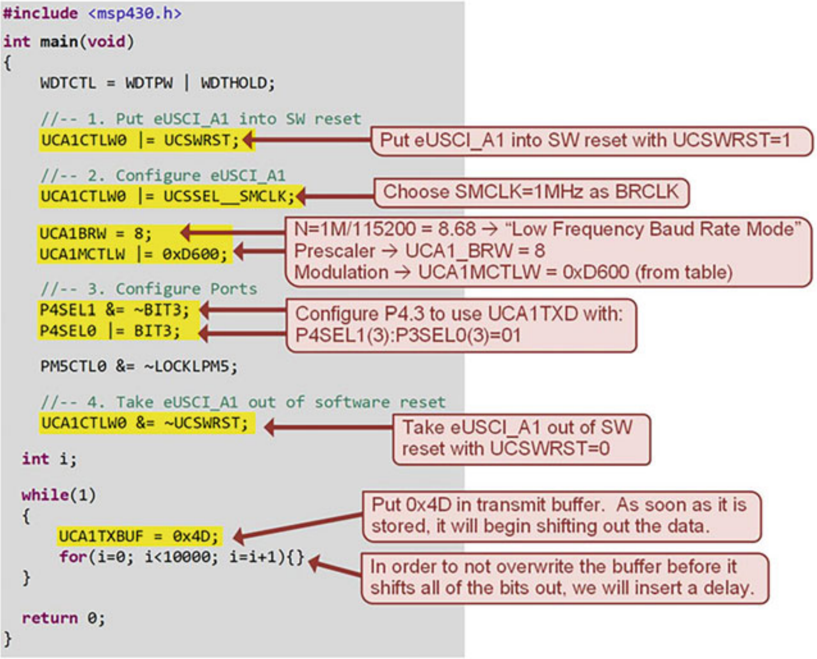


Example 14.2
Transmitting a byte over UART at 115200 baud (part 1)

Example: Transmitting a Byte Over UART at 115200 Baud (Part 2)

1) In CCS, create a new C/C++ Empty Project (with main.c) titled: **C_UART_Tx1_Sending_Byte_at_115200.**

2) Type in the following code in main.c after the statement to stop the watchdog timer.



```
#include <msp430.h>
int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;

    //-- 1. Put eUSCI_A1 into SW reset
    UCA1CTLW0  |= UCSWRST;

    //-- 2. Configure eUSCI_A1
    UCA1CTLW0  |= UCSSEL__SMCLK;

    UCA1BRW = 8;
    UCA1MCTLW  |= 0xD600;

    //-- 3. Configure Ports
    P4SEL1 &= ~BIT3;
    P4SEL0  |= BIT3;

    PM5CTL0 &= ~LOCKLPM5;

    //-- 4. Take eUSCI_A1 out of software reset
    UCA1CTLW0 &= ~UCSWRST;

  int i;

  while(1)
  {
      UCA1TXBUF = 0x4D;
      for(i=0; i<10000; i=i+1){}
  }

  return 0;
}
```

Put eUSCI_A1 into SW reset with UCSWRST=1

Choose SMCLK=1MHz as BRCLK

N=1M/115200 = 8.68 → "Low Frequency Baud Rate Mode"
Prescaler → UCA1_BRW = 8
Modulation → UCA1MCTLW = 0xD600 (from table)

Configure P4.3 to use UCA1TXD with:
P4SEL1(3):P3SEL0(3)=01

Take eUSCI_A1 out of SW reset with UCSWRST=0

Put 0x4D in transmit buffer. As soon as it is stored, it will begin shifting out the data.

In order to not overwrite the buffer before it shifts all of the bits out, we will insert a delay.

3) Debug your program and run it.

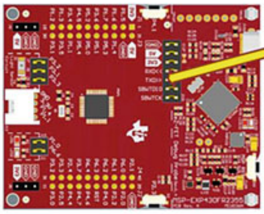
Note: We will first look at the Tx signal with an oscilloscope. If you don't have an oscilloscope, we will look at the signal in a different way in a later example.

Example 14.3
Transmitting a byte over UART at 115200 baud (part 2)

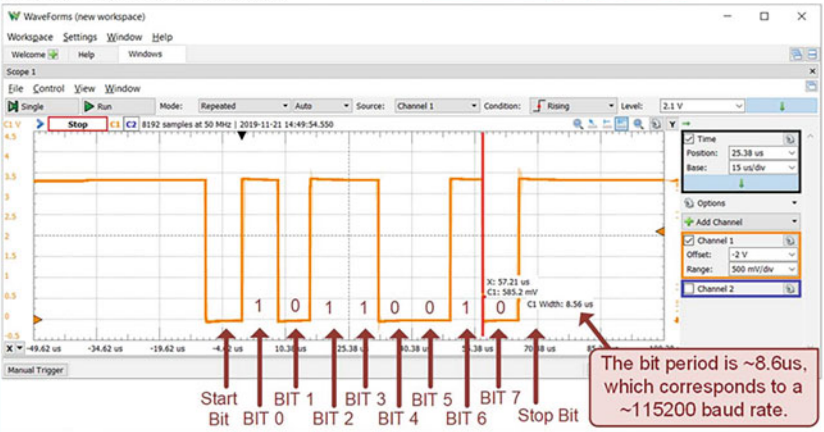
Example: Transmitting a Byte Over UART at 115200 Baud (Part 3)

4) Observe the UART with an oscilloscope. Remove the jumper on the pins labeled "TXD>>" on the J101 header of the MSP430FR2355 LaunchPad™ board. Probe the pin nearest the MSP430 MCU (MSP1). This pin is being driven by UCA1TXD.

5) Configure the oscilloscope waveform to center the UART frame. Measure the bit period of one of the short pulses.



Remove this jumper and probe with an oscilloscope.



Mode: Repeated, Auto Source: Channel 1 Condition: Rising Level: 2.1 V

8192 samples at 50 MHz | 2019-11-21 14:49:54.550

Time: Position: 25.38 us Base: 15 us/div

☑ Channel 1 Offset: -2 V Range: 500 mV/div

□ Channel 2

X: 57.21 us
C1: 585.2 mV

C1 Width: 8.56 us

The bit period is ~8.6us, which corresponds to a ~115200 baud rate.

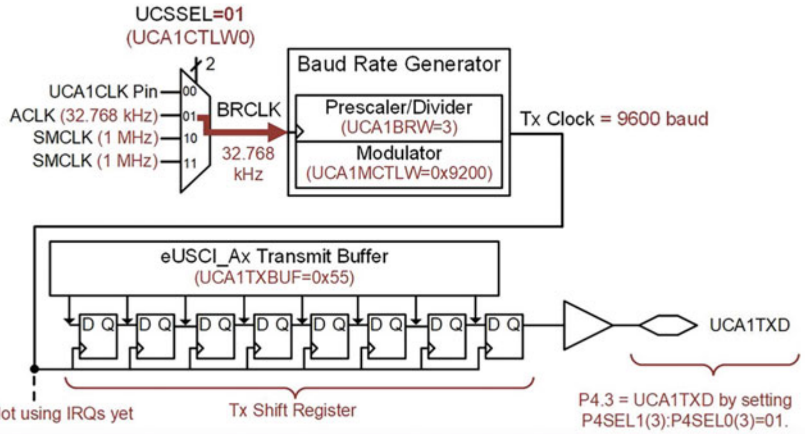
? ⇒ Did it work? Are you able to see the UART frame at 115200 baud? Notice that the sequence of bits received was 0b10110010. But remember that the UART sends the LSB first. Putting this in order from MSB we get: 0b01001101=0x4D.

Example 14.4
Transmitting a byte over UART at 115200 baud (part 3)

Now let's look at changing the baud rate from 115,200 to 9600 baud. Follow the design in Examples 14.5, 14.6, and 14.7 to gain an understanding of how to change the baud rate of the UART on the MSP430FR2355 MCU.

Example: Transmitting a Byte Over UART at 9600 Baud (Part 1)

Let's configure the eUSCI_A1 peripheral as a UART and transmit the value **0x55** at a rate of 9600 baud. The Tx output for eUSCI_A1 is on P4.3. We will use ACLK as the source clock (BRCLK) and the normal UART framing options (8-bit, LSB first, no parity, no address, no extra stop bit). The following is the eUSCI_A1 setup for this example.



Example 14.5
Transmitting a byte over UART at 9600 baud (part 1)

Example: Transmitting a Byte Over UART at 9600 Baud (Part 2)

1) In CCS, create a new C/C++ Empty Project (with main.c) titled:
C_UART_Tx2_Sending_Byte_at_9600.

2) Type in the following code in main.c after the statement to stop the watchdog timer.

```
#include <msp430.h>
int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;

    //-- 1. Put eUSCI_A1 into SW reset
    UCA1CTLW0 |= UCSWRST;

    //-- 2. Configure eUSCI_A1
    UCA1CTLW0 |= UCSSEL__ACLK;

    UCA1BRW = 3;
    UCA1MCTLW |= 0x9200;

    //-- 3. Configure Ports
    P4SEL1 &= ~BIT3;
    P4SEL0 |= BIT3;

    PM5CTL0 &= ~LOCKLPM5;

    //-- 4. Take eUSCI_A1 out of software reset
    UCA1CTLW0 &= ~UCSWRST;

  int i;

  while(1)
  {
      UCA1TXBUF = 0x55;
      for(i=0; i<10000; i=i+1){}
  }

  return 0;
}
```

Choose ACLK=32.768 kHz as BRCLK

N=32768/9600 = 3.41 → "Low Frequency Baud Rate Mode"
Prescaler → UCA1_BRW = 3
Modulation → UCA1MCTLW = 0x9200 (from table)

Put 0x55 in transmit buffer. As soon as it is stored, it will begin shifting out the data.

3) Debug your program and run it.

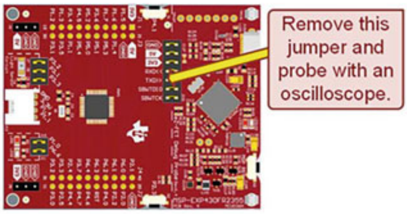
Note: We will first look at the Tx signal with an oscilloscope. If you don't have an oscilloscope, we will look at the signal in a different way in a later example.

Example 14.6
Transmitting a byte over UART at 9600 baud (part 2)

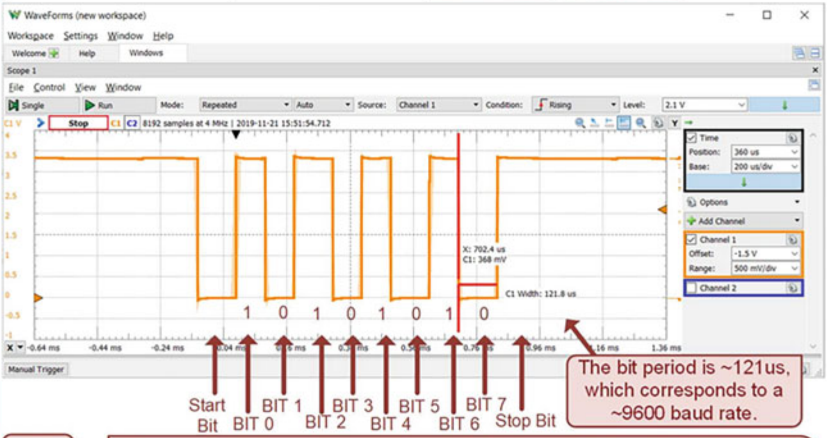
EXAMPLE: TRANSMITTING A BYTE OVER UART AT 9600 BAUD (PART 3)

4) Observe the UART with an oscilloscope. Remove the jumper on the pins labeled "TXD>>" on the J101 header of the MSP430FR2355 LaunchPad^{TM} board. Probe the pin nearest the MSP430 MCU (MSP1). This pin is being driven by UCA1TXD.

5) Configure the oscilloscope waveform to center the UART frame. Measure the bit period of one of the short pulses.



Remove this jumper and probe with an oscilloscope.



WaveForms (new workspace)

Scope 1

Mode: Repeated, Auto; Source: Channel 1; Condition: Rising; Level: 2.1 V

Stop | C1 | C2 | 8192 samples at 4 MHz | 2019-11-21 15:51:54.712

☑ Time — Position: 360 us; Base: 200 us/div

☑ Channel 1 — Offset: -1.5 V; Range: 500 mV/div

□ Channel 2

X: 702.4 us
C1: 368 mV

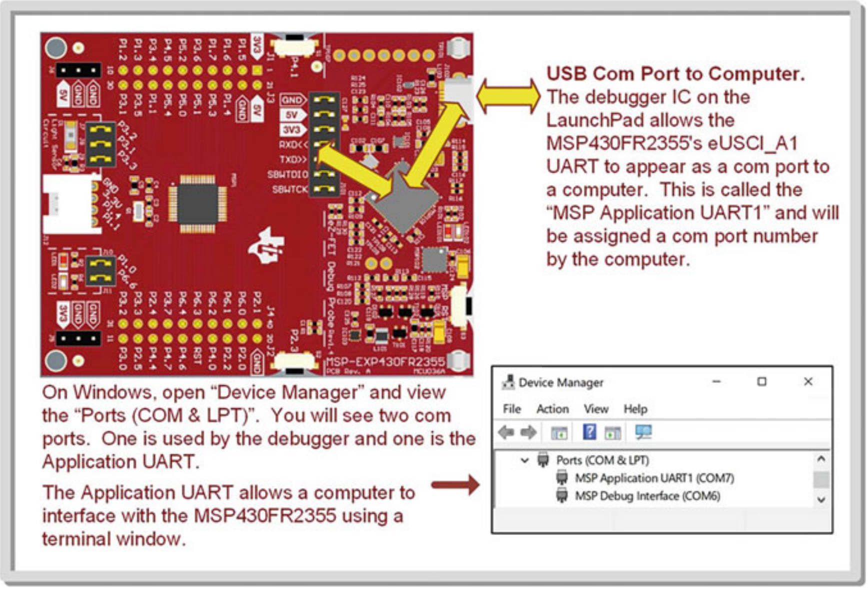
C1 Width: 121.8 us

The bit period is ~121us, which corresponds to a ~9600 baud rate.

? Did it work? Are you able to see the UART frame at 9600 baud? Notice that the sequence of bits received was 0b01010101. But remember that the UART sends the LSB first. Putting this in order from MSB we get: 0b10101010=0x55.

Example 14.7
Transmitting a byte over UART at 9600 baud (part 3)

Now let's begin using a *terminal* window from a computer to interface with the UART link on the MSP430FR2355. A terminal is a way to communicate with a device using standard I/O on a computer (i.e., a keyboard and mouse). A terminal uses the UART interface to send and receive bytes through a serial port on a computer (i.e., USB). The CCS design environment contains a built-in terminal feature that can be used to view data coming from the MSP430FR2355 LaunchPad™ board and also send information to the MCU board. The LaunchPad™ board contains a debug chip that handles downloading programs onto the MCU in addition to debugging programs. This chip also contains functionality to make the LaunchPad™ board appear as a standard "com" port to a computer. The com port will appear as the "Application UART1 (COMx)" when the LaunchPad™ board is plugged in. When setting up a terminal connection to the MSP430FR2355 LaunchPad™ board, the com port number of the Application UART1 is used as the connection ID. Figure 14.12 shows the connection path for the Application UART1 on the MSP430FR2355 LaunchPad™ board.



USB Com Port to Computer. The debugger IC on the LaunchPad allows the MSP430FR2355's eUSCI_A1 UART to appear as a com port to a computer. This is called the "MSP Application UART1" and will be assigned a com port number by the computer.

On Windows, open "Device Manager" and view the "Ports (COM & LPT)". You will see two com ports. One is used by the debugger and one is the Application UART.

The Application UART allows a computer to interface with the MSP430FR2355 using a terminal window.

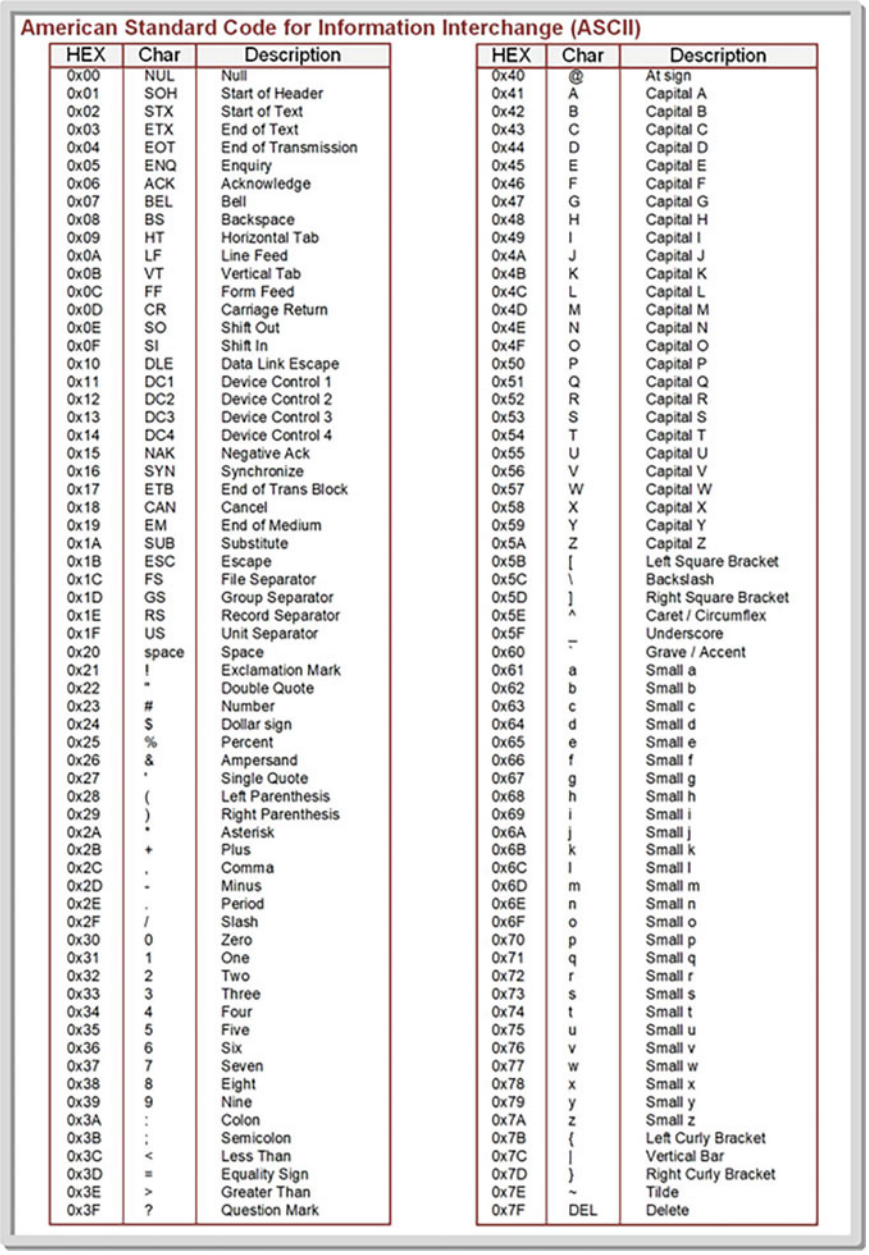
Device Manager

File Action View Help

Ports (COM & LPT)
- MSP Application UART1 (COM7)
- MSP Debug Interface (COM6)

Fig. 14.12
Application UART1 (COMx) port details on the MSP430FR2355™

By default, terminal windows interface with the standard I/O of the computer using a coding scheme called *ASCII*. ASCII stands for the *American Standard Code for Information Interchange*. In ASCII, every character in the American written language is assigned a unique 8-bit code. This coding approach allows each button press on a computer's keyboard to result in an 8-bit code being generated and understood by the computer's CPU. When writing a program, the data type "char" can be used to declare variables and store characters in ASCII. In C, when a statement such as `char Var1 = 'A';` is used, it will assign the ASCII code for the symbol "A" into the 8-bit variable labeled "Var1." Using the char datatype allows us to create programs that can send and receive ASCII characters using a terminal. This gives us the ability to send text to and from the MSP430FR2355 using the keyboard of a computer. Table 14.3 gives the definition of the 8-bit ASCII codes.



American Standard Code for Information Interchange (ASCII)

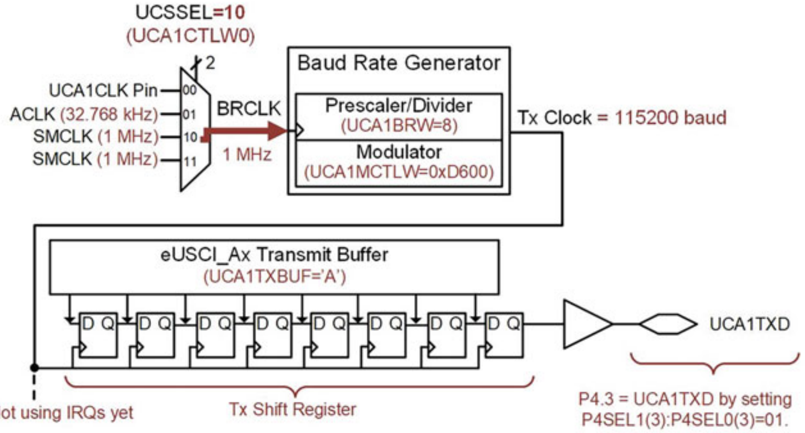
HEX	Char	Description	HEX	Char	Description
0x00	NUL	Null	0x40	@	At sign
0x01	SOH	Start of Header	0x41	A	Capital A
0x02	STX	Start of Text	0x42	B	Capital B
0x03	ETX	End of Text	0x43	C	Capital C
0x04	EOT	End of Transmission	0x44	D	Capital D
0x05	ENQ	Enquiry	0x45	E	Capital E
0x06	ACK	Acknowledge	0x46	F	Capital F
0x07	BEL	Bell	0x47	G	Capital G
0x08	BS	Backspace	0x48	H	Capital H
0x09	HT	Horizontal Tab	0x49	I	Capital I
0x0A	LF	Line Feed	0x4A	J	Capital J
0x0B	VT	Vertical Tab	0x4B	K	Capital K
0x0C	FF	Form Feed	0x4C	L	Capital L
0x0D	CR	Carriage Return	0x4D	M	Capital M
0x0E	SO	Shift Out	0x4E	N	Capital N
0x0F	SI	Shift In	0x4F	O	Capital O
0x10	DLE	Data Link Escape	0x50	P	Capital P
0x11	DC1	Device Control 1	0x51	Q	Capital Q
0x12	DC2	Device Control 2	0x52	R	Capital R
0x13	DC3	Device Control 3	0x53	S	Capital S
0x14	DC4	Device Control 4	0x54	T	Capital T
0x15	NAK	Negative Ack	0x55	U	Capital U
0x16	SYN	Synchronize	0x56	V	Capital V
0x17	ETB	End of Trans Block	0x57	W	Capital W
0x18	CAN	Cancel	0x58	X	Capital X
0x19	EM	End of Medium	0x59	Y	Capital Y
0x1A	SUB	Substitute	0x5A	Z	Capital Z
0x1B	ESC	Escape	0x5B	[	Left Square Bracket
0x1C	FS	File Separator	0x5C	\	Backslash
0x1D	GS	Group Separator	0x5D	]	Right Square Bracket
0x1E	RS	Record Separator	0x5E	^	Caret / Circumflex
0x1F	US	Unit Separator	0x5F	_	Underscore
0x20	space	Space	0x60	`	Grave / Accent
0x21	!	Exclamation Mark	0x61	a	Small a
0x22	"	Double Quote	0x62	b	Small b
0x23	#	Number	0x63	c	Small c
0x24	$	Dollar sign	0x64	d	Small d
0x25	%	Percent	0x65	e	Small e
0x26	&	Ampersand	0x66	f	Small f
0x27	'	Single Quote	0x67	g	Small g
0x28	(	Left Parenthesis	0x68	h	Small h
0x29	)	Right Parenthesis	0x69	i	Small i
0x2A	*	Asterisk	0x6A	j	Small j
0x2B	+	Plus	0x6B	k	Small k
0x2C	,	Comma	0x6C	l	Small l
0x2D	-	Minus	0x6D	m	Small m
0x2E	.	Period	0x6E	n	Small n
0x2F	/	Slash	0x6F	o	Small o
0x30	0	Zero	0x70	p	Small p
0x31	1	One	0x71	q	Small q
0x32	2	Two	0x72	r	Small r
0x33	3	Three	0x73	s	Small s
0x34	4	Four	0x74	t	Small t
0x35	5	Five	0x75	u	Small u
0x36	6	Six	0x76	v	Small v
0x37	7	Seven	0x77	w	Small w
0x38	8	Eight	0x78	x	Small x
0x39	9	Nine	0x79	y	Small y
0x3A	:	Colon	0x7A	z	Small z
0x3B	;	Semicolon	0x7B	{	Left Curly Bracket
0x3C	<	Less Than	0x7C	|	Vertical Bar
0x3D	=	Equality Sign	0x7D	}	Right Curly Bracket
0x3E	>	Greater Than	0x7E	~	Tilde
0x3F	?	Question Mark	0x7F	DEL	Delete

Table 14.3
American Standard Code for Information Interchange (ASCII)

Now let's look at defining the values to send over the UART using the type char in order to use the ASCII standard. We will insert a value of type char into the Tx buffer to be sent to the computer. We will then observe the bitstream with an oscilloscope to verify it is indeed the correct ASCII code per Table 14.3. We will then observe the ASCII symbol arriving using a terminal window built into CCS. Follow the design in Examples 14.8, 14.9, 14.10, and 14.11 to see how to use ASCII and a terminal window when transmitting with a UART.

Example: Transmitting a Character Over UART (part 1)

Let's continually send the character 'A' from the eUSCI_A1 UART at a rate of 115200 baud. We will first observe the character with an oscilloscope and then with a *terminal window* within CCS. The following is the eUSCI_A1 setup for this example.

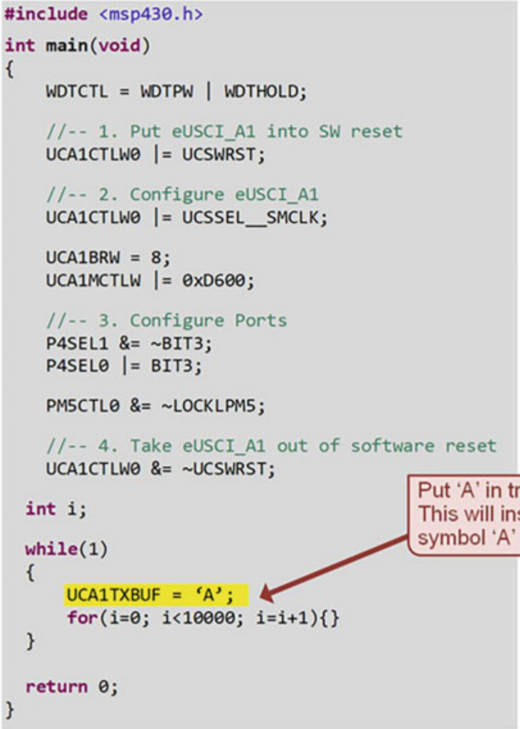


Example 14.8
Transmitting a character over UART (part 1)

Example: Transmitting a Character Over UART (part 2)

1) In CCS, create a new C/C++ Empty Project (with main.c) titled: **C_UART_Tx3_Sending_Char**.

2) Type in the following code in main.c after the statement to stop the watchdog timer.



```
#include <msp430.h>

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;

    //-- 1. Put eUSCI_A1 into SW reset
    UCA1CTLW0 |= UCSWRST;

    //-- 2. Configure eUSCI_A1
    UCA1CTLW0 |= UCSSEL__SMCLK;

    UCA1BRW = 8;
    UCA1MCTLW |= 0xD600;

    //-- 3. Configure Ports
    P4SEL1 &= ~BIT3;
    P4SEL0 |= BIT3;

    PM5CTL0 &= ~LOCKLPM5;

    //-- 4. Take eUSCI_A1 out of software reset
    UCA1CTLW0 &= ~UCSWRST;

  int i;

  while(1)
  {
      UCA1TXBUF = ‘A’;
      for(i=0; i<10000; i=i+1){}
  }

  return 0;
}
```

Put 'A' in transmit buffer using the type char. This will insert the ASCII code for the symbol 'A' (i.e., 0x41) into the buffer.

3) Debug your program and run it.

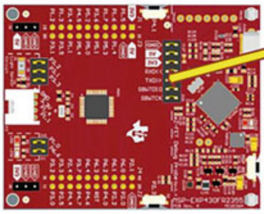
Note: We will first look at the Tx signal with an oscilloscope to view the ASCII code for 'A'. We will then look at the character using a terminal window.

Example 14.9
Transmitting a character over UART (part 2)

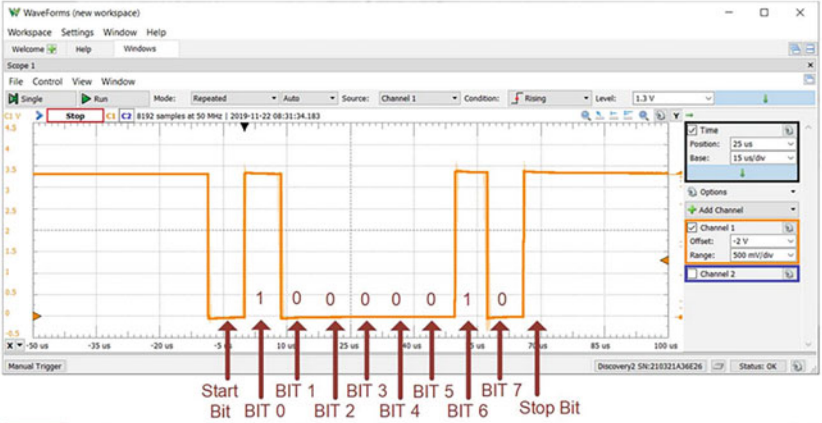
Example: Transmitting a Character Over UART (Part 3)

4) Observe the UART with an oscilloscope. Remove the jumper on the pins labeled “TXD>>” on the J101 header of the MSP430FR2355 LaunchPad^{TM} board. Probe the pin nearest the MSP430 MCU (MSP1). This pin is being driven by UCA1TXD.

5) Configure the oscilloscope waveform to center the UART frame.



Remove this jumper and probe with an oscilloscope.



WaveForms (new workspace)

Mode: Repeated, Auto

Source: Channel 1

Condition: Rising

Level: 1.3 V

8192 samples at 50 MHz | 2019-11-22 08:31:34.183

☑ Time
Position: 25 us
Base: 15 us/div

☑ Channel 1
Offset: -2 V
Range: 500 mV/div

□ Channel 2

Discovery2 SN:210321A36E26

Status: OK

?

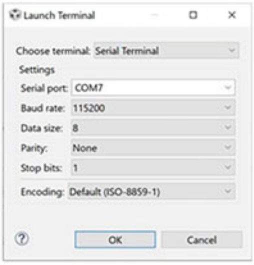
Did it work? You should have seen a bit stream arrive of 10000010. After rearranging this to have the MSB first, you get 01000001=0x41. This is the ASCII character for ‘A’ that was put into the Tx Buffer.

Example 14.10
Transmitting a character over UART (part 3)

EXAMPLE: TRANSMITTING A CHARACTER OVER UART (PART 4)

6) Now let's launch a terminal within CCS and observe the ASCII character arrive. In CCS, use the pull down menu View → Terminal. A terminal tab will appear within CCS (usually in the lower right corner along with the "Problems" and "Advice" tabs.

7) Configure the terminal window. Click on the "Open a Terminal" button on the terminal tab. This will bring up a configuration window. You will need to determine the Application UART1's com port on your computer. On Windows, this can be found using "Device Manager → Ports (COM & LPT)". Configure the terminal to use this com port number with a baud rate of 115200, a data size of 8, no parity, and 1 stop bit. Leave the encoding at its default value.



Choose terminal: Serial Terminal

Settings

Serial port: COM7

Baud rate: 115200

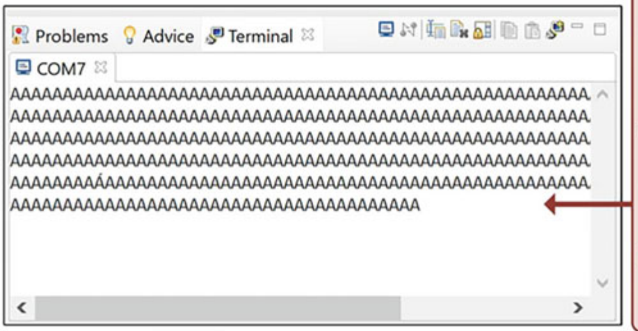
Data size: 8

Parity: None

Stop bits: 1

Encoding: Default (ISO-8859-1)

8) Plug your jumper back into the MSP430FR2355 LaunchPadTM board and observe the character 'A' arrive.



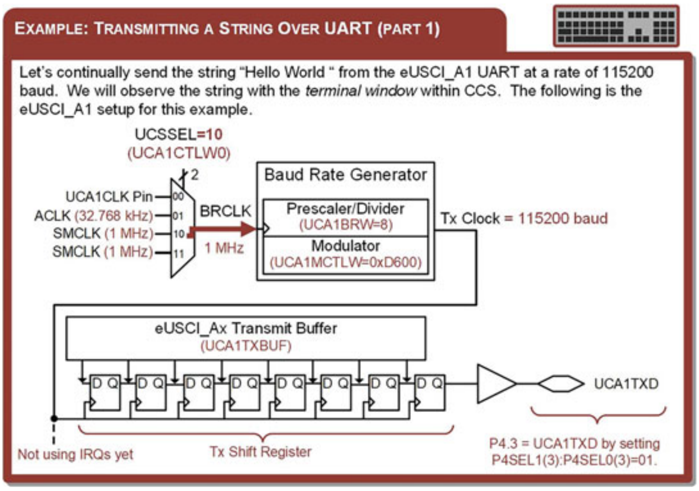
```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAÁAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Note that since the Application UART1 shares the USB cable with the debugger, the terminal window can be sluggish when in debug mode. You can "Terminate" the debugging session in CCS and your program will still be running on the MCU. Do this to observe the real-time operation of the UART data arriving.

? Did it work? You should see the character 'A' continually be printed in the terminal window.

Example 14.11
Transmitting a character over UART (part 4)

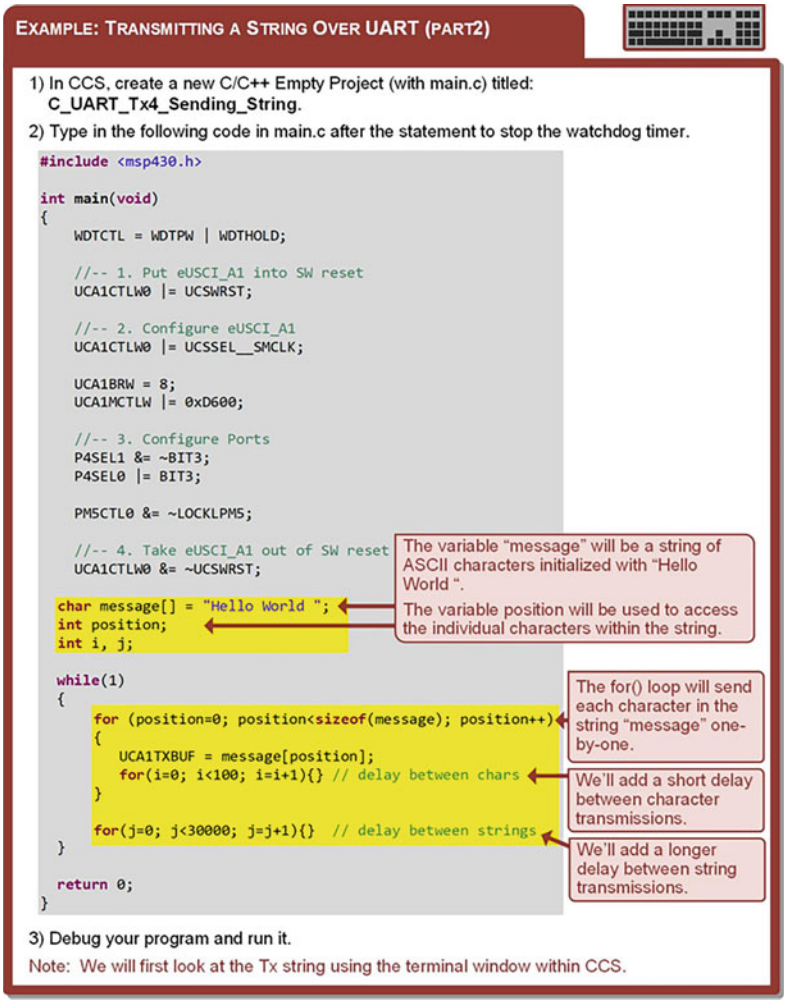
Now let's look at sending a string of characters using a for() loop. We will create a string that says "Hello World" and continually sends it over the UART Tx to be observed with a terminal. We will need to insert delay between transmits of each character in the string to avoid overwriting the Tx buffer before the prior character has been shifted out. Follow the design in Examples 14.12, 14.13, and 14.14 to see how to send strings using the MSP430FR2355 UART.



EXAMPLE: TRANSMITTING A STRING OVER UART (PART 1)

Let's continually send the string "Hello World " from the eUSCI_A1 UART at a rate of 115200 baud. We will observe the string with the *terminal window* within CCS. The following is the eUSCI_A1 setup for this example.

Example 14.12
Transmitting a string over UART (part 1)



EXAMPLE: TRANSMITTING A STRING OVER UART (PART2)

1) In CCS, create a new C/C++ Empty Project (with main.c) titled:
C_UART_Tx4_Sending_String.

2) Type in the following code in main.c after the statement to stop the watchdog timer.

```
#include <msp430.h>

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;

    //-- 1. Put eUSCI_A1 into SW reset
    UCA1CTLW0 |= UCSWRST;

    //-- 2. Configure eUSCI_A1
    UCA1CTLW0 |= UCSSEL__SMCLK;

    UCA1BRW = 8;
    UCA1MCTLW |= 0xD600;

    //-- 3. Configure Ports
    P4SEL1 &= ~BIT3;
    P4SEL0 |= BIT3;

    PM5CTL0 &= ~LOCKLPM5;

    //-- 4. Take eUSCI_A1 out of SW reset
    UCA1CTLW0 &= ~UCSWRST;

    char message[] = "Hello World ";
    int position;
    int i, j;

    while(1)
    {
        for (position=0; position<sizeof(message); position++)
        {
          UCA1TXBUF = message[position];
          for(i=0; i<100; i=i+1){} // delay between chars
        }

        for(j=0; j<30000; j=j+1){}  // delay between strings
    }

    return 0;
}
```

The variable "message" will be a string of ASCII characters initialized with "Hello World ".

The variable position will be used to access the individual characters within the string.

The for() loop will send each character in the string "message" one-by-one.

We'll add a short delay between character transmissions.

We'll add a longer delay between string transmissions.

3) Debug your program and run it.

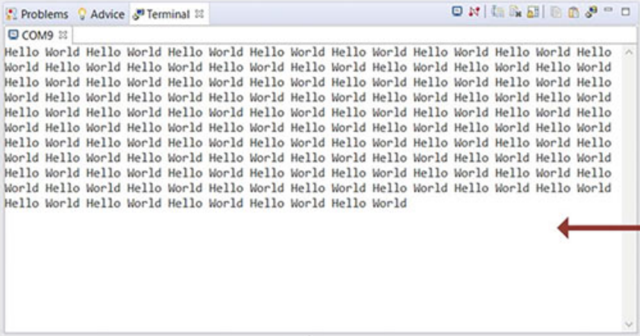
Note: We will first look at the Tx string using the terminal window within CCS.

Example 14.13
Transmitting a string over UART (part 2)

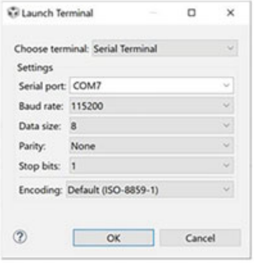
EXAMPLE: TRANSMITTING A STRING OVER UART (PART 3)

4) Launch a terminal within CCS and observe the string arrive continually. Configure the terminal to use the com port of the Application UART1 with a baud rate of 115200, a data size of 8, no parity, and 1 stop bit. Leave the encoding at its default value.

You should see "Hello World " arrive continually on the terminal.



```
Hello World Hello World Hello World Hello World Hello World Hello World Hello World Hello
World Hello World Hello World Hello World Hello World Hello World Hello World Hello World
Hello World Hello World Hello World Hello World Hello World Hello World Hello World Hello
World Hello World Hello World Hello World Hello World Hello World Hello World Hello World
Hello World Hello World Hello World Hello World Hello World Hello World Hello World Hello
World Hello World Hello World Hello World Hello World Hello World Hello World Hello World
Hello World Hello World Hello World Hello World Hello World Hello World Hello World Hello
World Hello World Hello World Hello World Hello World Hello World Hello World Hello World
Hello World Hello World Hello World Hello World Hello World Hello World Hello World Hello
World Hello World Hello World Hello World Hello World Hello World Hello World Hello World
Hello World Hello World Hello World Hello World Hello World
```



Choose terminal: Serial Terminal

Settings

Serial port: COM7

Baud rate: 115200

Data size: 8

Parity: None

Stop bits: 1

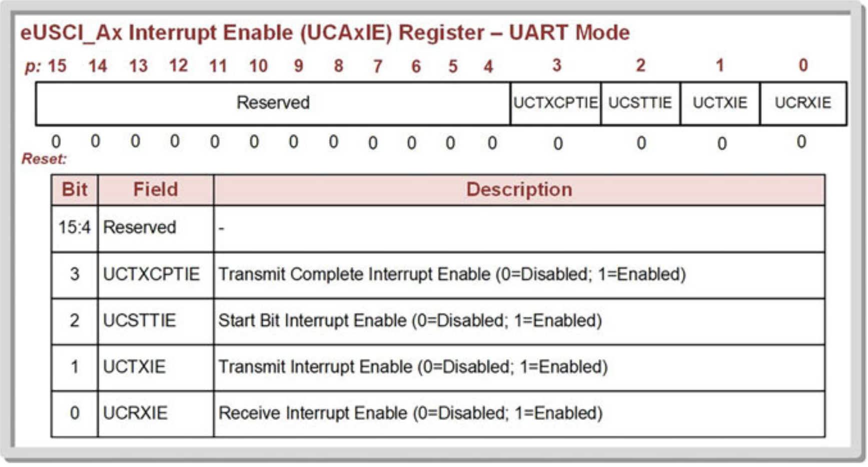
Encoding: Default (ISO-8859-1)

If it is running sluggish, terminate the debug session so that only the Application UART1 is communicating with your computer over the USB connection.

Did it work? You should see "Hello World " be continually printed to the terminal window.

Example 14.14
Transmitting a string over UART (part 3)

One of the drawbacks of the prior examples was that delay loops were needed to space out when data was put into the Tx buffer. This was to avoid overwriting the shift register with a new character before the last byte was fully shifted out. Inserting manual delay to control the timing of a peripheral is an inefficient use of the CPU and an ideal application for interrupts. MSP430 provides a variety of interrupts for the eUSCI_Ax peripheral, including two dedicated for UART transmission. The first Tx IRQ is called the *transmit interrupt* (UCTXIFG). The UCTXIFG flag is cleared when the Tx buffer data is written to. The UCTXIFG is set when the Tx buffer data has been moved to the shift register and the buffer is empty. UCTXIFG only indicates that the Tx buffer is empty, not that the last character has been fully shifted out. The second Tx IRQ is called the *transmit complete interrupt* (UCTXCPTIFG). The UCTXCPTIFG is set when the entire byte in the Tx shift register has been shifted out and it is ready for a new character. The two Tx IRQs are enabled with the UCTXIE and UCTXCPTIE bits within the UCAxIE register. The flags for the two Tx IRQs are called UCTXIFG and UCTXCPTIFG and are held in the UCAxIFG register. The eUSCI_Ax peripheral only contains one interrupt vector address for all of the interrupts within the system. In order to help identify which interrupt flag has been asserted and should be serviced first, a prioritized set of ID numbers is provided in the UCAxIV register. Figures 14.13, 14.14, and 14.15 give the details of the UCAxIE, UCAxIFG, and UCAxIV registers, respectively.

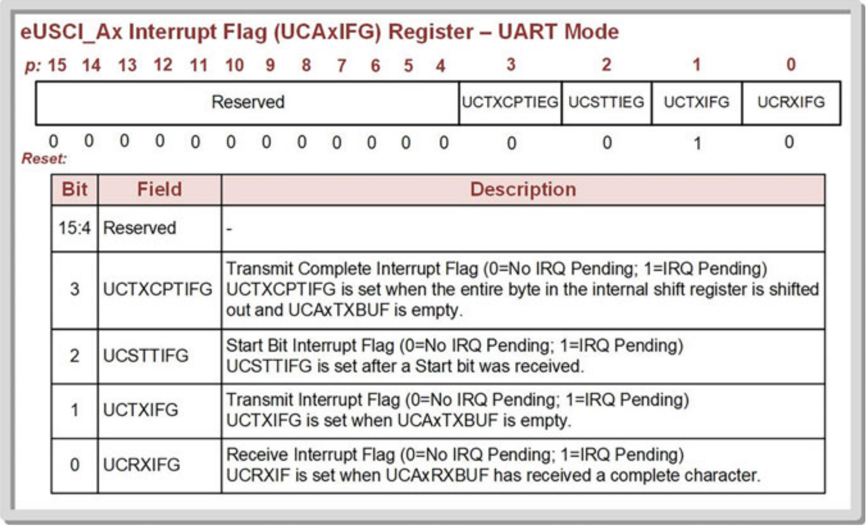


eUSCI_Ax Interrupt Enable (UCAxIE) Register – UART Mode

p:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Reserved												UCTXCPTIE	UCSTTIE	UCTXIE	UCRXIE
Reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	Field	Description
15:4	Reserved	-
3	UCTXCPTIE	Transmit Complete Interrupt Enable (0=Disabled; 1=Enabled)
2	UCSTTIE	Start Bit Interrupt Enable (0=Disabled; 1=Enabled)
1	UCTXIE	Transmit Interrupt Enable (0=Disabled; 1=Enabled)
0	UCRXIE	Receive Interrupt Enable (0=Disabled; 1=Enabled)

Fig. 14.13
eUSCI_Ax interrupt enable (UCAxIE) register

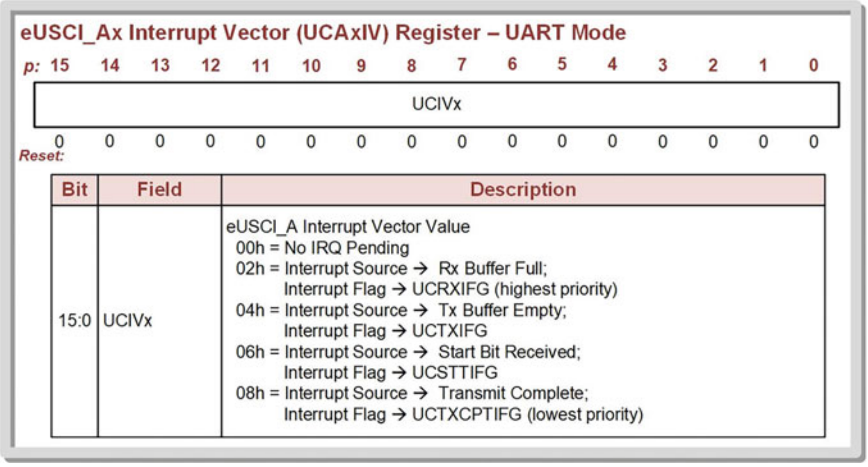


eUSCI_Ax Interrupt Flag (UCAxIFG) Register – UART Mode

p:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Reserved												UCTXCPTIEG	UCSTTIEG	UCTXIFG	UCRXIFG
Reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

Bit	Field	Description
15:4	Reserved	-
3	UCTXCPTIFG	Transmit Complete Interrupt Flag (0=No IRQ Pending; 1=IRQ Pending) UCTXCPTIFG is set when the entire byte in the internal shift register is shifted out and UCAxTXBUF is empty.
2	UCSTTIFG	Start Bit Interrupt Flag (0=No IRQ Pending; 1=IRQ Pending) UCSTTIFG is set after a Start bit was received.
1	UCTXIFG	Transmit Interrupt Flag (0=No IRQ Pending; 1=IRQ Pending) UCTXIFG is set when UCAxTXBUF is empty.
0	UCRXIFG	Receive Interrupt Flag (0=No IRQ Pending; 1=IRQ Pending) UCRXIF is set when UCAxRXBUF has received a complete character.

Fig. 14.14
eUSCI_Ax interrupt flag (UCAxIFG) register



eUSCI_Ax Interrupt Vector (UCAxIV) Register – UART Mode

p:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	UCIVx															
Reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

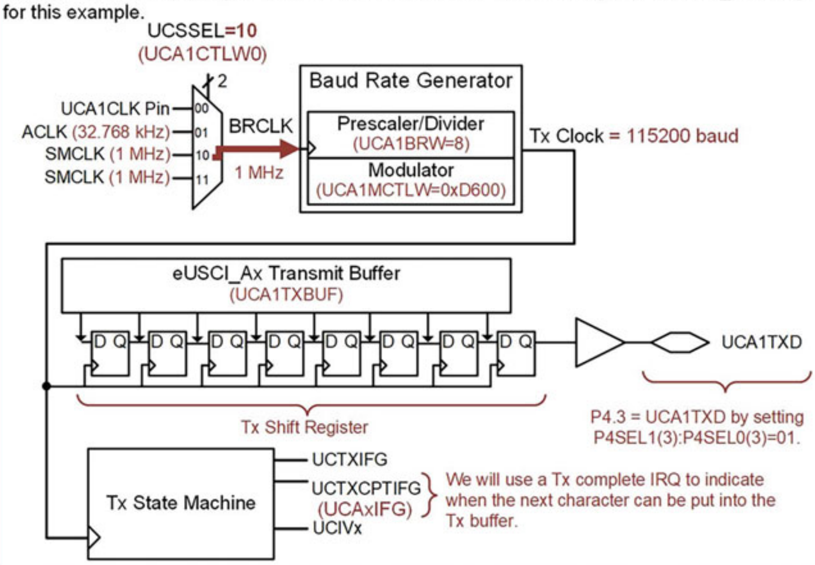
Bit	Field	Description
15:0	UCIVx	eUSCI_A Interrupt Vector Value 00h = No IRQ Pending 02h = Interrupt Source → Rx Buffer Full; Interrupt Flag → UCRXIFG (highest priority) 04h = Interrupt Source → Tx Buffer Empty; Interrupt Flag → UCTXIFG 06h = Interrupt Source → Start Bit Received; Interrupt Flag → UCSTTIFG 08h = Interrupt Source → Transmit Complete; Interrupt Flag → UCTXCPTIFG (lowest priority)

Fig. 14.15
eUSCI_Ax interrupt vector (UCAxIV) register

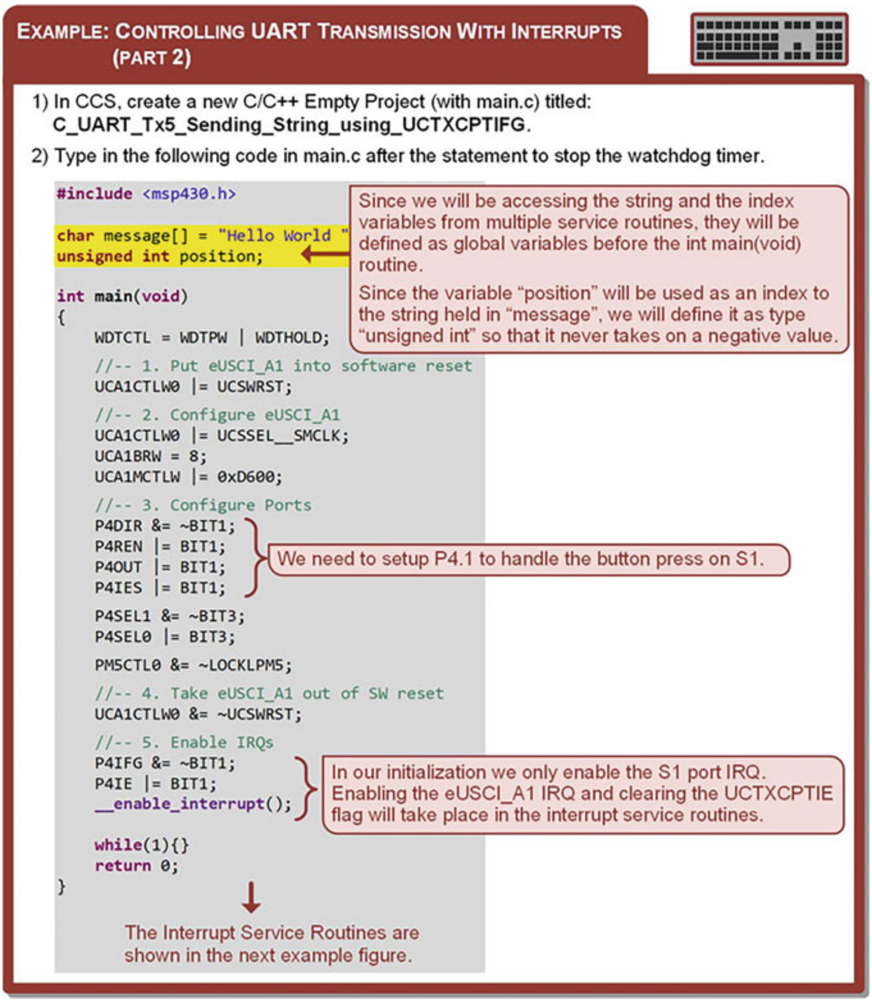
Let's now look at an example of using a UART Tx IRQ in order to track the status of the shift register when sending a string of characters. Follow the design in Examples 14.15, 14.16, and 14.17 to gain experience using the eUSCI_A1 Tx IRQs. In this example, we will send a string of characters (i.e., "Hello World"). We will begin the transmission with a button press on S1 of the LaunchPad™ board. When the S1 button is pressed, it will trigger a port interrupt. The S1 port interrupt will set the initial character position in the string that is to be sent (i.e., the index of the string). It will then enable the UCTXCPTIFG interrupt and write the first character to the Tx buffer. After this initial character is put into the Tx buffer with the UCTXCPTIFG IRQ enabled, the system will shift out the character and then assert the UCTXCPTIFG flag when complete. Once UCTXCPTIFG is asserted, it will trigger an eUSCI_A1 interrupt. Within the eUSCI_A1 service routine, we will move the current index within the string to the next location and put the next character into the Tx buffer. We will also insert functionality in the eUSCI_A1 ISR to track when the entire string has been sent and the UCTXCPTIFG interrupt can be disabled.

Example: Controlling UART Transmission With Interrupts (Part 1)

Let's design a program that will send the string "Hello World " from the eUSCI_A1 UART each time S1 is pressed on the LaunchPad^{TM} board. We will observe the string with a terminal window within CCS. We will use a Tx *complete interrupt* (UCTXCPTIFG) to control when the next character of the string is inserted into the Tx buffer. The following is the eUSCI_A1 setup for this example.



Example 14.15
Controlling UART transmission with interrupts (part 1)



Example: Controlling UART Transmission With Interrupts (Part 2)

1) In CCS, create a new C/C++ Empty Project (with main.c) titled:
C_UART_Tx5_Sending_String_using_UCTXCPTIFG.

2) Type in the following code in main.c after the statement to stop the watchdog timer.

```
#include <msp430.h>

char message[] = "Hello World "
unsigned int position;

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;
    //-- 1. Put eUSCI_A1 into software reset
    UCA1CTLW0 |= UCSWRST;
    //-- 2. Configure eUSCI_A1
    UCA1CTLW0 |= UCSSEL__SMCLK;
    UCA1BRW = 8;
    UCA1MCTLW |= 0xD600;
    //-- 3. Configure Ports
    P4DIR &= ~BIT1;
    P4REN |= BIT1;
    P4OUT |= BIT1;
    P4IES |= BIT1;
    P4SEL1 &= ~BIT3;
    P4SEL0 |= BIT3;
    PM5CTL0 &= ~LOCKLPM5;
    //-- 4. Take eUSCI_A1 out of SW reset
    UCA1CTLW0 &= ~UCSWRST;
    //-- 5. Enable IRQs
    P4IFG &= ~BIT1;
    P4IE |= BIT1;
    __enable_interrupt();

    while(1){}
    return 0;
}
```

Since we will be accessing the string and the index variables from multiple service routines, they will be defined as global variables before the int main(void) routine.

Since the variable “position” will be used as an index to the string held in “message”, we will define it as type “unsigned int” so that it never takes on a negative value.

We need to setup P4.1 to handle the button press on S1.

In our initialization we only enable the S1 port IRQ. Enabling the eUSCI_A1 IRQ and clearing the UCTXCPTIE flag will take place in the interrupt service routines.

The Interrupt Service Routines are shown in the next example figure.

Example 14.16
Controlling UART transmission with interrupts (part 2)

EXAMPLE: CONTROLLING UART TRANSMISSION WITH INTERRUPTS (PART 3)

Continue entering the ISR code for this example as shown below.

```
//-- Interrupt Service Routines ------------
#pragma vector = PORT4_VECTOR
__interrupt void ISR_Port4_S1(void)
{
    position = 0;
    UCA1IE |= UCTXCPTIE;
    UCA1IFG &= ~UCTXCPTIFG;
    UCA1TXBUF = message[position];

    P4IFG &= ~BIT1;
}
```

When S1 is pressed, the ISR will initialize the string index variable “position” to the location of the first character in the string.

Next, it enables the UCTXCPTIE IRQ and clears the UCTXCPTIFG flag.

It then puts the first character of the string into the Tx buffer.

Once the first character is placed in the Tx buffer, the UART will start shifting it out. Once the shifting is complete and the buffer is ready for the next character, the UCTXCPTIFG IRQ will trigger.

```
#pragma vector = EUSCI_A1_VECTOR
__interrupt void ISR_EUSCI_A1(void)
{
    if(position == sizeof(message)) {
        UCA1IE &= ~UCTXCPTIE;
    }
    else {
        position++;
        UCA1TXBUF = message[position];
    }
    UCA1IFG &= ~UCTXCPTIFG;
}
```

When this routine triggers for the first time, the first character has already been sent.

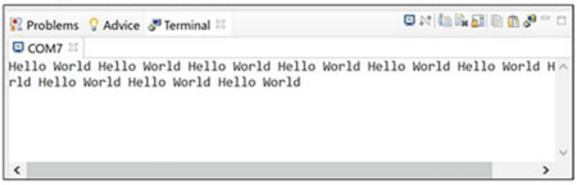
First, check whether all of the characters in the string have been sent. If they have, disable the UCTXCPTIE IRQ.

If more characters need to be sent, increment the position index and place the next character in the Tx buffer.

At the end of the ISR we need to clear the UCTXCPTIFG flag.

3) Save and debug your program. Run and then terminate your program. Your program will be running but CCS will not be sharing the USB connection between the debugger and Application UART1.

4) Launch a terminal within CCS and configure it to communicate with the MSP430FR2355 LaunchPad^{TM} board.



```
Hello World Hello World Hello World Hello World Hello World Hello World H
rld Hello World Hello World Hello World
```

Did it work? You should be able to press S1 and see “Hello World “ be printed to the terminal window.

Example 14.17
Controlling UART transmission with interrupts (part 3)

14.1.3 UART Receive on the MSP430FR2355

Now let’s look at the UART Rx functionality of the MSP430FR2355. The Rx system is very similar to the Tx system in that it contains a shift register that receives the serial data. Once the frame has been received, it is put into the Rx receiver buffer (UCAxRXBUF) in order to convert the information back to a parallel format. The same clock generator is used for both the Tx and Rx circuits and is configured using the UCAxBRW and UCAxMCTLW registers. The Rx baud rate generator is configured with the desired baud rate, and the system automatically handles creating the oversampling clock for the Rx shift register. The Rx system also contains a state machine that monitors the incoming data and creates status flags that can be used to generate interrupts. Figure 14.16 gives a general block diagram of the UART Rx system within the eUSCI_Ax peripheral.

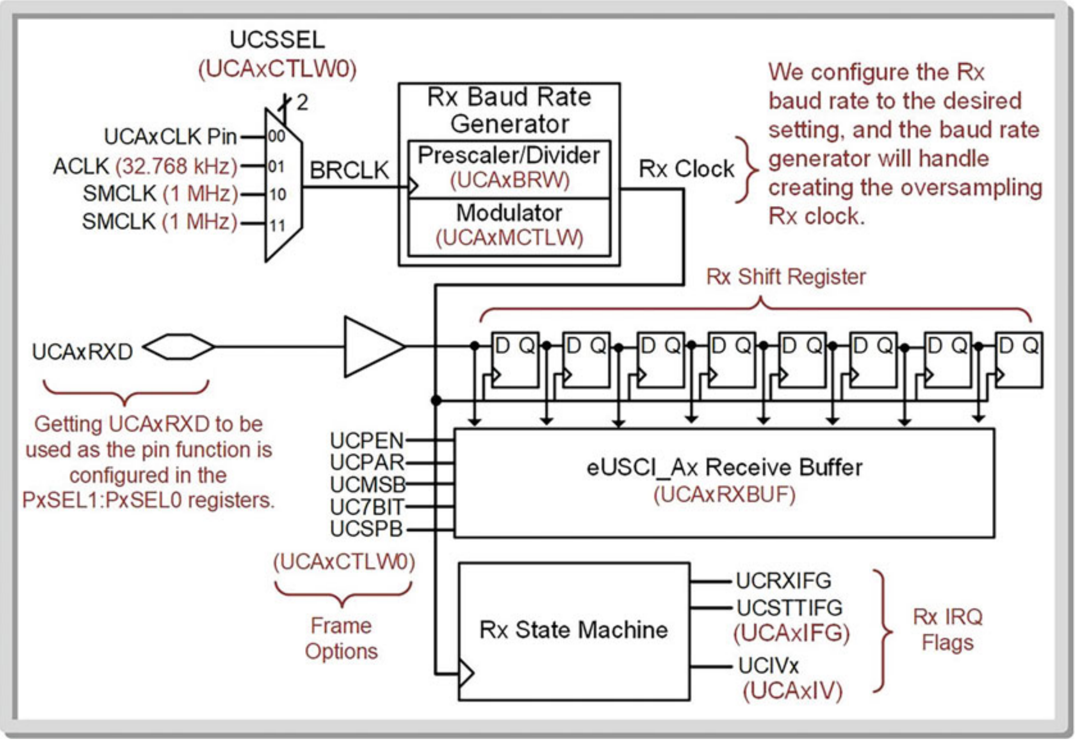
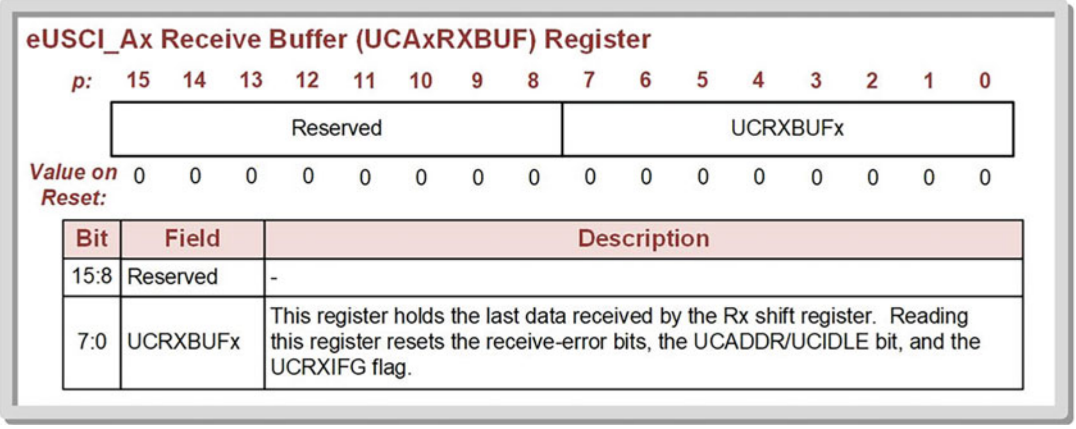


Fig. 14.16
UART Rx general block diagram of the eUSCI_Ax peripheral on the MSP430FR2355

Figure 14.17 gives the details of the uUSCI_Ax receive buffer (UCAxRXBUF).



eUSCI_Ax Receive Buffer (UCAxRXBUF) Register

p:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Reserved								UCRXBUFx							
Value on Reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	Field	Description
15:8	Reserved	-
7:0	UCRXBUFx	This register holds the last data received by the Rx shift register. Reading this register resets the receive-error bits, the UCADDR/UCIDLE bit, and the UCRXIFG flag.

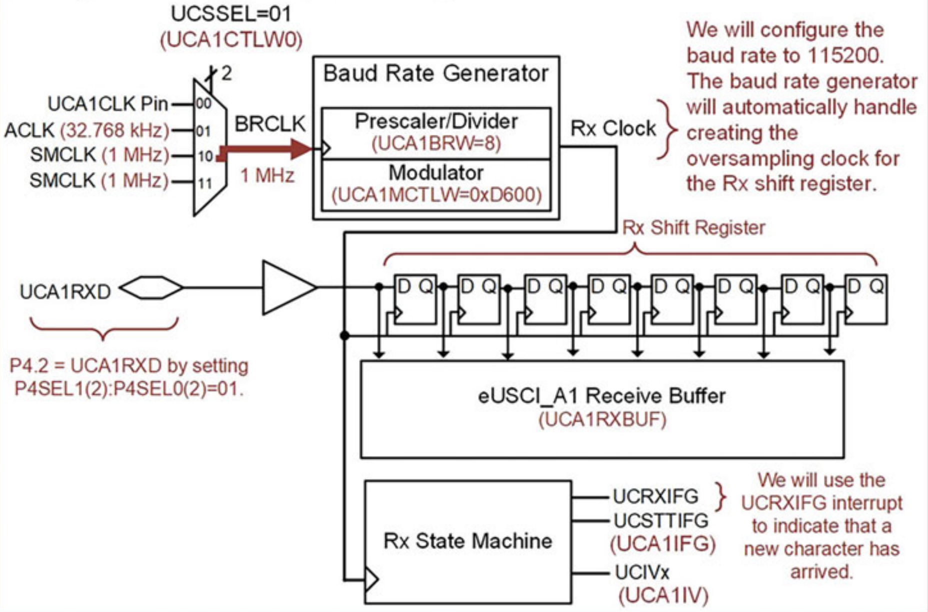
Fig. 14.17
eUSCI_Ax receive buffer (UCAxRXBUF) register

The eUSCI_Ax peripheral provides two interrupts to indicate the status of incoming characters. The *start bit interrupt* (UCSTTIFG) will trigger when the system sees a HIGH-to-LOW transition on the Rx pin indicating a new frame is being received. The *receive interrupt* (UCRXIFG) will trigger when a new character has been received and the data is available in the Rx buffer. The UCRXIFG will be cleared when the Rx buffer is read. This means that we don't need to explicitly clear the IRQ flag for the receive interrupt as we do with other peripherals. The two Rx IRQs are enabled with the UCRXIE and UCSTTIE bits within the UCAxIE register. The flags for the two Rx IRQs are called UCRXIFG and UCSTTIFG and are held in the UCAxIFG register. The UCAxIV register can be used to indicate the highest priority event when multiple IRQs are enabled since the eUSCI_Ax only has one interrupt vector.

Follow the design in Examples 14.18 and 14.19 to see how to use the eUSCI_A1 UART receive functionality. Note that the eUSCI_A1 Rx pin is connected to the Application UART1 port. In this example the program will toggle LED1 on the LaunchPad™ board anytime the character "t" is received. Characters will be sent to the LaunchPad™ board using the terminal within CCS.

EXAMPLE: RECEIVING CHARACTERS ON THE eUSCI_A1 UART (PART 1)

Let's design a program that will toggle LED1 on the LaunchPad™ board when the character 't' is received on the eUSCI_A1 UART Rx. We will send characters to the LaunchPad™ board using the terminal within CCS. The terminal handles converting keystrokes into the corresponding ASCII code. We will use the UCRXIFG interrupt to indicate when a new frame has arrived. The following is the eUSCI_A1 setup for this example.



We will configure the baud rate to 115200. The baud rate generator will automatically handle creating the oversampling clock for the Rx shift register.

P4.2 = UCA1RXD by setting P4SEL1(2):P4SEL0(2)=01.

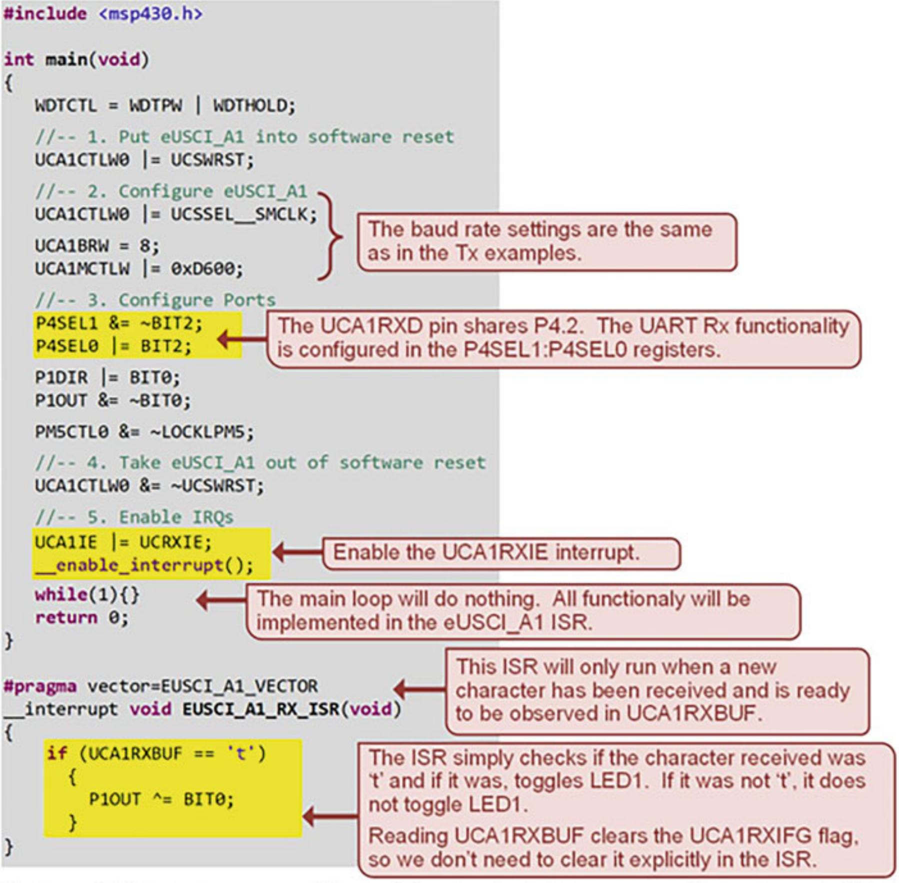
We will use the UCRXIFG interrupt to indicate that a new character has arrived.

Example 14.18
Receiving characters on the eUSCI_A1 UART (part 1)

Example: Receiving Characters on the eUSCI_A1 UART (Part 2)

1) In CCS, create a new C/C++ Empty Project (with main.c) titled: **C_UART_Rx1_Toggling_LED1**.

2) Type in the following code in main.c after the statement to stop the watchdog timer.



```c
#include <msp430.h>

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;
    //-- 1. Put eUSCI_A1 into software reset
    UCA1CTLW0 |= UCSWRST;
    //-- 2. Configure eUSCI_A1
    UCA1CTLW0 |= UCSSEL__SMCLK;
    UCA1BRW = 8;
    UCA1MCTLW |= 0xD600;
    //-- 3. Configure Ports
    P4SEL1 &= ~BIT2;
    P4SEL0 |= BIT2;
    P1DIR |= BIT0;
    P1OUT &= ~BIT0;
    PM5CTL0 &= ~LOCKLPM5;
    //-- 4. Take eUSCI_A1 out of software reset
    UCA1CTLW0 &= ~UCSWRST;
    //-- 5. Enable IRQs
    UCA1IE |= UCRXIE;
    __enable_interrupt();
    while(1){}
    return 0;
}

#pragma vector=EUSCI_A1_VECTOR
__interrupt void EUSCI_A1_RX_ISR(void)
{
    if (UCA1RXBUF == 't')
    {
        P1OUT ^= BIT0;
    }
}
```

The baud rate settings are the same as in the Tx examples.

The UCA1RXD pin shares P4.2. The UART Rx functionality is configured in the P4SEL1:P4SEL0 registers.

Enable the UCA1RXIE interrupt.

The main loop will do nothing. All functionaly will be implemented in the eUSCI_A1 ISR.

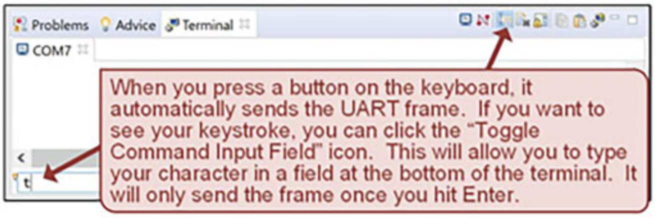
This ISR will only run when a new character has been received and is ready to be observed in UCA1RXBUF.

The ISR simply checks if the character received was 't' and if it was, toggles LED1. If it was not 't', it does not toggle LED1.

Reading UCA1RXBUF clears the UCA1RXIFG flag, so we don't need to clear it explicitly in the ISR.

3) Save and debug your program. Run and then terminate your program. Your program will be running but CCS will not be sharing the USB connection between the debugger and Application UART1.

4) Launch a terminal within CCS and configure it to communicate with the MSP430FR2355 LaunchPad^{TM} board.



When you press a button on the keyboard, it automatically sends the UART frame. If you want to see your keystroke, you can click the "Toggle Command Input Field" icon. This will allow you to type your character in a field at the bottom of the terminal. It will only send the frame once you hit Enter.

?

Did it work? Initially you should be able to just press 't' on the keyboard and LED1 will toggle. Make sure the terminal window is active by clicking in it. Next try the "Toggle Command Input Field" option.

Example 14.19
Receiving characters on the eUSCI_A1 UART (part 2)

CONCEPT CHECK

CC14.1 The UART receiver oversampling scheme seems like a very complicated way to recover the transmitted signal. Why not just add another wire and send a clock?

A) The original intent of the UART was to implement a serial communication scheme with the least amount of wires. A half-duplex UART can be implemented with just one wire. Adding an extra pin to a computer system today may not add much cost, but the cost of cables in long distance communication links can quickly become cost prohibitive.

B) If we added a clock, UART would be too similar to SPI and confuse people.

C) If we added a clock, we wouldn't be able to use the term "baud".

D) If we added a clock, we wouldn't be able to run at the slower speeds that UART supports.

14.2 Serial Peripheral Interface (SPI)

14.2.1 The SPI Protocol

The SPI protocol transmits serial data between devices using a shared clock and dedicated lines for data out and data in. This protocol uses more pins than a UART (three at a minimum) but can achieve higher data rates due to the synchronous nature of the link. A SPI system uses the concept of a *master* and a *slave* when setting up the link. The master is the device that generates the clock that all devices use to transfer information. The master clock is called the *serial clock* (SCLK). The data lines are denoted *slave in, master out* (SIMO) and *slave out, master in* (SOMI). Some devices exchange the order of the letters and instead use MOSI and MISO. The MSP430 documentation uses SIMO and SOMI, so this book will also use that notation. Figure 14.18 shows a typical SPI master/slave configuration in its simplest form known as *three-wire mode*.

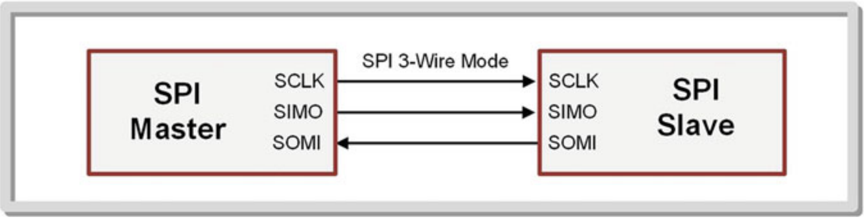


Fig. 14.18
Single master/slave SPI configuration (three-wire mode)

The default SPI data frame is 8 bits long with the LSB being sent first. The SPI Tx and Rx devices use shift registers to transfer the serial bit stream. When transmitting, the data is shifted out on the rising edge of SCLK. When receiving, the data is shifted in on the falling edge of SCLK. When not transmitting, SCLK is held in an idle state. This allows the clock edge used by the receiver to be centered within the bit period of the data bits. Figure 14.19 shows the clocking scheme for a SPI system. The MSP430FR2355 also provides framing options such as MSB first, the clocking polarity, and 7- vs. 8-bit size in addition to clocking options such as the edge polarity and the state when idle.

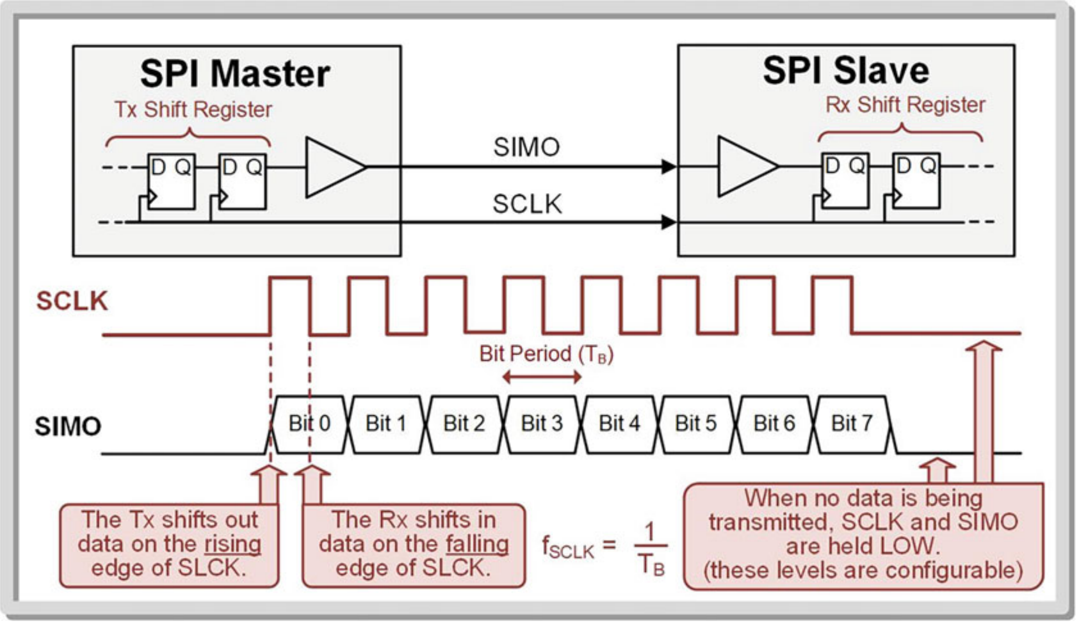

SPI Master
Tx Shift Register
D Q
D Q
SIMO
SCLK
SPI Slave
Rx Shift Register
D Q
D Q
SCLK
Bit Period (T_B)
SIMO
Bit 0
Bit 1
Bit 2
Bit 3
Bit 4
Bit 5
Bit 6
Bit 7
The Tx shifts out data on the rising edge of SLCK.
The Rx shifts in data on the falling edge of SLCK.
$f_{SCLK} = \frac{1}{T_B}$
When no data is being transmitted, SCLK and SIMO are held LOW. (these levels are configurable)

Fig. 14.19
SPI clocking scheme

When multiple slaves are used in the system, a *slave transmit enable* (STE) line can be generated by the master in order to dictate which slave is being communicated with. This is also called the *slave select* (SS) in some SPI devices. There are different topologies that can be used when communicating with multiple slaves. The first is a *bus configuration* in which the SIMO and SOMI lines are all shared between the master and slaves. Each slave is assigned its own STE. Only one slave is active at any given time. Once more than one STE is needed, the MSP430FR2355 user's guide recommends using port pins to create dedicated STEs. The second configuration is called *daisy-chained* where devices are wired to form a single, continuous data loop among all the SPI devices. In the daisy-chained configuration, the master counts the number of clocks that have occurred in order to track where in the loop the current data resides. Figure 14.20 shows SPI topologies that use STEs.

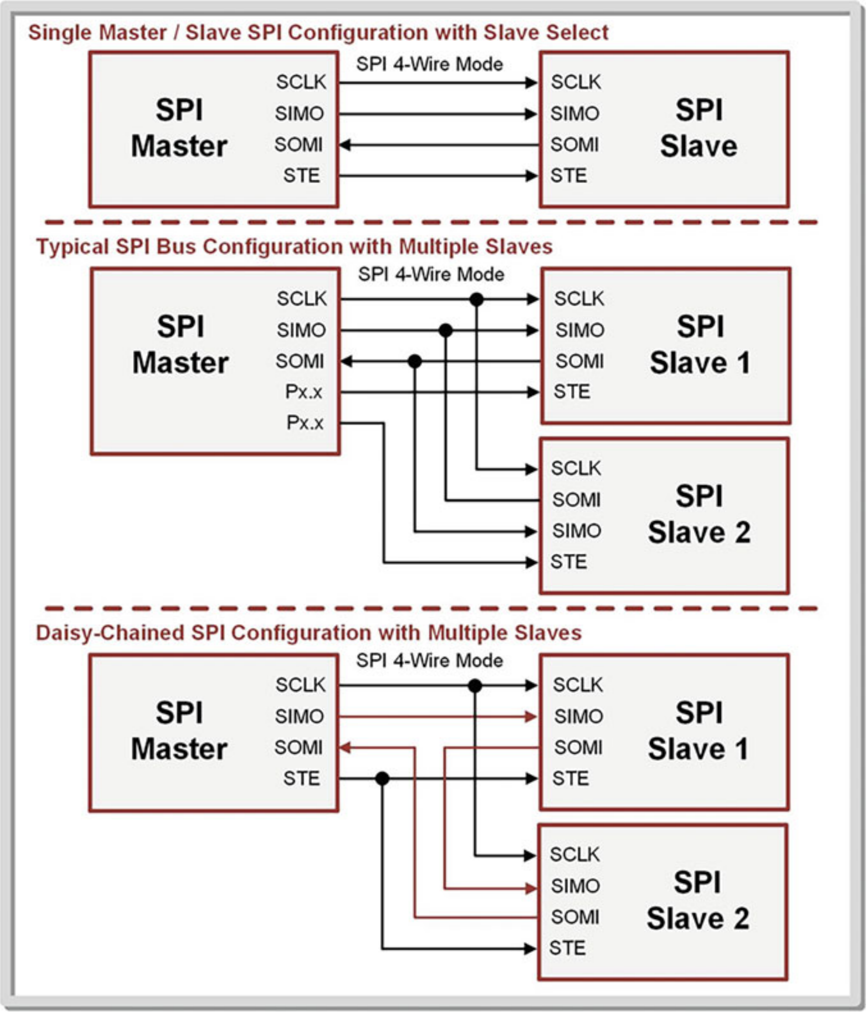
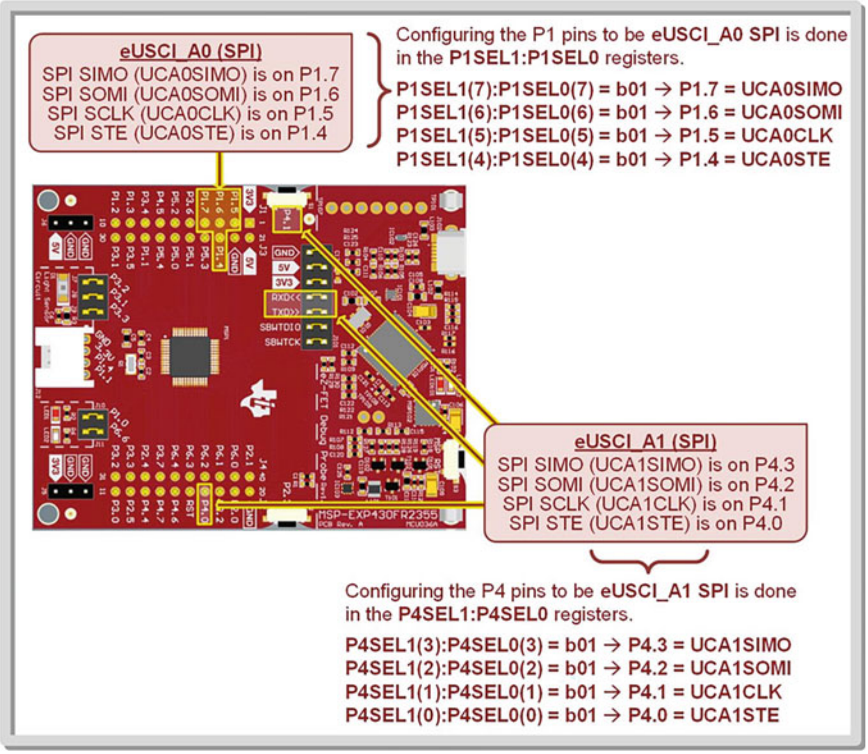


Fig. 14.20
SPI topologies that use slave transfer enable (STE) lines

The MSP430FR2355 provides four SPI modules, one on each of the serial communication peripherals (eUSCI_A0, eUSCI_A1, eUSCI_B0, and eUSCI_B1). The eUSCI_Ax peripherals provide either UART or SPI while the eUSCI_Bx peripherals provide either I2C or SPI. The PxSEL1:PxSEL0 registers are used to select which serial peripheral drives the pins on the MSP430FR2355 MCU. Figure 14.21 shows the location of the eUSCI_A0 and eUSCI_A1 SPI pins on the LaunchPad™ board.



eUSCI_A0 (SPI)
SPI SIMO (UCA0SIMO) is on P1.7
SPI SOMI (UCA0SOMI) is on P1.6
SPI SCLK (UCA0CLK) is on P1.5
SPI STE (UCA0STE) is on P1.4

Configuring the P1 pins to be **eUSCI_A0 SPI** is done in the **P1SEL1:P1SEL0** registers.

P1SEL1(7):P1SEL0(7) = b01 → P1.7 = UCA0SIMO
P1SEL1(6):P1SEL0(6) = b01 → P1.6 = UCA0SOMI
P1SEL1(5):P1SEL0(5) = b01 → P1.5 = UCA0CLK
P1SEL1(4):P1SEL0(4) = b01 → P1.4 = UCA0STE

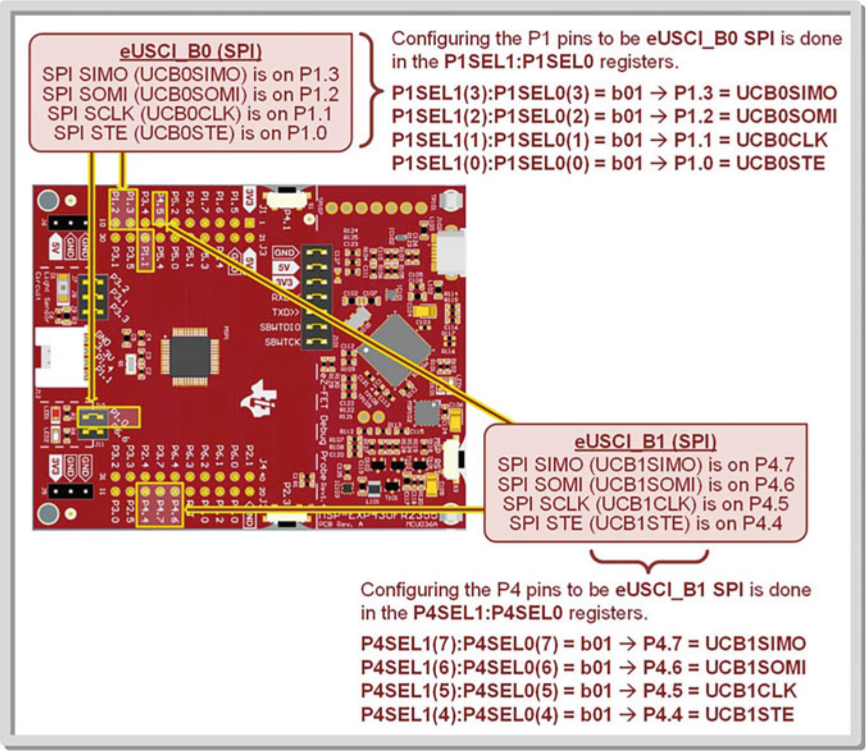
eUSCI_A1 (SPI)
SPI SIMO (UCA1SIMO) is on P4.3
SPI SOMI (UCA1SOMI) is on P4.2
SPI SCLK (UCA1CLK) is on P4.1
SPI STE (UCA1STE) is on P4.0

Configuring the P4 pins to be **eUSCI_A1 SPI** is done in the **P4SEL1:P4SEL0** registers.

P4SEL1(3):P4SEL0(3) = b01 → P4.3 = UCA1SIMO
P4SEL1(2):P4SEL0(2) = b01 → P4.2 = UCA1SOMI
P4SEL1(1):P4SEL0(1) = b01 → P4.1 = UCA1CLK
P4SEL1(0):P4SEL0(0) = b01 → P4.0 = UCA1STE

Fig. 14.21
eUSCI_Ax SPI pins on MSP430FR2355 LaunchPad™ board

Figure 14.22 shows the location of the eUSCI_B0 and eUSCI_B1 SPI pins on the LaunchPad™ board.



eUSCI_B0 (SPI)
SPI SIMO (UCB0SIMO) is on P1.3
SPI SOMI (UCB0SOMI) is on P1.2
SPI SCLK (UCB0CLK) is on P1.1
SPI STE (UCB0STE) is on P1.0

Configuring the P1 pins to be **eUSCI_B0 SPI** is done in the **P1SEL1:P1SEL0** registers.

P1SEL1(3):P1SEL0(3) = b01 → P1.3 = UCB0SIMO
P1SEL1(2):P1SEL0(2) = b01 → P1.2 = UCB0SOMI
P1SEL1(1):P1SEL0(1) = b01 → P1.1 = UCB0CLK
P1SEL1(0):P1SEL0(0) = b01 → P1.0 = UCB0STE

eUSCI_B1 (SPI)
SPI SIMO (UCB1SIMO) is on P4.7
SPI SOMI (UCB1SOMI) is on P4.6
SPI SCLK (UCB1CLK) is on P4.5
SPI STE (UCB1STE) is on P4.4

Configuring the P4 pins to be **eUSCI_B1 SPI** is done in the **P4SEL1:P4SEL0** registers.

P4SEL1(7):P4SEL0(7) = b01 → P4.7 = UCB1SIMO
P4SEL1(6):P4SEL0(6) = b01 → P4.6 = UCB1SOMI
P4SEL1(5):P4SEL0(5) = b01 → P4.5 = UCB1CLK
P4SEL1(4):P4SEL0(4) = b01 → P4.4 = UCB1STE

Fig. 14.22
eUSCI_Bx SPI pins on MSP430FR2355 LaunchPad™ board

14.2.2 SPI Master Operation on the MSP430FR2355

Let's start learning about the SPI protocol by looking at the transmit capabilities of the MSP430FR2355's eUSCI_Ax in master mode. Figure 14.23 shows a general block diagram of the SPI Tx component of the eUSCI_Ax peripherals on the MSP430FR2355.

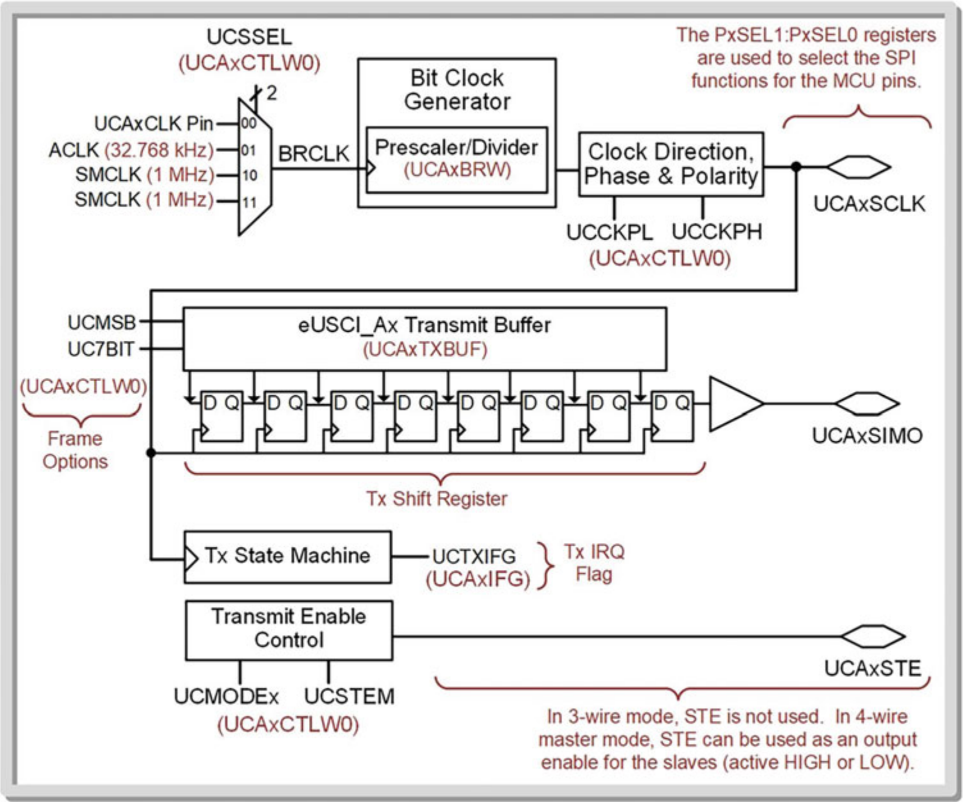


Fig. 14.23
SPI Tx general block diagram of the eUSCI_Ax peripheral on the MSP430FR2355

The basic concept of operation of the SPI system is that we first configure it to have the desired bit rate and frame characteristics. Then we store the data to be transmitted into a Tx buffer, and a shift register automatically sends the data out over the SIMO pin in a serial pattern. The master sends out eight transitions on SCLK corresponding to each bit that was sent on SIMO. In three-pin mode, the STE is not used. In four-pin master mode, the STE pin can be configured to be either an active high or low output enable for the slaves. The SPI transmit system can also provide one interrupt called *transmit interrupt* (UCTXIFG) that is used to indicate when the Tx buffer is empty. The SPI peripherals are configured using a variety of registers that are shared with the UART (or I2C) peripheral. In SPI mode, some of the bit fields within the registers take on different functionality. This is most obvious in the UCAxCTLW0 register in which many of the UART settings aren't applicable to the SPI protocol. The complete list of SPI configuration registers is below.

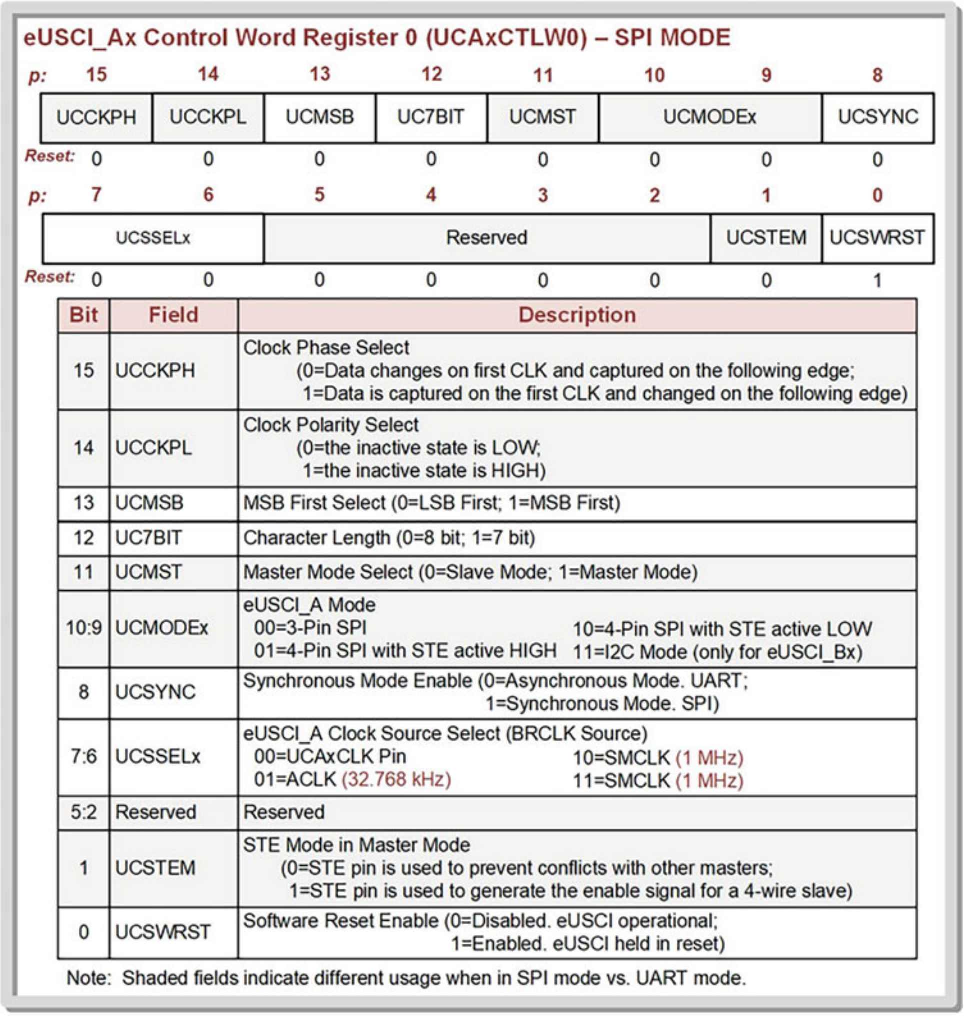
- *eUSCI_Ax control word 0* (UCAxCTLW0) – has different bit fields when in SPI mode.
- *eUSCI_Ax bit rate control word* (UCAxBRW) – same function as in UART mode.
- *eUSCI_Ax status* (UCAxSTATW) – has different bit fields when in SPI mode.
- *eUSCI_Ax receive buffer* (UCAxRXBUF) – same function as in UART mode.
- *eUSCI_Ax transmit buffer* (UCAxTXBUF) – same function as in UART mode.
- *eUSCI_Ax interrupt enable* (UCAxIE) – only has TXIE and RXIE fields.
- *eUSCI_Ax interrupt flag* (UCAxIFG) – only has TXIFG and RXIFG fields.
- *eUSCI_Ax interrupt vector* (UCAxIV) – only has codes for TXIFG and RXIFG IRQs.

The recommended order of steps from the MSP430 user's guide to set up the SPI peripheral is:

1. Set the UCSWRST bit in the UCAxCTLW0 configuration register to put the eUSCI_Ax peripheral into software reset.
2. Initialize all eUSCI_Ax configuration registers.
3. Configure ports.
4. Clear UCSWRST to take the eUSCI_Ax peripheral out of reset.
5. Enable interrupts (optional) in the UCAxIE configuration register (UCRXIE or UCTXIE).

14.2.2.1 Transmitting Data as the SPI Master

As with the UART, the first step in setting up the SPI peripheral is to put the system into software reset to avoid erroneous data from being transmitted during setup. This is done by setting the UCSWRST bit in the eUSCI_Ax control word 0 (UCAxCTLW0) register. Upon reset, UCSWRST = 1, so eUSCI_Ax is in software reset by default; however, it is good practice to explicitly set this bit in order to ensure the system is disabled. Figure 14.24 gives the details of the UCAxCTLW0 register where the UCSWRST resides. This is the same register as is used to configure the UART; however, many of the fields are used for different settings when configuring the peripheral as a SPI.



eUSCI_Ax Control Word Register 0 (UCAxCTLW0) – SPI MODE

p:	15	14	13	12	11	10	9	8
	UCCKPH	UCCKPL	UCMSB	UC7BIT	UCMST	UCMODEx		UCSYNC
Reset:	0	0	0	0	0	0	0	0

p:	7	6	5	4	3	2	1	0
	UCSSELx		Reserved				UCSTEM	UCSWRST
Reset:	0	0	0	0	0	0	0	1

Bit	Field	Description
15	UCCKPH	Clock Phase Select (0=Data changes on first CLK and captured on the following edge; 1=Data is captured on the first CLK and changed on the following edge)
14	UCCKPL	Clock Polarity Select (0=the inactive state is LOW; 1=the inactive state is HIGH)
13	UCMSB	MSB First Select (0=LSB First; 1=MSB First)
12	UC7BIT	Character Length (0=8 bit; 1=7 bit)
11	UCMST	Master Mode Select (0=Slave Mode; 1=Master Mode)
10:9	UCMODEx	eUSCI_A Mode 00=3-Pin SPI 10=4-Pin SPI with STE active LOW 01=4-Pin SPI with STE active HIGH 11=I2C Mode (only for eUSCI_Bx)
8	UCSYNC	Synchronous Mode Enable (0=Asynchronous Mode. UART; 1=Synchronous Mode. SPI)
7:6	UCSSELx	eUSCI_A Clock Source Select (BRCLK Source) 00=UCAxCLK Pin 10=SMCLK (1 MHz) 01=ACLK (32.768 kHz) 11=SMCLK (1 MHz)
5:2	Reserved	Reserved
1	UCSTEM	STE Mode in Master Mode (0=STE pin is used to prevent conflicts with other masters; 1=STE pin is used to generate the enable signal for a 4-wire slave)
0	UCSWRST	Software Reset Enable (0=Disabled. eUSCI operational; 1=Enabled. eUSCI held in reset)

Note: Shaded fields indicate different usage when in SPI mode vs. UART mode.

Fig. 14.24
eUSCI_Ax control word register 0 (UCAxCTLW0): SPI mode

The next configuration step that is handled in the UCAxCTLW0 register is selecting the clock for the eUSCI_Ax peripheral (i.e., BRCLK). This is done using the UCSSELx bits. On the MSP430FR2355, the default clock source is the external UCAxCLK pin on the MCU. We are given the choice of selecting either ACLK (UCSSELx = 01) or SMCLK (UCSSELx = 10 or 11) as internal clock sources for the eUSCI_Ax. On the MSP430FR2355 LaunchPad™ board, we will always use either ACLK or SMCLK for the source.

The next critical setting for the SPI is its bit rate. Setting the bit rate for SPI is much simpler than for a UART due to its synchronous nature. The only setting that alters the rate of the clock is in the UCAxBRW register. The value that is stored into UCAxBRW will divide the incoming clock source (i.e., BRCLK) to produce the SCLK that will be used by the master and all slaves. If UCAxBRW is left at its default value of x0000, the bit clock will simply be BRCLK. On the MSP430FR2355 the fastest clock available is SMCLK=1 MHz, which sets the fastest bit rate that the SPI system can achieve on this MCU. Using the UCAxBRW register to divide down the BRCLK gives a nearly limitless number of bit rates that are slower than 1 MHz. Note that the UCAxBRW register is the same register that is used in the UART baud rate generation circuit as the prescaler/divider of the incoming clock. This register is used in the same way for SPI as a clock prescaler/divider.

There are also two options for the way that data is clocked in the SPI system. The UCCKPH bit dictates the relative phase between the data and SCLK when sending or receiving data. When UCCKPH = 0, data is changed on the first edge of SCLK and captured on the following edge (default). When UCCKPH = 1, data is *captured* on the first edge of SCLK and changed on the following edge. The UCCKPL bit dictates the polarity of the clock when inactive. Recall that when SCLK is not transmitting, it remains at a constant logic level. If UCCKPL = 0, the inactive state is LOW (default). If UCCKPL = 1, the inactive state is HIGH.

Just as with the UART interface, many of the SPI peripheral settings are configured in the UCAxCTLW0 register. First, the UCSYNC bit sets the peripheral to either UART (UCSYNC = 0 for Asynchronous) or SPI (UCSYNC = 1 for Synchronous) mode. Next, the UCMST bit sets whether the SPI peripheral is in master (UCMST = 1) or slave (UCMST = 0) mode. The framing options for the SPI data transmission are also set in UCAxCTLW0, including sending MSB vs. LSB first (UCMSB) and 7-bit vs. 8-bit data length (UC7BIT). The UCMODEx bits dictate whether the peripheral is set up as a three-wire SPI, four-wire SPI with active HIGH STE, four-wire SPI with active LOW STE, or as an I2C (only for eUSCI_Bx). If the SPI system is set up to use STE, its functionality is further configured using the UCSTEM bit. If UCSTEM = 1 and four-wire mode is enabled, the STE bit is configured as an output for use in master mode. If UCSTEM = 0 and four-wire mode is enabled, the STE bit is configured as an input for use in slave mode.

At this point, we have covered all of the eUSCI_Ax settings to configure the SPI system on eUSCI_Ax. The next step is to configure the ports on the MSP430FR2355 to use the SPI SCLK, SIMO, SOMI, and STE (optionally). Just as in the UART, this is done using the PxSEL1 and PxSEL0 configuration registers. The settings for these port configuration registers were given in Table 14.2 in Sect. 14.1.2.

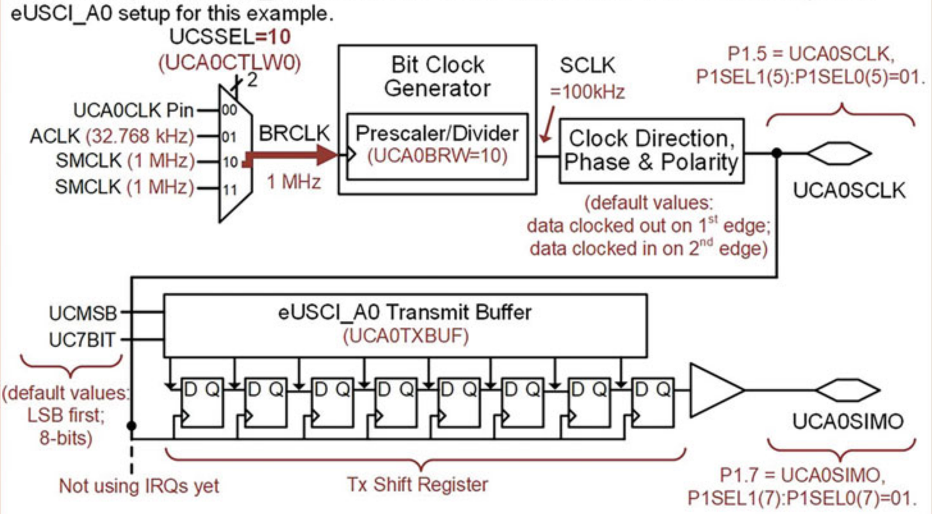
Now that the SPI ports have been configured, the next step is to take the eUSCI_Ax peripheral out of software reset by clearing the UCSWRST bit in the UCAxCTLW0 register.

Once taken out of software reset, the eUSCI_Ax SPI module is enabled and will wait in an idle state until information is ready to be transmitted. While in the idle state, no SCLK is generated. A Tx is initiated by the master by writing a byte of data to the eUSCI_Ax Transmit Buffer (UCAxTXBUF). When this occurs, the data in UCAxTXBUF is moved into the Tx shift register, and the bit rate generator produces eight clocks to shift out the data over SIMO (when in 8-bit mode). The UCTXIFG flag provides the status of the transmission. When UCTXIFG = 0, data is being shifted out and new data should not be written to the Tx buffer. When UCTXIFG = 1, new data can be written to the Tx buffer. The UCTXIFG flag resides in the UCAxIFG register, just as in the UART setup.

Let's start programming the SPI peripheral on the LaunchPad™ board by first looking at sending out an 8-bit value that is continually stored to the transmit buffer separated by delay loops. Follow the design in Examples 14.20, 14.21, 14.22, and 14.23 to gain experience using the MSP430FR2355 SPI transmit in master mode.

Example: Transmitting a Byte From a SPI Master (Part 1)

Let's configure the eUSCI_A0 peripheral as a SPI master and continually transmit the value **0x4D.** We will observe SCLK and SIMO using both an oscilloscope and a logic analyzer. We will use SMCLK as the source clock (BRCLK) and divide it down by 10 to produce an SCLK=100 kHz. We will configure the SPI peripheral in 3-wire mode using the default framing options (8-bit and LSB first). For eUSCI_A0, SCLK is on P1.5 and SIMO is on P1.7. The following is the eUSCI_A0 setup for this example.

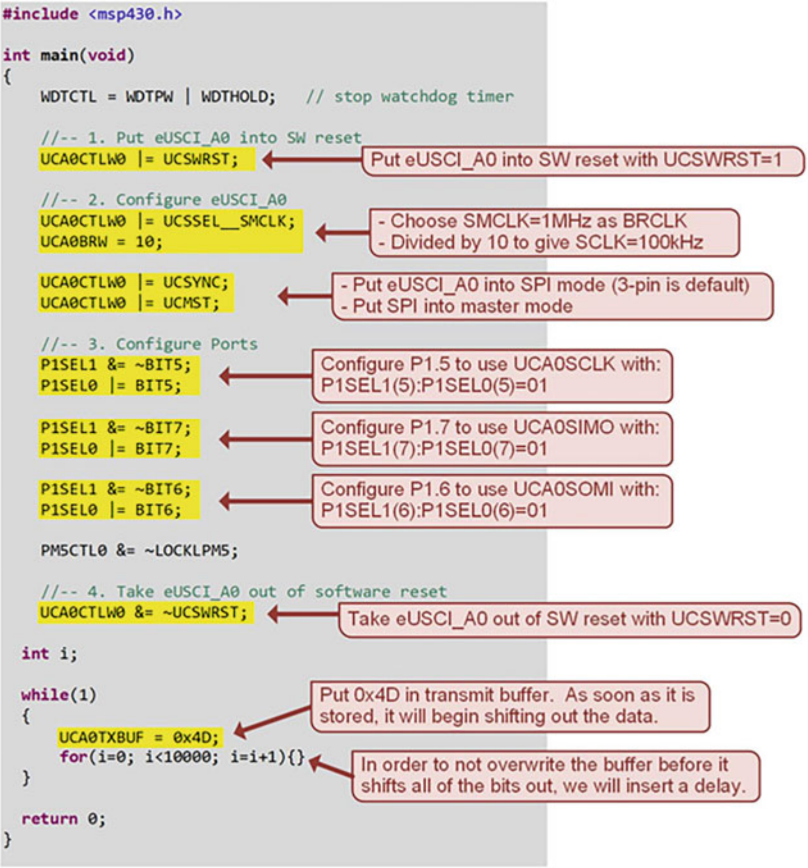


Example 14.20
Transmitting a byte from a SPI master (part 1)

Example: Transmitting a Byte From a SPI Master (Part 2)

1) In CCS, create a new C/C++ Empty Project (with main.c) titled:
C_SPI_Tx1_Hex_on_A0.

2) Type in the following code in main.c after the statement to stop the watchdog timer.



```
#include <msp430.h>

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

    //-- 1. Put eUSCI_A0 into SW reset
    UCA0CTLW0 |= UCSWRST;

    //-- 2. Configure eUSCI_A0
    UCA0CTLW0 |= UCSSEL__SMCLK;
    UCA0BRW = 10;

    UCA0CTLW0 |= UCSYNC;
    UCA0CTLW0 |= UCMST;

    //-- 3. Configure Ports
    P1SEL1 &= ~BIT5;
    P1SEL0 |= BIT5;

    P1SEL1 &= ~BIT7;
    P1SEL0 |= BIT7;

    P1SEL1 &= ~BIT6;
    P1SEL0 |= BIT6;

    PM5CTL0 &= ~LOCKLPM5;

    //-- 4. Take eUSCI_A0 out of software reset
    UCA0CTLW0 &= ~UCSWRST;

  int i;

  while(1)
  {
     UCA0TXBUF = 0x4D;
     for(i=0; i<10000; i=i+1){}
  }

  return 0;
}
```

Put eUSCI_A0 into SW reset with UCSWRST=1

- Choose SMCLK=1MHz as BRCLK
- Divided by 10 to give SCLK=100kHz

- Put eUSCI_A0 into SPI mode (3-pin is default)
- Put SPI into master mode

Configure P1.5 to use UCA0SCLK with: P1SEL1(5):P1SEL0(5)=01

Configure P1.7 to use UCA0SIMO with: P1SEL1(7):P1SEL0(7)=01

Configure P1.6 to use UCA0SOMI with: P1SEL1(6):P1SEL0(6)=01

Take eUSCI_A0 out of SW reset with UCSWRST=0

Put 0x4D in transmit buffer. As soon as it is stored, it will begin shifting out the data.

In order to not overwrite the buffer before it shifts all of the bits out, we will insert a delay.

3) Debug your program and run it.

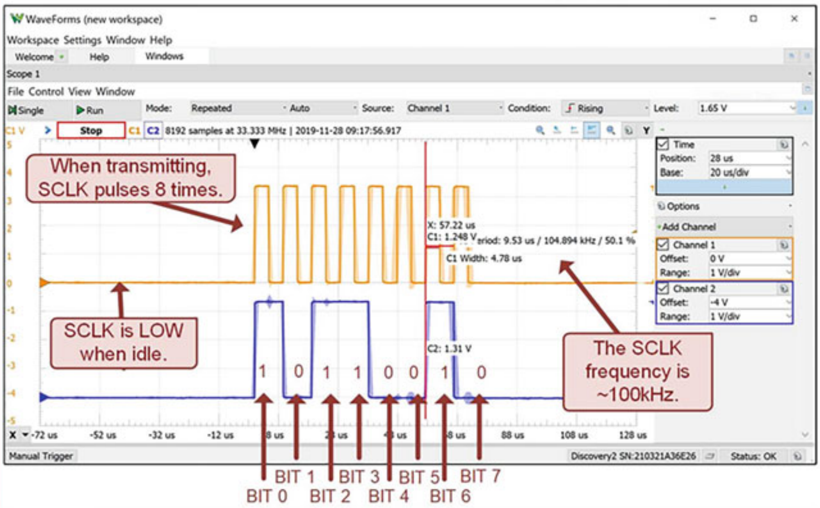
Note: We will first look at the SPI output signals with an oscilloscope to verify its proper operation.

Example 14.21
Transmitting a byte from a SPI master (part 2)

Example: Transmitting a Byte From a SPI Master (Part 3)

4) Observe the SPI outputs on the LaunchPad™ board with an oscilloscope. Probe SCLK on the P1.5 pin of the J1 header. Probe SIMO on the P1.7 pin of the J1 header.

5) Configure the oscilloscope waveform to center the SPI frame. Measure the bit period of one of the clock pulses. You should see the following waveforms on your oscilloscope.



Mode: Repeated · Auto · Source: Channel 1 · Condition: Rising · Level: 1.65 V

8192 samples at 33.333 MHz | 2019-11-28 09:17:56.917

X: 57.22 us
C1: 1.248 V
Period: 9.53 us / 104.894 kHz / 50.1 %
C1 Width: 4.78 us
C2: 1.31 V

Time
Position: 28 us
Base: 20 us/div

Channel 1
Offset: 0 V
Range: 1 V/div

Channel 2
Offset: -4 V
Range: 1 V/div

Manual Trigger

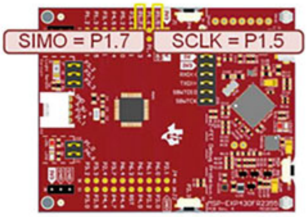
Discovery2 SN:210321A36E26 Status: OK

? Did it work? Are you able to see the same SPI clock and data shown above? Notice that the sequence of bits received was 0b10110010. But remember that SPI sends the LSB first. Putting this in order from MSB we get: 0b01001101=0x4D.

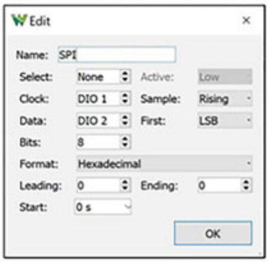
Example 14.22
Transmitting a byte from a SPI master (part 3)

Example: Transmitting a Byte From a SPI Master (Part 4)

6) Now observe the SPI outputs on the LaunchPad™ board with a *logic analyzer*. A logic analyzer is only able to display the digital values of the signal, but provides more channels. Logic analyzers are also able to display the data packet values of common protocols such as SPI. Probe SCLK on the P1.5 pin of the J1 header. Probe SIMO on the P1.7 pin of the J1 header.



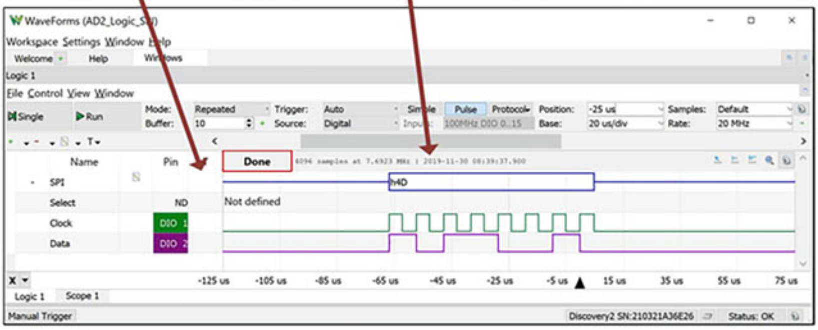
5) Configure the logic analyzer waveform to center the SPI frame. Setup the trigger of the logic analyzer to watch for multiple transitions on SCLK. Setup the SPI protocol feature of the logic analyzer (i.e., LSB first, the signals for SCLK and SIMO). You should see the following waveforms on your logic analyzer.



Name: SPI
Select: None Active: Low
Clock: DIO 1 Sample: Rising
Data: DIO 2 First: LSB
Bits: 8
Format: Hexadecimal
Leading: 0 Ending: 0
Start: 0 s

Notice this is the same information as observed on the oscilloscope with the exception that only the digital values (i.e., HIGH or LOW) are displayed.

The logic analyzer can display the SPI data packet in its original order (i.e., MSB first).



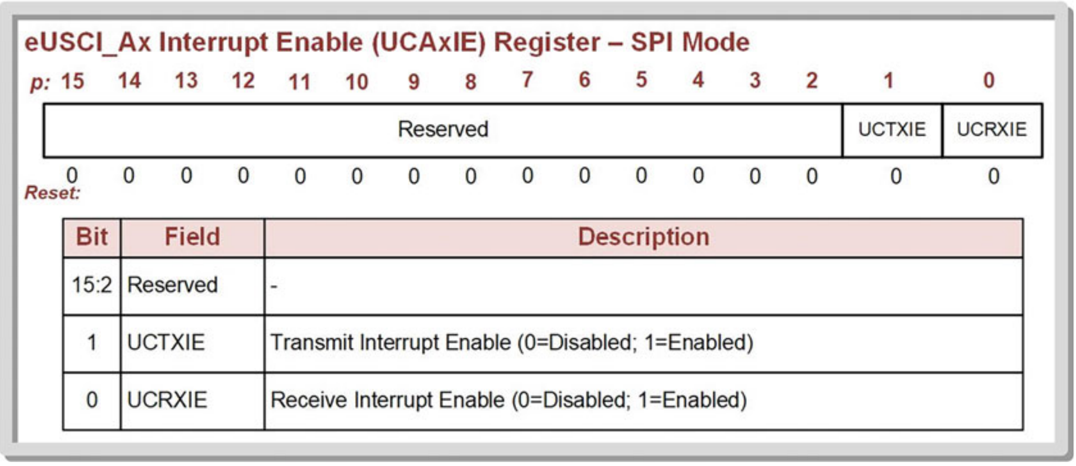
Mode: Repeated Buffer: 10
Trigger: Auto Source: Digital
Input: 100MHz DIO 0..15
Position: -25 us Base: 20 us/div
Samples: Default Rate: 20 MHz
Done 4096 samples at 7.4923 MHz | 2019-11-30 08:39:37.900
SPI: h4D
Select: ND Not defined
Clock: DIO 1
Data: DIO 2
Discovery2 SN:210321A36E26 Status: OK

Did it work? Are you able to get the logic analyzer to display the same information as the oscilloscope? Can you get the SPI decoding feature to work so that you can see the data byte in its original form (i.e., 0x4D)? A logic analyzer is a powerful debug tool when you start to send more information across the serial interfaces.

Example 14.23
Transmitting a byte from a SPI master (part 4)

Now let's look at how we can send a packet of information across a SPI bus. Once we start sending more than a single byte over SPI, we need to track whether the Tx buffer has finished sending the last byte and is ready for more data. This can be accomplished using the *transmit interrupt* (TXIFG) within the SPI system. Each time the Tx buffer is written to, it will clear the TXIFG flag. Each time the Tx buffer is empty and is ready for more information, the TXIFG flag will be set. The TXIFG flag can be used to trigger an interrupt if TXIE and global interrupts are enabled.

The TXIFG flag resides within the UCAxIFG register and is shown in Fig. 14.25. The TXIE bit resides within the UCAxIE register and is shown in Fig. 14.26. Figure 14.27 shows the IRQ vector register UCAxIV. Note that these three registers are the same as used in UART mode, but with a reduced number of fields corresponding to only two IRQs in the SPI system (UCTXIFG and UCRXIFG).

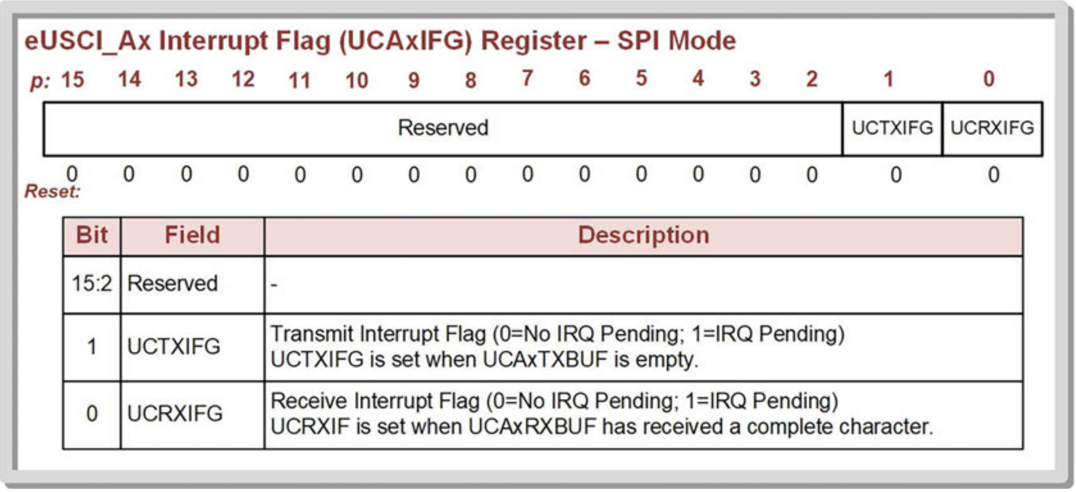


eUSCI_Ax Interrupt Enable (UCAxIE) Register – SPI Mode

p:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Reserved														UCTXIE	UCRXIE
Reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	Field	Description
15:2	Reserved	-
1	UCTXIE	Transmit Interrupt Enable (0=Disabled; 1=Enabled)
0	UCRXIE	Receive Interrupt Enable (0=Disabled; 1=Enabled)

Fig. 14.25
eUSCI_Ax interrupt enable (UCAxIE) register: SPI mode

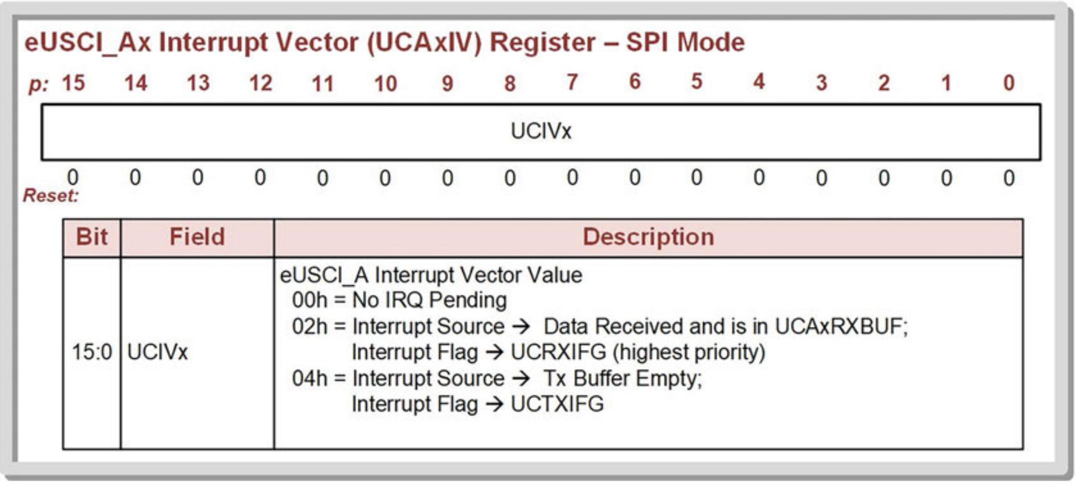


eUSCI_Ax Interrupt Flag (UCAxIFG) Register – SPI Mode

p:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Reserved														UCTXIFG	UCRXIFG
Reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	Field	Description
15:2	Reserved	-
1	UCTXIFG	Transmit Interrupt Flag (0=No IRQ Pending; 1=IRQ Pending) UCTXIFG is set when UCAxTXBUF is empty.
0	UCRXIFG	Receive Interrupt Flag (0=No IRQ Pending; 1=IRQ Pending) UCRXIF is set when UCAxRXBUF has received a complete character.

Fig. 14.26
eUSCI_Ax interrupt flag (UCAxIFG) register: SPI mode



eUSCI_Ax Interrupt Vector (UCAxIV) Register – SPI Mode

p:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	UCIVx															
Reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

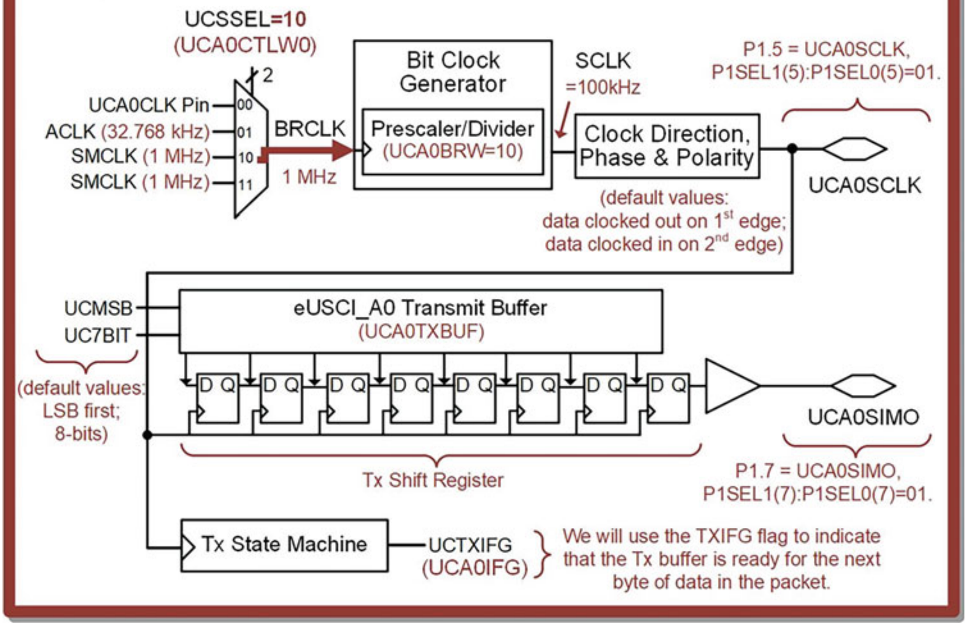
Bit	Field	Description
15:0	UCIVx	eUSCI_A Interrupt Vector Value 00h = No IRQ Pending 02h = Interrupt Source → Data Received and is in UCAxRXBUF; Interrupt Flag → UCRXIFG (highest priority) 04h = Interrupt Source → Tx Buffer Empty; Interrupt Flag → UCTXIFG

Fig. 14.27
eUSCI_Ax interrupt vector (UCAxIV) register: SPI mode

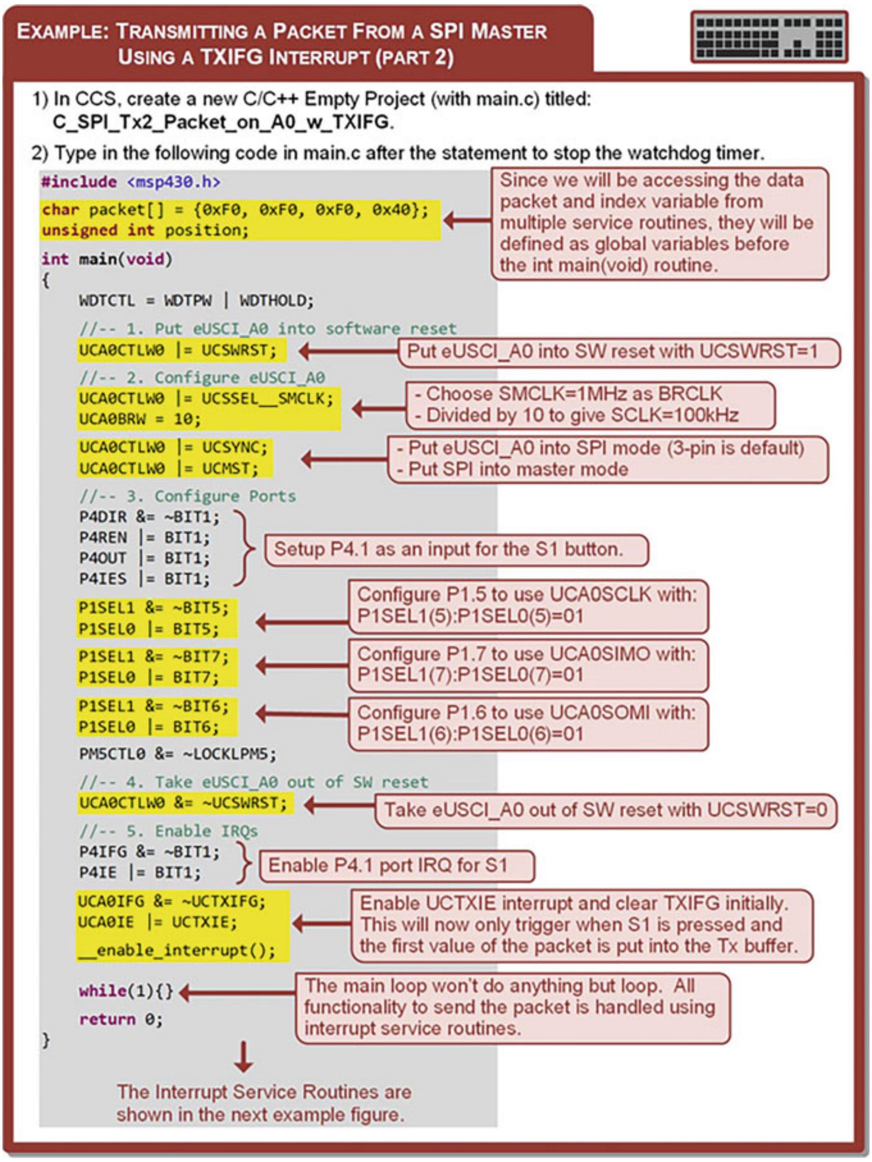
Let’s do an example where we send a packet of four bytes out of a SPI master. We will start the transmission upon a button press on S1. We will use a port interrupt to start the transmission when S1 is pressed by placing the first byte of the packet into the Tx buffer. Once the first byte is transmitted, the TXIFG flag will be asserted indicating that the buffer is ready for the next byte of information. We will use this flag to trigger an eUSCI_A0 interrupt that will place the next byte of the packet into the Tx buffer. This process will repeat as each byte in the packet is sent over SPI. We will require some logic within the SPI service routine to check whether the entire packet has been sent and we are done. If all bytes in the packet have been sent, we will disable the SPI transmission by clearing the TXIFG flag without loading any new data into the Tx buffer. Follow Examples 14.24, 14.25, 14.26, 14.27, and 14.28 to gain experience sending packets of information out of a SPI master.

EXAMPLE: TRANSMITTING A PACKET FROM A SPI MASTER USING A TXIFG INTERRUPT (PART 1)

Let’s configure the eUSCI_A0 peripheral as a SPI master and send out a 4-byte packet each time S1 is pressed. The values we will send are **{0xF0, 0xF0, 0xF0, 0x40}**. We will use a TXIFG interrupt to track when the Tx buffer is ready for the next byte in the packet. We will observe SCLK and SIMO using both an oscilloscope and a logic analyzer. The following is the eUSCI_A0 setup for this example.


UCSSEL=10
(UCA0CTLW0)
2
UCA0CLK Pin
ACLK (32.768 kHz)
SMCLK (1 MHz)
SMCLK (1 MHz)
00
01
10
11
BRCLK
1 MHz
Bit Clock Generator
Prescaler/Divider
(UCA0BRW=10)
SCLK
=100kHz
Clock Direction, Phase & Polarity
(default values: data clocked out on 1st edge; data clocked in on 2nd edge)
P1.5 = UCA0SCLK, P1SEL1(5):P1SEL0(5)=01.
UCA0SCLK
UCMSB
UC7BIT
(default values: LSB first; 8-bits)
eUSCI_A0 Transmit Buffer
(UCA0TXBUF)
D Q
Tx Shift Register
UCA0SIMO
P1.7 = UCA0SIMO, P1SEL1(7):P1SEL0(7)=01.
Tx State Machine
UCTXIFG
(UCA0IFG)
We will use the TXIFG flag to indicate that the Tx buffer is ready for the next byte of data in the packet.

Example 14.24
Transmitting a packet from a SPI master using a TXIFG interrupt (part 1)



EXAMPLE: TRANSMITTING A PACKET FROM A SPI MASTER USING A TXIFG INTERRUPT (PART 2)

1) In CCS, create a new C/C++ Empty Project (with main.c) titled:
C_SPI_Tx2_Packet_on_A0_w_TXIFG.

2) Type in the following code in main.c after the statement to stop the watchdog timer.

```
#include <msp430.h>

char packet[] = {0xF0, 0xF0, 0xF0, 0x40};
unsigned int position;

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;

    //-- 1. Put eUSCI_A0 into software reset
    UCA0CTLW0 |= UCSWRST;

    //-- 2. Configure eUSCI_A0
    UCA0CTLW0 |= UCSSEL__SMCLK;
    UCA0BRW = 10;

    UCA0CTLW0 |= UCSYNC;
    UCA0CTLW0 |= UCMST;

    //-- 3. Configure Ports
    P4DIR &= ~BIT1;
    P4REN |= BIT1;
    P4OUT |= BIT1;
    P4IES |= BIT1;

    P1SEL1 &= ~BIT5;
    P1SEL0 |= BIT5;

    P1SEL1 &= ~BIT7;
    P1SEL0 |= BIT7;

    P1SEL1 &= ~BIT6;
    P1SEL0 |= BIT6;

    PM5CTL0 &= ~LOCKLPM5;

    //-- 4. Take eUSCI_A0 out of SW reset
    UCA0CTLW0 &= ~UCSWRST;

    //-- 5. Enable IRQs
    P4IFG &= ~BIT1;
    P4IE |= BIT1;

    UCA0IFG &= ~UCTXIFG;
    UCA0IE |= UCTXIE;

    __enable_interrupt();

    while(1){}

    return 0;
}
```

Since we will be accessing the data packet and index variable from multiple service routines, they will be defined as global variables before the int main(void) routine.

Put eUSCI_A0 into SW reset with UCSWRST=1

- Choose SMCLK=1MHz as BRCLK
- Divided by 10 to give SCLK=100kHz

- Put eUSCI_A0 into SPI mode (3-pin is default)
- Put SPI into master mode

Setup P4.1 as an input for the S1 button.

Configure P1.5 to use UCA0SCLK with: P1SEL1(5):P1SEL0(5)=01

Configure P1.7 to use UCA0SIMO with: P1SEL1(7):P1SEL0(7)=01

Configure P1.6 to use UCA0SOMI with: P1SEL1(6):P1SEL0(6)=01

Take eUSCI_A0 out of SW reset with UCSWRST=0

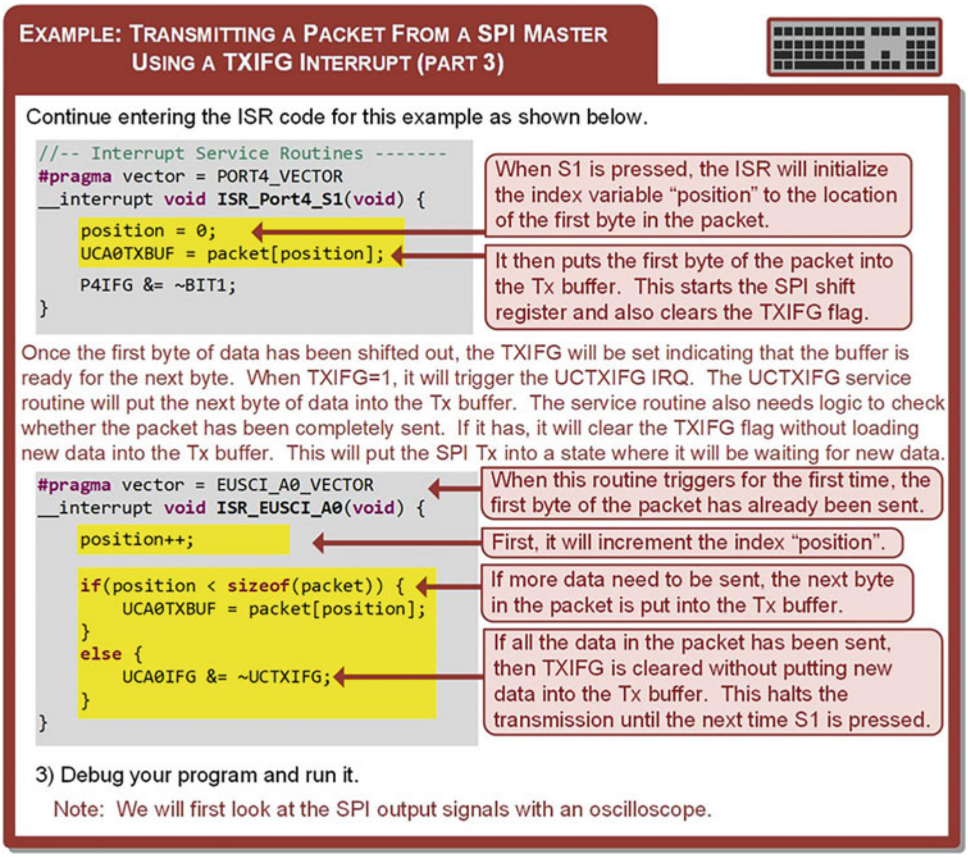
Enable P4.1 port IRQ for S1

Enable UCTXIE interrupt and clear TXIFG initially. This will now only trigger when S1 is pressed and the first value of the packet is put into the Tx buffer.

The main loop won't do anything but loop. All functionality to send the packet is handled using interrupt service routines.

The Interrupt Service Routines are shown in the next example figure.

Example 14.25
Transmitting a packet from a SPI master using a TXIFG interrupt (part 2)



Example: Transmitting a Packet From a SPI Master Using a TXIFG Interrupt (Part 3)

Continue entering the ISR code for this example as shown below.

```
//-- Interrupt Service Routines -------
#pragma vector = PORT4_VECTOR
__interrupt void ISR_Port4_S1(void) {
    position = 0;
    UCA0TXBUF = packet[position];
    P4IFG &= ~BIT1;
}
```

When S1 is pressed, the ISR will initialize the index variable "position" to the location of the first byte in the packet.

It then puts the first byte of the packet into the Tx buffer. This starts the SPI shift register and also clears the TXIFG flag.

Once the first byte of data has been shifted out, the TXIFG will be set indicating that the buffer is ready for the next byte. When TXIFG=1, it will trigger the UCTXIFG IRQ. The UCTXIFG service routine will put the next byte of data into the Tx buffer. The service routine also needs logic to check whether the packet has been completely sent. If it has, it will clear the TXIFG flag without loading new data into the Tx buffer. This will put the SPI Tx into a state where it will be waiting for new data.

```
#pragma vector = EUSCI_A0_VECTOR
__interrupt void ISR_EUSCI_A0(void) {
    position++;

    if(position < sizeof(packet)) {
        UCA0TXBUF = packet[position];
    }
    else {
        UCA0IFG &= ~UCTXIFG;
    }
}
```

When this routine triggers for the first time, the first byte of the packet has already been sent.

First, it will increment the index "position".

If more data need to be sent, the next byte in the packet is put into the Tx buffer.

If all the data in the packet has been sent, then TXIFG is cleared without putting new data into the Tx buffer. This halts the transmission until the next time S1 is pressed.

3) Debug your program and run it.

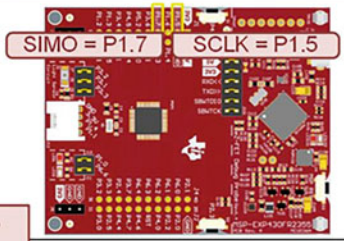
Note: We will first look at the SPI output signals with an oscilloscope.

Example 14.26
Transmitting a packet from a SPI master using a TXIFG interrupt (part 3)

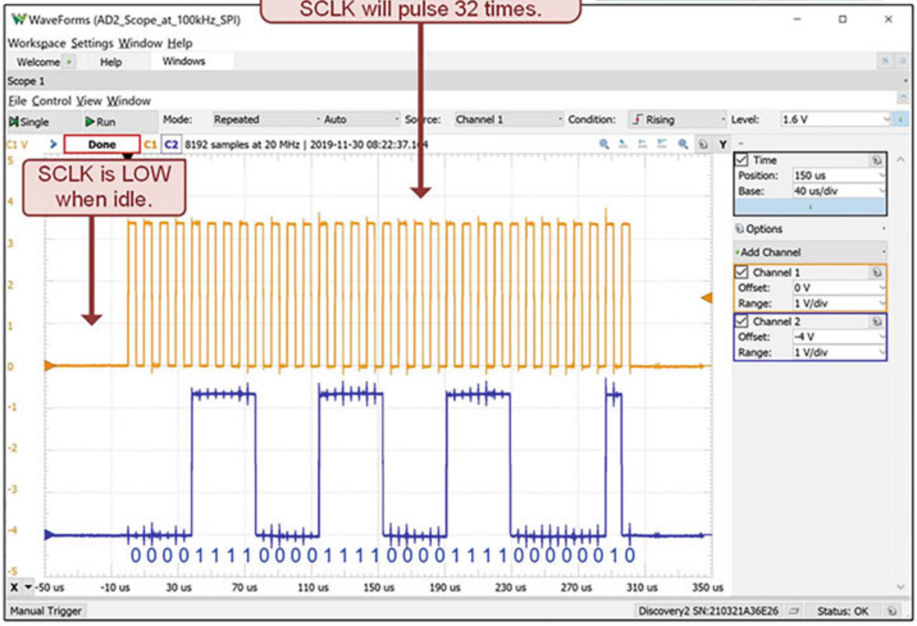
Example: Transmitting a Packet From a SPI Master Using a TXIFG Interrupt (Part 4)

4) Observe the SPI outputs on the LaunchPad^{TM} board with an oscilloscope. Probe SCLK on the P1.5 pin of the J1 header. Probe SIMO on the P1.7 pin of the J1 header.

5) Configure the oscilloscope waveform to center the SPI frames. You should see the following waveforms on your oscilloscope.



When transmitting 4x bytes, SCLK will pulse 32 times.


WaveForms (AD2_Scope_at_100kHz_SPI)

Workspace Settings Window Help

Welcome | Help | Windows

Scope 1

File Control View Window

Single | Run | Mode: Repeated | Auto | Source: Channel 1 | Condition: Rising | Level: 1.6 V

Done | C1 | C2 | 8192 samples at 20 MHz | 2019-11-30 08:22:37.14

Time — Position: 150 us — Base: 40 us/div

Options

Add Channel

☑ Channel 1 — Offset: 0 V — Range: 1 V/div

☑ Channel 2 — Offset: -4 V — Range: 1 V/div

Manual Trigger | Discovery2 SN:210321A36E26 | Status: OK

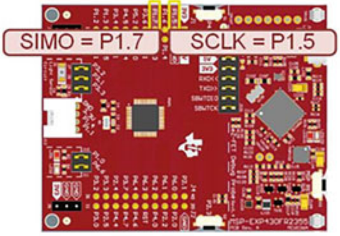
SCLK is LOW when idle.

? Did it work? Are you able to see 4x bytes of information sent each time S1 is pressed? Remember that we are sending the LSB first. After rearranging the bits into their original order, we get (0xF0, 0xF0, 0xF0, 0x40}.

Example 14.27
Transmitting a packet from a SPI master using a TXIFG interrupt (part 4)

Example: Transmitting a Packet From a SPI Master Using a TXIFG Interrupt (Part 5)

6) Now observe the SPI outputs on the LaunchPad™ board with a *logic analyzer*. The SPI tool of the logic analyzer will allow us to see the data in its original format. This allows us to more quickly determine if the SPI output is correct. Connect a logic analyzer to the SPI output pins. Probe SCLK on the P1.5 pin of the J1 header. Probe SIMO on the P1.7 pin of the J1 header.

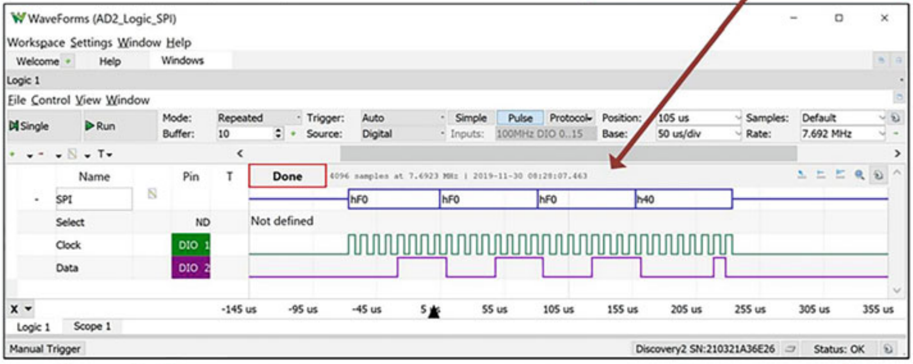


SIMO = P1.7

SCLK = P1.5

7) Configure the logic analyzer waveform to center the SPI data. Set up the trigger of the logic analyzer to watch for multiple transitions on SCLK. Set up the SPI protocol feature of the logic analyzer (i.e., LSB first, the signals for SCLK and SIMO). You should see the following waveforms on your logic analyzer.

The data packets observed by the logic analyzer match what we intended to send.



WaveForms (AD2_Logic_SPI)

Mode: Repeated

Buffer: 10

Trigger: Auto

Source: Digital

Inputs: 100MHz DIO 0..15

Position: 105 us

Base: 50 us/div

Samples: Default

Rate: 7.692 MHz

Done 4096 samples at 7.6923 MHz | 2019-11-30 08:28:07.463

Name	Pin	
SPI		hF0 hF0 hF0 h40
Select	ND	Not defined
Clock	DIO 1	
Data	DIO 2	

Manual Trigger

Discovery2 SN:210321A36E26

Status: OK

?

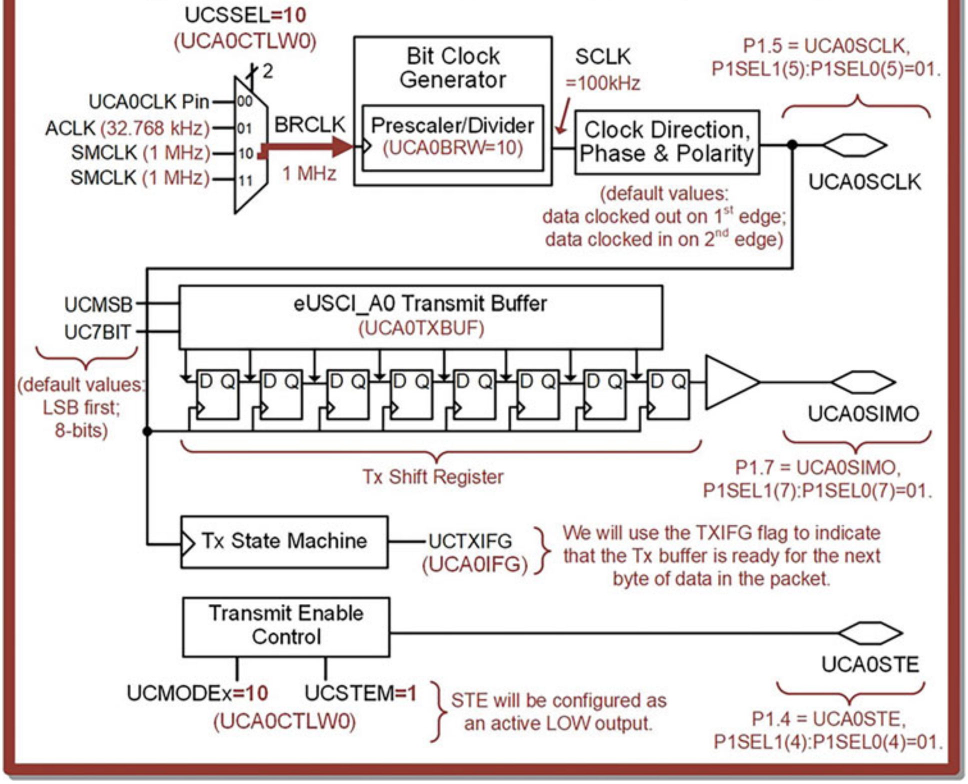
Did it work? Are you able to get the logic analyzer to display the same information as the oscilloscope? Can you get the SPI decoding feature to work so that you can see the data in its original form?

Example 14.28
Transmitting a packet from a SPI master using a TXIFG interrupt (part 5)

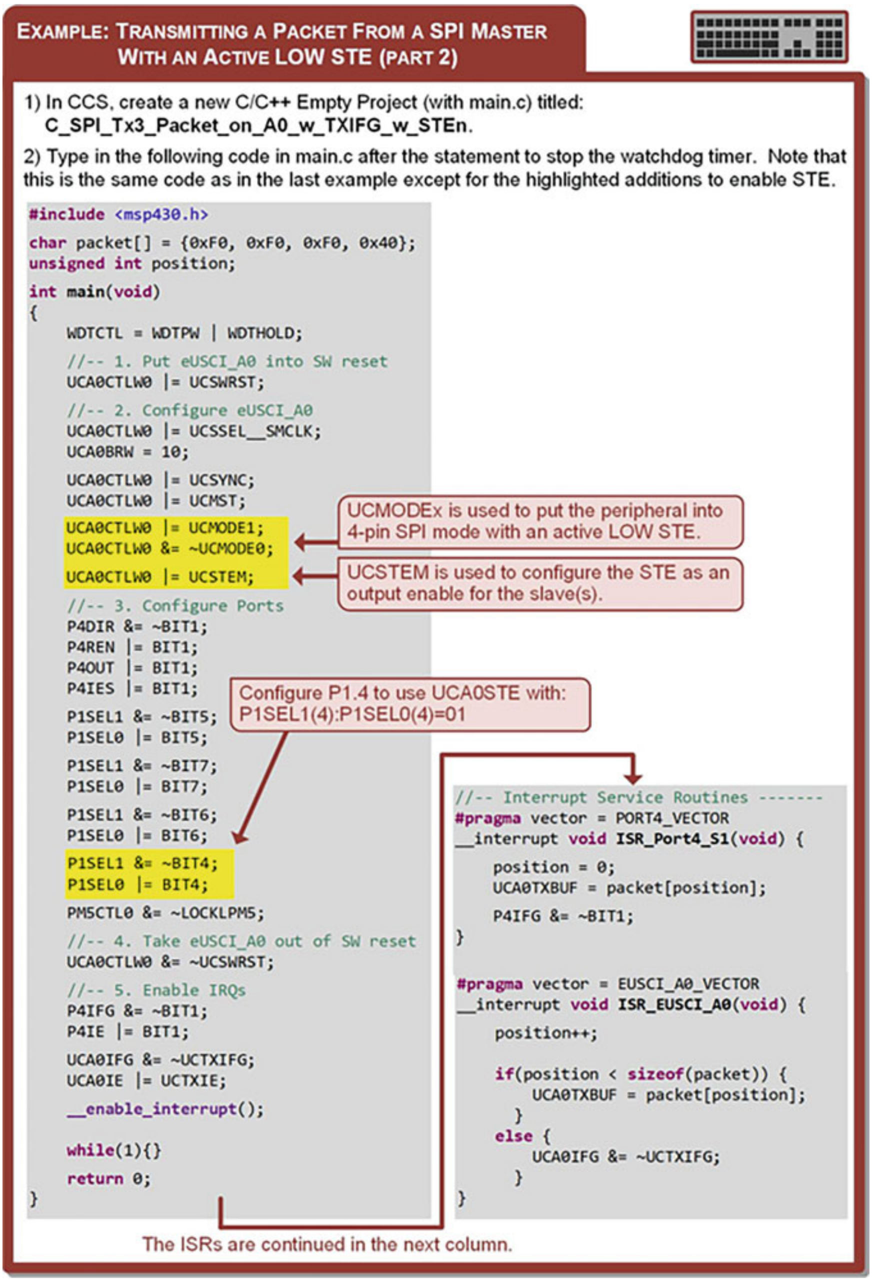
Now let's look at generating an active LOW STE signal by the master for use as an enable for the slave(s). To generate an STE, the UCMODEx bits in the UCAxCTLW0 register are used to put the peripheral into four-pin SPI mode with an active LOW STE. Then the UCSTEM bit in the UCAxCTLW0 register configures the STE to be an output. Follow the design in Examples 14.29, 14.30, and 14.31 to see how to generate an active LOW STE by the master.

Example: Transmitting a Packet From a SPI Master With an Active LOW STE (Part 1)

Let's configure the eUSCI_A0 peripheral as a SPI master and send out a 4-byte packet each time S1 is pressed. The values we will send are **{0xF0, 0xF0, 0xF0, 0x40}**. We will also generate an active LOW STE for use by the receiving slave(s). We will observe SCLK, SIMO, and STE using a logic analyzer. The following is the eUSCI_A0 setup for this example.



Example 14.29
Transmitting a packet from a SPI master with an active LOW STE (part 1)



EXAMPLE: TRANSMITTING A PACKET FROM A SPI MASTER WITH AN ACTIVE LOW STE (PART 2)

1) In CCS, create a new C/C++ Empty Project (with main.c) titled:
C_SPI_Tx3_Packet_on_A0_w_TXIFG_w_STEn.

2) Type in the following code in main.c after the statement to stop the watchdog timer. Note that this is the same code as in the last example except for the highlighted additions to enable STE.

```
#include <msp430.h>

char packet[] = {0xF0, 0xF0, 0xF0, 0x40};
unsigned int position;

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;

    //-- 1. Put eUSCI_A0 into SW reset
    UCA0CTLW0 |= UCSWRST;

    //-- 2. Configure eUSCI_A0
    UCA0CTLW0 |= UCSSEL__SMCLK;
    UCA0BRW = 10;

    UCA0CTLW0 |= UCSYNC;
    UCA0CTLW0 |= UCMST;

    UCA0CTLW0 |= UCMODE1;
    UCA0CTLW0 &= ~UCMODE0;

    UCA0CTLW0 |= UCSTEM;

    //-- 3. Configure Ports
    P4DIR &= ~BIT1;
    P4REN |= BIT1;
    P4OUT |= BIT1;
    P4IES |= BIT1;

    P1SEL1 &= ~BIT5;
    P1SEL0 |= BIT5;

    P1SEL1 &= ~BIT7;
    P1SEL0 |= BIT7;

    P1SEL1 &= ~BIT6;
    P1SEL0 |= BIT6;

    P1SEL1 &= ~BIT4;
    P1SEL0 |= BIT4;

    PM5CTL0 &= ~LOCKLPM5;

    //-- 4. Take eUSCI_A0 out of SW reset
    UCA0CTLW0 &= ~UCSWRST;

    //-- 5. Enable IRQs
    P4IFG &= ~BIT1;
    P4IE |= BIT1;

    UCA0IFG &= ~UCTXIFG;
    UCA0IE |= UCTXIE;

    __enable_interrupt();

    while(1){}

    return 0;
}
```

UCMODEx is used to put the peripheral into 4-pin SPI mode with an active LOW STE.

UCSTEM is used to configure the STE as an output enable for the slave(s).

Configure P1.4 to use UCA0STE with: P1SEL1(4):P1SEL0(4)=01

```
//-- Interrupt Service Routines -------
#pragma vector = PORT4_VECTOR
__interrupt void ISR_Port4_S1(void) {

    position = 0;
    UCA0TXBUF = packet[position];

    P4IFG &= ~BIT1;
}

#pragma vector = EUSCI_A0_VECTOR
__interrupt void ISR_EUSCI_A0(void) {

    position++;

    if(position < sizeof(packet)) {
        UCA0TXBUF = packet[position];
    }
    else {
        UCA0IFG &= ~UCTXIFG;
    }
}
```

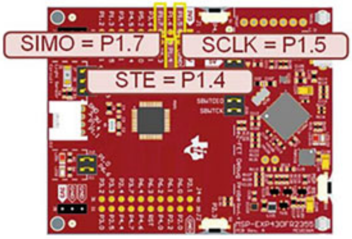
The ISRs are continued in the next column.

Example 14.30
Transmitting a packet from a SPI master with an active LOW STE (part 2)

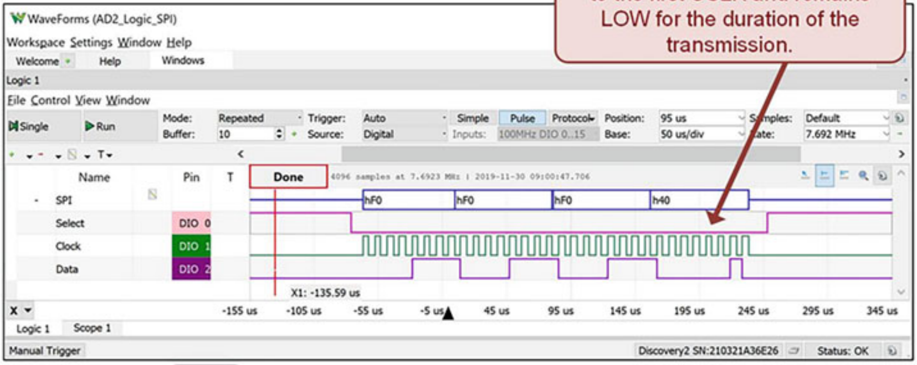
Example: Transmitting a Packet From a SPI Master With an Active LOW STE (part 3)

3) Debug your program and run it.

4) Now observe the SPI outputs on the LaunchPad™ board with a *logic analyzer*. Probe SCLK on the P1.5 pin of the J1 header. Probe SIMO on the P1.7 pin of the J1 header. Probe STE on the P1.4 pin of the J1 header.



5) Configure the logic analyzer waveform to center the SPI data. Set up the trigger of the logic analyzer to watch for multiple transitions on SCLK. Set up the SPI protocol feature of the logic analyzer (i.e., LSB first, the signals for SCLK, SIMO, and STE). You should see the following waveforms on your logic analyzer.



Mode: Repeated
Buffer: 10
Trigger: Auto
Source: Digital
Inputs: 100MHz DIO 0..15
Position: 95 us
Base: 50 us/div
Samples: Default
Rate: 7.692 MHz

Done
4096 samples at 7.6923 MHz | 2019-11-30 09:00:47.706
X1: -135.59 us
Manual Trigger
Discovery2 SN:210321A36E26
Status: OK

Notice how STE goes LOW prior to the first SCLK and remains LOW for the duration of the transmission.

? Did it work? Are you able to see the active LOW STE during the data transmission?

Example 14.31
Transmitting a packet from a SPI master with an active LOW STE (part 3)

14.2.2.2 *Receiving Data as the SPI Master*

When configured as the master and receiving data on SOMI, the master still produces the SCLK pulses to shift the data out of the slave and into the master. The data is received through an Rx shift register. When all bits have been shifted in, the byte of data is moved into an RX buffer (UCAxRXBUF). The Rx system tracks the incoming data and sets the RXIFG flag in the UCAxIFG register when new data has arrived in the buffer. Interrupts can be generated by the RXIFG by enabling the RXIE bit in the UCAxIE register. Figure 14.28 shows the concept for a SPI receiver.

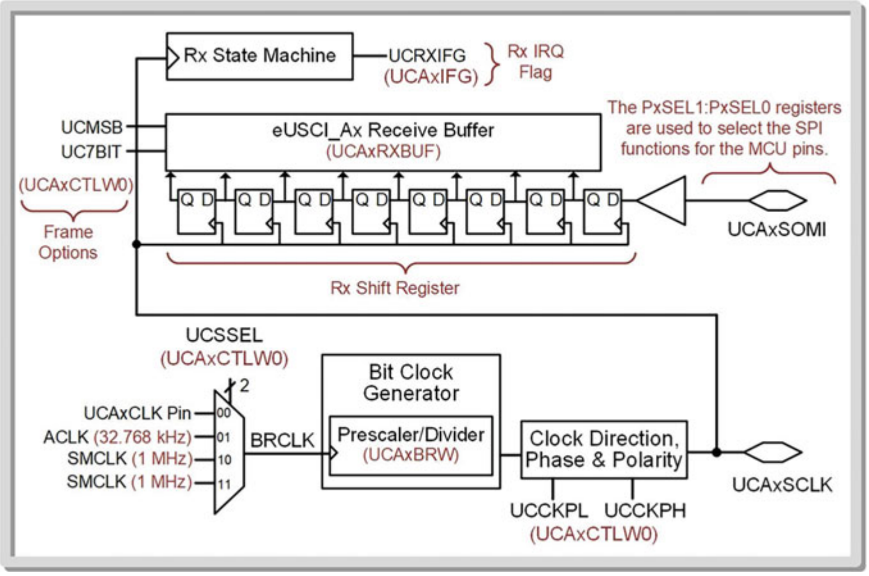


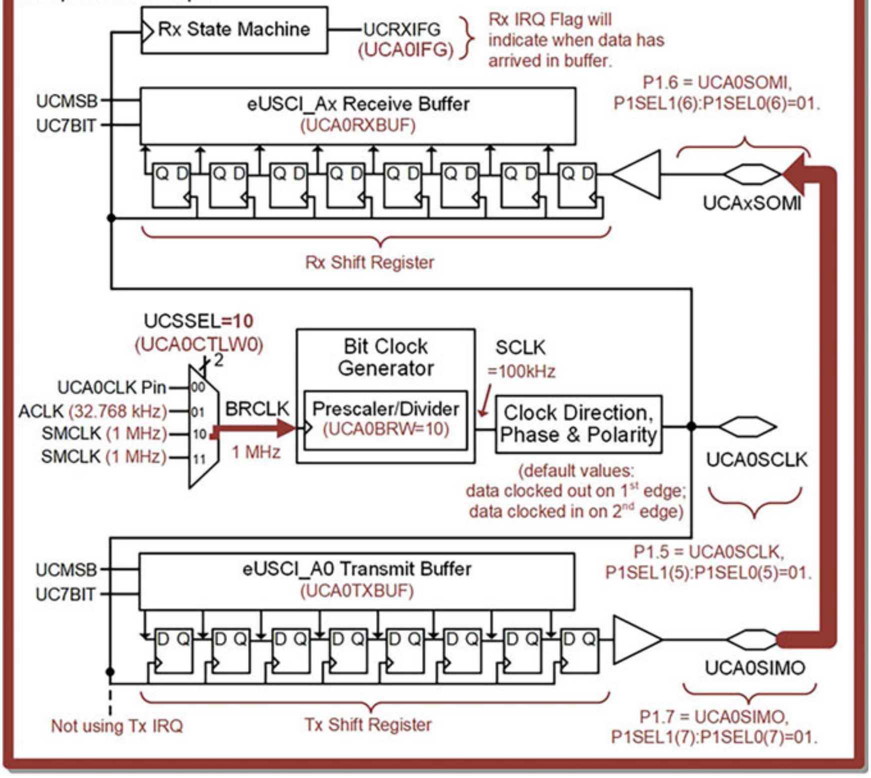
Fig. 14.28
SPI Rx general block diagram of the eUSCI_Ax peripheral on the MSP430FR2355

Since the master generates SCLK, it must send the necessary clock pulses to the slave in order for the slave to shift out its data back into the master Rx shift register. This is accomplished by writing a dummy byte of any value to the Tx register on the master in order to force the system to generate eight SCLK pulses.

Let's look at an example of receiving data as a master. Since we want to do this example using only the MSP430FR2355 LaunchPad™ board, we will simply connect the SIMO pin of eUSCI_A0 to the SOMI pin of eUSCI_A0. This will allow us to send out a byte of data on SIMO and watch it arrive in the Rx buffer eight clock cycles later. While this example is somewhat pointless, it does illustrate how a master requests data from a slave by writing to the Tx buffer and then waiting for an RXIFG interrupt to indicate that data is available in the Rx buffer. Follow the design in Examples 14.32, 14.33, 14.34, and 14.35 to see how to receive data as a SPI master. In this design, the push-button switches S1 and S2 are used to send different bytes of data (0 × 10 and 0 × 66) from the Tx to the Rx. When the Rx receives 0 × 10, it will toggle LED1. When the Rx receives 0 × 66, it will toggle LED2.

Example: Receiving a Byte as a SPI Master (Part 1)

Let's configure the eUSCI_A0 peripheral as a SPI master and see how to receive data. In order to accomplish this example using only the LaunchPad^{TM} board, we will physically connect the SIMO pin to the SOMI pin of the J1 header. We will use S1 and S2 button presses to transmit different bytes of data (**0x10** and **0x66**). When the data is received by the Rx buffer, it will assert the RXIFG flag and trigger an interrupt. In the eUSCI_A0 interrupt service routine, the data in the Rx buffer will be stored into a variable. If the data received was 0x10, the ISR will toggle LED1. If the data received was 0x66, the ISR will toggle LED2. The following is the eUSCI_A0 setup for this example.

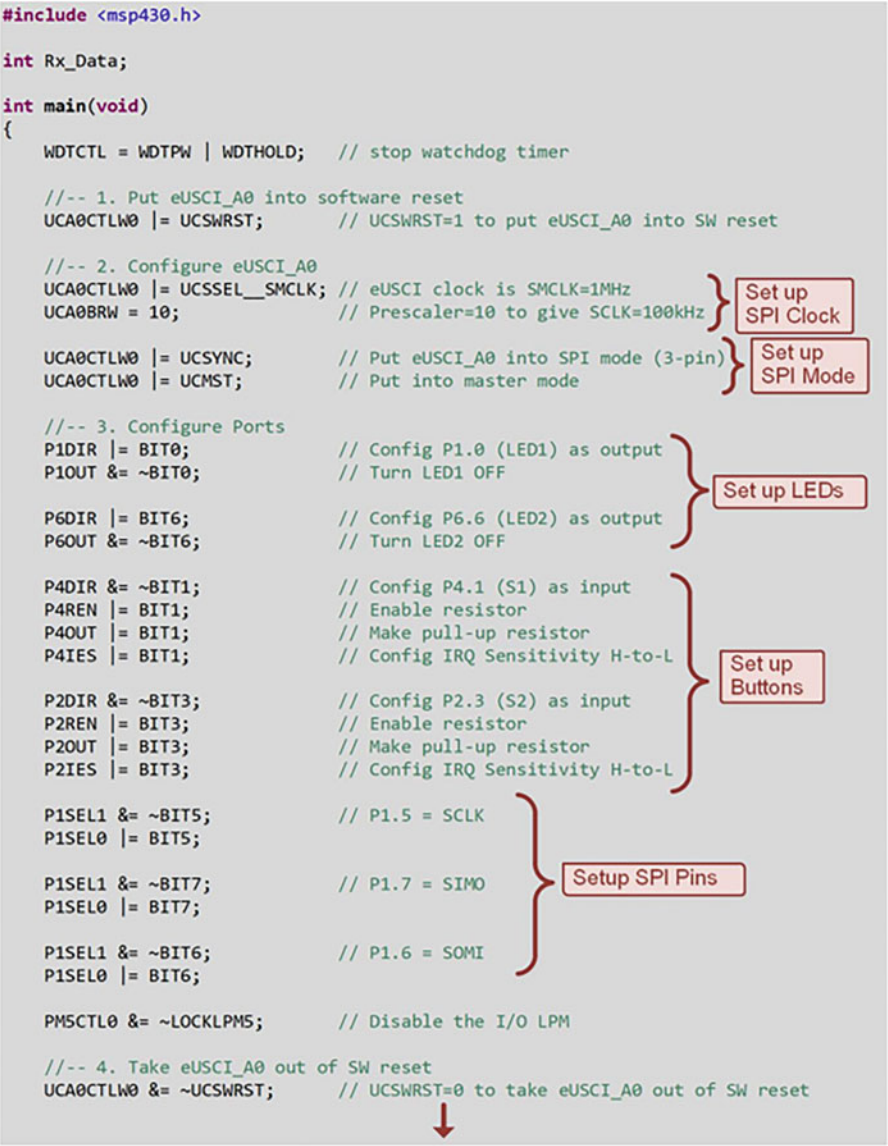


Example 14.32
Receiving a byte as a SPI master (part 1)

Example: Receiving a Byte as a SPI Master (Part 2)

1) In CCS, create a new C/C++ Empty Project (with main.c) titled:
C_SPI_Rx1_Hex_on_A0.

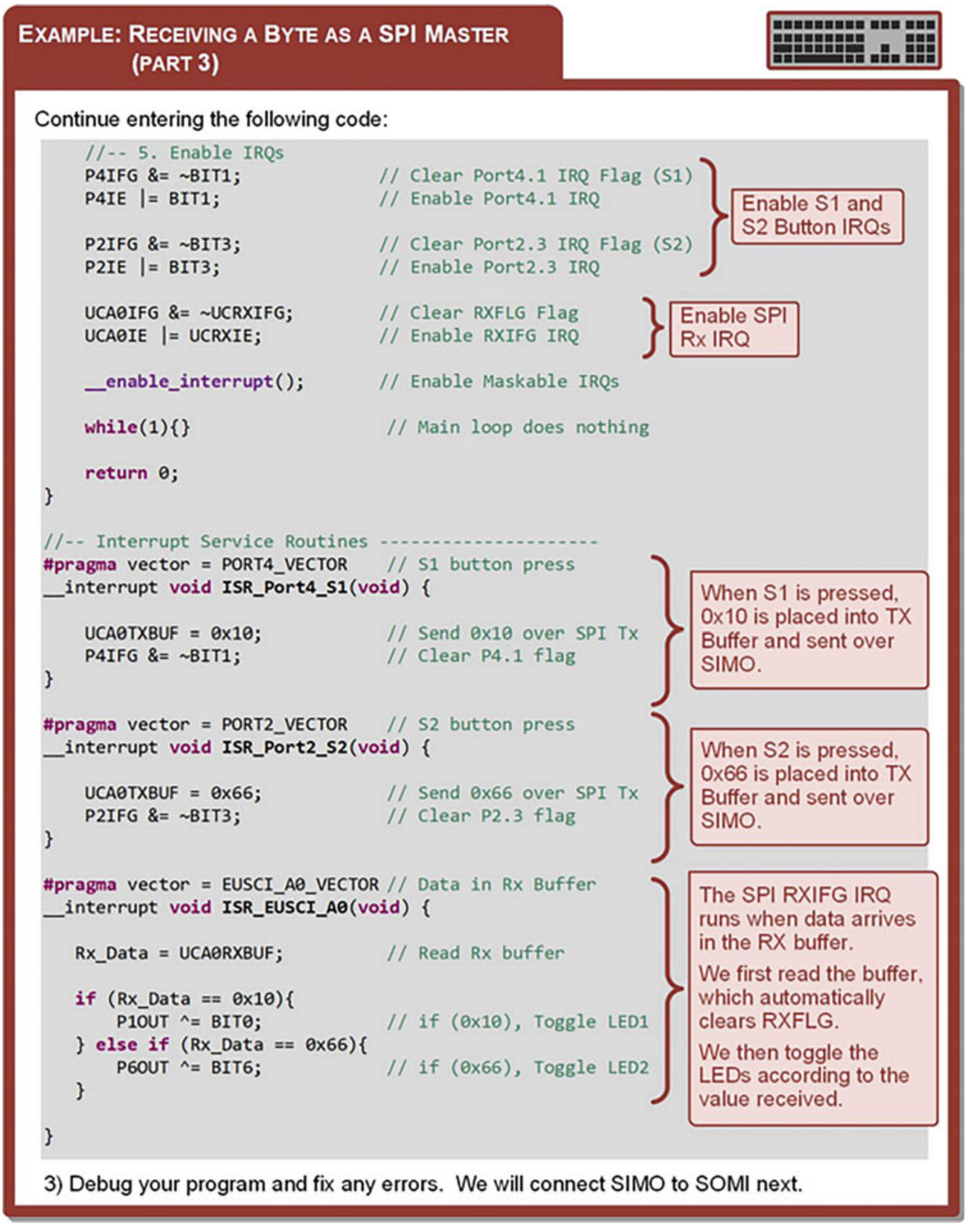
2) Type in the following code in main.c after the statement to stop the watchdog timer.



```
#include <msp430.h>

int Rx_Data;

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

    //-- 1. Put eUSCI_A0 into software reset
    UCA0CTLW0 |= UCSWRST;       // UCSWRST=1 to put eUSCI_A0 into SW reset

    //-- 2. Configure eUSCI_A0
    UCA0CTLW0 |= UCSSEL__SMCLK; // eUSCI clock is SMCLK=1MHz
    UCA0BRW = 10;               // Prescaler=10 to give SCLK=100kHz

    UCA0CTLW0 |= UCSYNC;        // Put eUSCI_A0 into SPI mode (3-pin)
    UCA0CTLW0 |= UCMST;         // Put into master mode

    //-- 3. Configure Ports
    P1DIR |= BIT0;              // Config P1.0 (LED1) as output
    P1OUT &= ~BIT0;             // Turn LED1 OFF

    P6DIR |= BIT6;              // Config P6.6 (LED2) as output
    P6OUT &= ~BIT6;             // Turn LED2 OFF

    P4DIR &= ~BIT1;             // Config P4.1 (S1) as input
    P4REN |= BIT1;              // Enable resistor
    P4OUT |= BIT1;              // Make pull-up resistor
    P4IES |= BIT1;              // Config IRQ Sensitivity H-to-L

    P2DIR &= ~BIT3;             // Config P2.3 (S2) as input
    P2REN |= BIT3;              // Enable resistor
    P2OUT |= BIT3;              // Make pull-up resistor
    P2IES |= BIT3;              // Config IRQ Sensitivity H-to-L

    P1SEL1 &= ~BIT5;            // P1.5 = SCLK
    P1SEL0 |= BIT5;

    P1SEL1 &= ~BIT7;            // P1.7 = SIMO
    P1SEL0 |= BIT7;

    P1SEL1 &= ~BIT6;            // P1.6 = SOMI
    P1SEL0 |= BIT6;

    PM5CTL0 &= ~LOCKLPM5;       // Disable the I/O LPM

    //-- 4. Take eUSCI_A0 out of SW reset
    UCA0CTLW0 &= ~UCSWRST;      // UCSWRST=0 to take eUSCI_A0 out of SW reset
```

Code continued in next example figure.

Example 14.33
Receiving a byte as a SPI master (part 2)



Example: Receiving a Byte as a SPI Master (Part 3)

Continue entering the following code:

```
    //-- 5. Enable IRQs
    P4IFG &= ~BIT1;             // Clear Port4.1 IRQ Flag (S1)
    P4IE  |= BIT1;              // Enable Port4.1 IRQ

    P2IFG &= ~BIT3;             // Clear Port2.3 IRQ Flag (S2)
    P2IE  |= BIT3;              // Enable Port2.3 IRQ

    UCA0IFG &= ~UCRXIFG;        // Clear RXFLG Flag
    UCA0IE  |= UCRXIE;          // Enable RXIFG IRQ

    __enable_interrupt();       // Enable Maskable IRQs

    while(1){}                   // Main loop does nothing

    return 0;
}

//-- Interrupt Service Routines --------------------
#pragma vector = PORT4_VECTOR    // S1 button press
__interrupt void ISR_Port4_S1(void) {

    UCA0TXBUF = 0x10;            // Send 0x10 over SPI Tx
    P4IFG &= ~BIT1;              // Clear P4.1 flag
}

#pragma vector = PORT2_VECTOR    // S2 button press
__interrupt void ISR_Port2_S2(void) {

    UCA0TXBUF = 0x66;            // Send 0x66 over SPI Tx
    P2IFG &= ~BIT3;              // Clear P2.3 flag
}

#pragma vector = EUSCI_A0_VECTOR // Data in Rx Buffer
__interrupt void ISR_EUSCI_A0(void) {

   Rx_Data = UCA0RXBUF;          // Read Rx buffer

   if (Rx_Data == 0x10){
       P1OUT ^= BIT0;            // if (0x10), Toggle LED1
   } else if (Rx_Data == 0x66){
       P6OUT ^= BIT6;            // if (0x66), Toggle LED2
   }

}
```

Enable S1 and S2 Button IRQs

Enable SPI Rx IRQ

When S1 is pressed, 0x10 is placed into TX Buffer and sent over SIMO.

When S2 is pressed, 0x66 is placed into TX Buffer and sent over SIMO.

The SPI RXIFG IRQ runs when data arrives in the RX buffer.

We first read the buffer, which automatically clears RXFLG.

We then toggle the LEDs according to the value received.

3) Debug your program and fix any errors. We will connect SIMO to SOMI next.

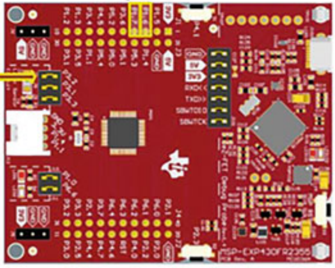
Example 14.34
Receiving a byte as a SPI master (part 3)

Example: Receiving a Byte as a SPI Master (Part 4)

4) Connect the eUSCI_A0 SIMO pin to the SOMI pin on the LaunchPad^{TM} board.

5) Run and test your program. You should be able to press S1 and see LED1 toggle. You should be able to press S2 and see LED2 toggle.

If it doesn't work, check your eUSCI_A0 settings in the Register Viewer in CCS.



eUSCO_A0 SIMO is on P1.7

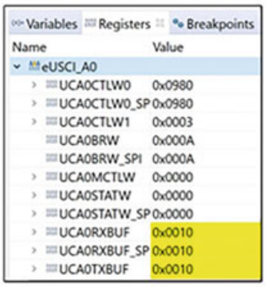
eUSCO_A0 SOMI is on P1.6

You can borrow one of these jumpers to connect P1.6 to P1.7.

One of the drawbacks of only using the LEDs to test your SPI program is that you can't tell for sure if the SPI bus is working (i.e., you may have accidently downloaded one of your prior digital I/O programs that does the same thing). Let's verify that the Rx buffer is actually getting the data we are sending by looking at the Rx Buffer in the Register Viewer of CCS.

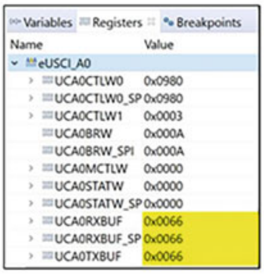
6) View the Rx Buffer in the debugger.

- Press S1 and then pause your program in CCS. At this point, you can view the current status of the eUSCI_A0 registers. Open the Register Viewer and browse to the eUSCI_A0 register section. Expand the registers and locate UCA0RXBUF. You should see the value: **0x10**.



Name	Value
eUSCI_A0	
UCA0CTLW0	0x0980
UCA0CTLW0_SP	0x0980
UCA0CTLW1	0x0003
UCA0BRW	0x000A
UCA0BRW_SPI	0x000A
UCA0MCTLW	0x0000
UCA0STATW	0x0000
UCA0STATW_SP	0x0000
UCA0RXBUF	0x0010
UCA0RXBUF_SP	0x0010
UCA0TXBUF	0x0010

- Run your program again and press S2. Pause your program in CCS and view the UCA0RXBUF register. You should see the value: **0x66**.



Name	Value
eUSCI_A0	
UCA0CTLW0	0x0980
UCA0CTLW0_SP	0x0980
UCA0CTLW1	0x0003
UCA0BRW	0x000A
UCA0BRW_SPI	0x000A
UCA0MCTLW	0x0000
UCA0STATW	0x0000
UCA0STATW_SP	0x0000
UCA0RXBUF	0x0066
UCA0RXBUF_SP	0x0066
UCA0TXBUF	0x0066

? Did it work? Can you see the values you sent over SPI in the RX Buffer? This verifies that your program is truly doing what you intended.

Example 14.35
Receiving a byte as a SPI master (part 4)

14.2.3 SPI Slave Operation on the MSP430FR2355

14.2.3.1 Transmitting Data as a SPI Slave

When acting as a SPI slave, the device must be configured to match the system settings of the master. These include the three-pin vs. four-pin mode, STE polarity, LSB vs. MSB first, 8-bit vs. 7-bit, clock polarity, and clock phase. The device must also be put into slave mode using the UCMST setting. The clock source and bit rate does not need to be configured since the SCLK will be sent by the master. When the MCU sees UCMST = 0, it knows to use the SCLK being received instead of from its own clock generator.

Transmitting as a SPI slave consists of placing data into the Tx buffer and then waiting for the master to send eight SCLK pulses to shift the data out. Once data is placed into the Tx buffer, the slave can monitor the TXFLG to see when the data has been completely shifted out. Since the slave does not control SCLK, it will not know when the data has been shifted out. As such, using a TXFLG interrupt to indicate the data has been transmitted is recommended.

14.2.3.2 Receiving Data as a SPI Slave

When configured as a SPI slave, receiving data consists of waiting passively until data has been shifted into the Rx buffer. Once a full frame has arrived, the RXFLG flag is asserted. As the slave, this is the only method that exists to indicate that data has arrived. As such, using an RXFLG interrupt to indicate the receipt of data is the most efficient way to configure the SPI receiver.

Concept Check

CC14.2 Is there such thing as “half-duplex” in a SPI link?

A) Yes. You just need to connect the SIMO and SOMI pins together and take turns driving the bus.

B) No. SPI uses dedicated signals to send data in specific directions. SIMO always transfers data from the master and SOMI always transfers data from the slave.

14.3 Inter-integrated Circuit (I2C) Bus

14.3.1 The I2C Protocol

The inter-integrated circuit (I^2C=I2C, pronounced “I-squared-C”) standard is a serial interface implemented with a two-wire link that can support multiple masters and multiple slaves. An I2C bus contains a clock line (SCL = *serial clock*) and a data line (SDA = *serial data*). An I2C link is always half-duplex, meaning that all devices share the data line with only one device transmitting at any given time.

The I2C bus supports multiple drivers on the same signal line by using the concept of an *open-drain* output stage. In this type of driver, the output consists of an n-type MOSFET (metal oxide semiconductor, field effect transistor) or NMOS transistor. The NMOS transistor works as a voltage-controlled switch that can be used to pull the signal line to a logic level. The NMOS switch will turn ON and close when its gate (the control terminal) voltage is at V_{CC}. The NMOS switch will turn OFF when its gate voltage is at GND. When the NMOS is ON, a conduction path is formed between its other two terminals (the source and drain), which effectively closes the switch. When the NMOS is OFF, there is no conduction path between the source and drain, which effectively opens the switch. By connecting the drain terminal of the NMOS to the signal line and the source terminal to GND, an open-drain output stage can pull the signal line to a logic LOW by shorting it to GND; however, an open-drain output stage doesn’t have the ability to pull the signal line to a logic HIGH. To accomplish driving the line HIGH, a pull-up resistor is used. By placing a pull-up resistor to V_{CC} on the signal line, it will be pulled to a logic HIGH when the NMOS is OFF. Figure 14.29 shows the concept of an open-drain output.

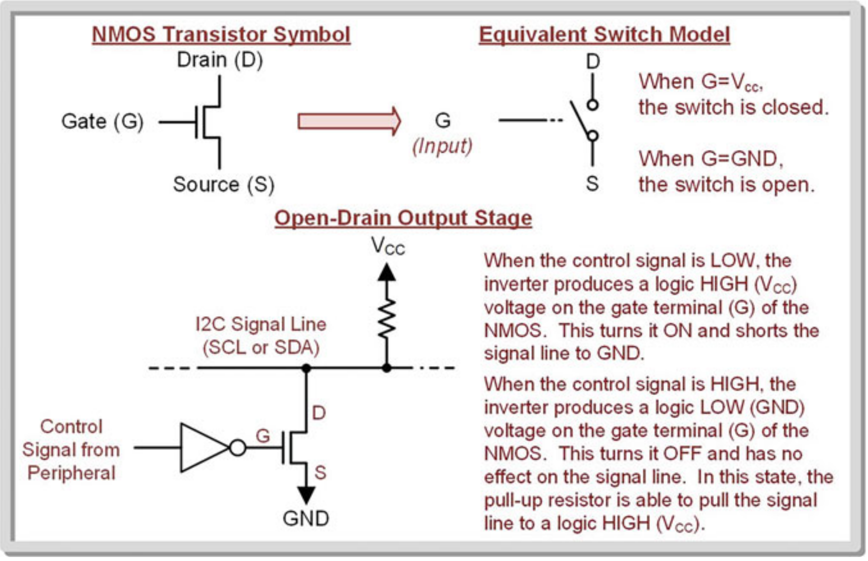

NMOS Transistor Symbol
Drain (D)
Gate (G)
Source (S)
Equivalent Switch Model
D
G
(Input)
S
When G=V$_{cc}$, the switch is closed.
When G=GND, the switch is open.
Open-Drain Output Stage
V$_{CC}$
I2C Signal Line (SCL or SDA)
D
G
S
GND
Control Signal from Peripheral
When the control signal is LOW, the inverter produces a logic HIGH (V$_{CC}$) voltage on the gate terminal (G) of the NMOS. This turns it ON and shorts the signal line to GND.
When the control signal is HIGH, the inverter produces a logic LOW (GND) voltage on the gate terminal (G) of the NMOS. This turns it OFF and has no effect on the signal line. In this state, the pull-up resistor is able to pull the signal line to a logic HIGH (V$_{CC}$).

Fig. 14.29
Open-drain transmitter used on I2C buses

The open-drain output stage inherently creates a negative logic scheme because the NMOS control signal needs to be a HIGH in order to pull the signal line to a LOW. In order to reverse the logic scheme from the point of view of the MCU, an inverter is placed before the NMOS transistor. This allows the peripheral to use positive logic (i.e., driving the output stage with a HIGH produces a signal line HIGH and driving a LOW produces a signal line LOW).

Figure 14.30 shows the architecture of an I2C link highlighting the required pull-up resistors needed on both SCK and SDA.

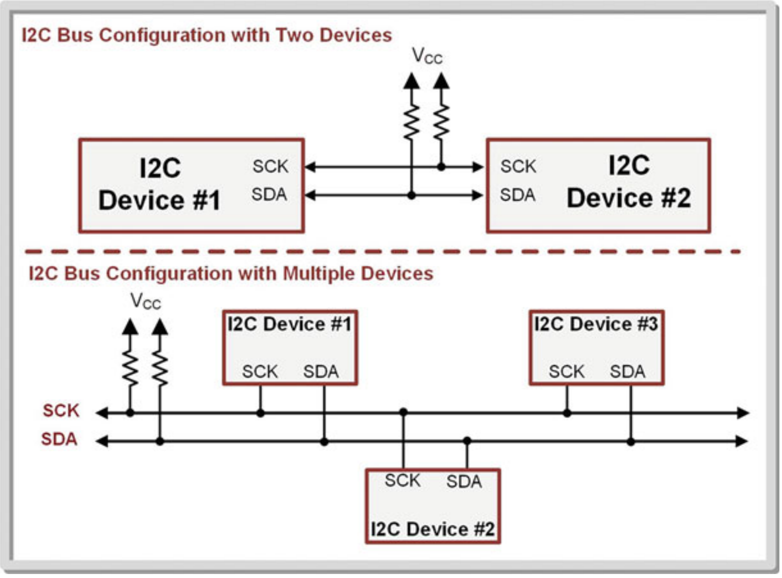


Fig. 14.30
I2C bus configurations

I2C uses the concept of masters and slaves when communicating over the bus. The master is the device that initiates communication and controls the clock. Multiple masters are also supported on an I2C bus. Each slave on the bus has a unique and predetermined address called the *slave address*. This address is used by the master to indicate which slave it wants to communicate with. Some I2C devices have a hard-coded slave address that can't be changed. Other devices may provide a portion of a hard-coded slave address and allow the user to insert pull-up or pull-down resistors on pins to set the remaining bits of the slave address.

When the bus is *idle*, both SDA and SCL are held high by the pull-up resistors, and no I2C device is attempting to communicate. When devices are driving the bus, it is said to be *busy*.

I2C information is transferred in *messages*. A typical I2C message begins with a START (S) condition and ends with a STOP (P) conditions. A master initiates a new message by generating a *START* (S) condition by pulling SDA LOW while SCL is still HIGH. This tells every device on the bus that a master is about to start communicating and they should get ready. As soon as the START condition is generated, the SCL will be pulled LOW and start pulsing to provide the clock for the message. The master is responsible for pulsing the clock. The master ends a message by generating a STOP condition. A STOP condition occurs when there is a LOW-to-HIGH transition on SDA while SCL is HIGH. Once SDA goes HIGH, SCL also remains HIGH indicating that the bus is idle again.

Within a message, the data is divided into frames and control/status signals. Each clock pulse within the I2C message is numbered by *periods*. The first clock pulse after a message is initiated is denoted "period 1." The second clock pulse is denoted "period 2," and so on. Both the master and slaves count the number of periods that have occurred since the message started in order to know when certain frames and signals should be present on the bus. This is critical so that each device knows when it is allowed to communicate within the message. This avoids multiple devices from pulling the SDA line down the wrong time and causing an overcurrent situation. When discussing the details of an I2C message, many documents will refer to the period number to identify what should be happening on the bus at any given clock pulse.

After the master generates the START condition, it first sends the slave address that it wishes to communicate with. I2C slave addresses can either be 7 bit (default) or 10 bit. The slave address is followed by the read/write signal indicating which type of transaction is being requested in the message. The START condition, slave address, and read/write signal constitute periods 1→8.

Period 9 of the message is reserved for the slave acknowledge (ACK) or no-acknowledge (NACK) signal. After the slave address and read/write signal are sent by the master, each slave on the bus checks whether it is being addressed. If a slave exists with the specified slave address, it will send an ACK signal back to the master by pulling SDA LOW. If the master sees the ACK signal, it knows a slave exists with the specified address and proceeds with the message. If no device exists on the bus with the specified slave address, no device will pull down SDA. This will result in period 9 remaining HIGH and will be interpreted as a NACK. A NACK in period 9 tells the master that no slave exists with the specified address. The master then generates a STOP condition and ends the message.

After a successful ACK from the slave, data is then sent 8 bits at a time starting with the MSB. After each byte is sent, the receiving device sends an ACK signal indicating that it successfully received the data. When the master is writing to a slave, the master sends the 8 bits of data and the slave produces the ACK/NACK signal. When the master is reading from a slave, the slave sends the 8-bits of data and the master produces the ACK/NACK signal. After the data has been sent and acknowledged, the master can end the message by generating the STOP condition. Figure 14.31 shows the message breakdown of a typical I2C bus when transferring one byte of data between a master and slave.

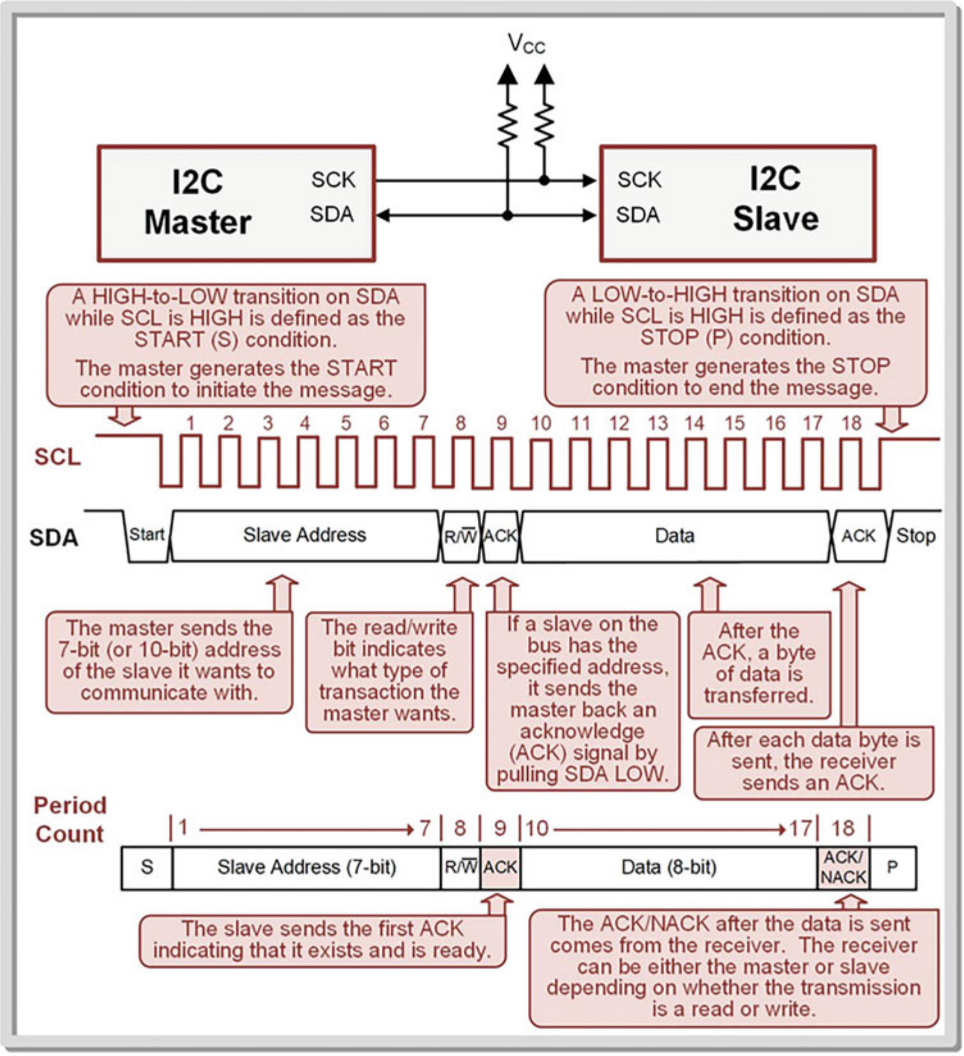
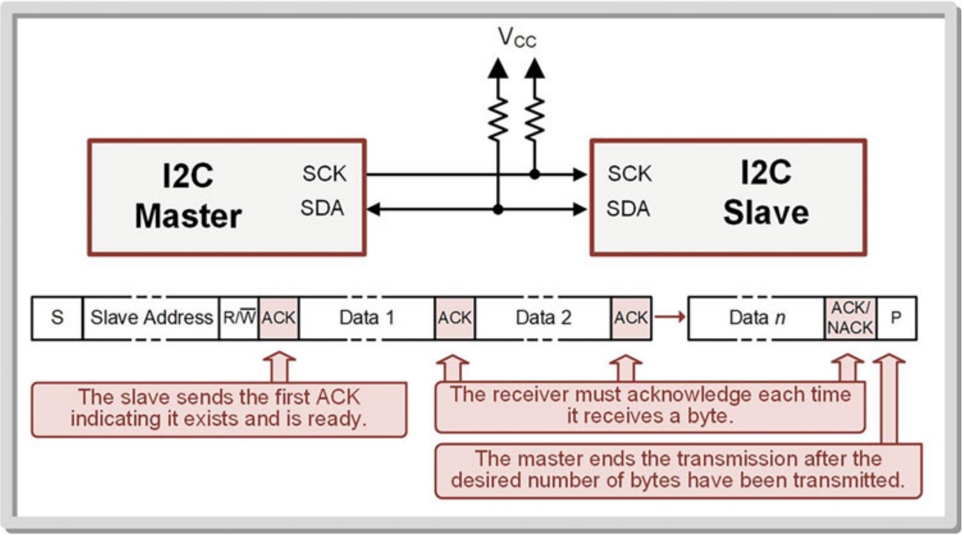

Vcc
I2C Master
SCK
SDA
SCK
SDA
I2C Slave
A HIGH-to-LOW transition on SDA while SCL is HIGH is defined as the START (S) condition.
The master generates the START condition to initiate the message.
A LOW-to-HIGH transition on SDA while SCL is HIGH is defined as the STOP (P) condition.
The master generates the STOP condition to end the message.
SCL
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
SDA
Start
Slave Address
R/W̄
ACK
Data
ACK
Stop
The master sends the 7-bit (or 10-bit) address of the slave it wants to communicate with.
The read/write bit indicates what type of transaction the master wants.
If a slave on the bus has the specified address, it sends the master back an acknowledge (ACK) signal by pulling SDA LOW.
After the ACK, a byte of data is transferred.
After each data byte is sent, the receiver sends an ACK.
Period Count
1
7
8
9
10
17
18
S
Slave Address (7-bit)
R/W̄
ACK
Data (8-bit)
ACK/ NACK
P
The slave sends the first ACK indicating that it exists and is ready.
The ACK/NACK after the data is sent comes from the receiver. The receiver can be either the master or slave depending on whether the transmission is a read or write.

Fig. 14.31
I2C bus protocol (transferring 1 byte of data)

Multiple bytes of data can also be sent within one message. Figure 14.32 shows the I2C protocol when transferring multiple bytes of data in a message.



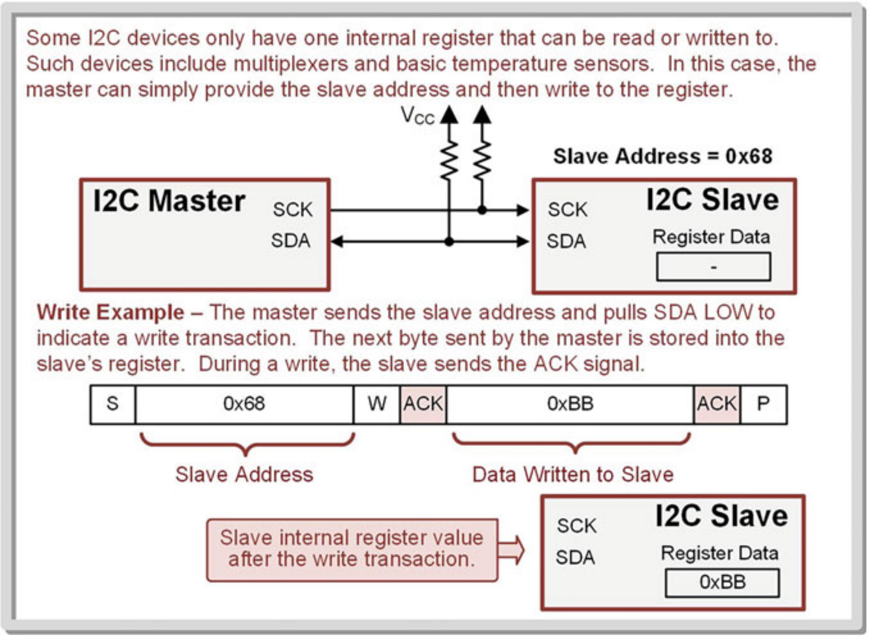
The slave sends the first ACK indicating it exists and is ready.

The receiver must acknowledge each time it receives a byte.

The master ends the transmission after the desired number of bytes have been transmitted.

Fig. 14.32
I2C bus protocol (transferring multiple bytes of data)

I2C devices contain individual registers that hold their information. In the simplest case, an I2C device contains only a single register. In this situation, read and write transactions are simply initiated by the master to access information in the slave. Figure 14.33 shows the I2C bus protocol for writing to a device with only a single register.

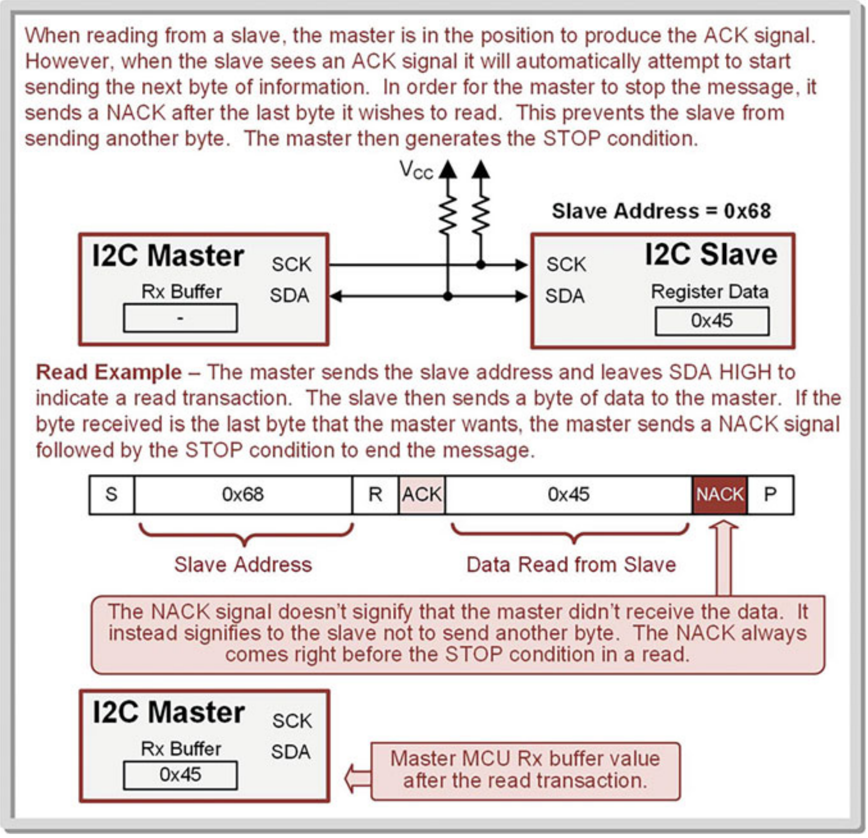

Some I2C devices only have one internal register that can be read or written to. Such devices include multiplexers and basic temperature sensors. In this case, the master can simply provide the slave address and then write to the register.

Write Example – The master sends the slave address and pulls SDA LOW to indicate a write transaction. The next byte sent by the master is stored into the slave's register. During a write, the slave sends the ACK signal.

Slave internal register value after the write transaction.

Fig. 14.33
I2C bus protocol when writing to a device with a single register

When reading from an I2C device with only a single register, the master initiates a message, sends the slave address, and leaves the read/write period HIGH to indicate that a read is being conducted. When data is transferred from the slave to the master, the master is in the position to produce the ACK signal; however, when the slave sees an ACK, it will automatically send another byte of data. In order for the master to tell the slave to stop sending data, it produces a NACK signal instead of an ACK after the last byte of data it wants to read. Figure 14.34 shows the I2C bus protocol when reading from a slave with only a single register.



When reading from a slave, the master is in the position to produce the ACK signal. However, when the slave sees an ACK signal it will automatically attempt to start sending the next byte of information. In order for the master to stop the message, it sends a NACK after the last byte it wishes to read. This prevents the slave from sending another byte. The master then generates the STOP condition.

Read Example – The master sends the slave address and leaves SDA HIGH to indicate a read transaction. The slave then sends a byte of data to the master. If the byte received is the last byte that the master wants, the master sends a NACK signal followed by the STOP condition to end the message.

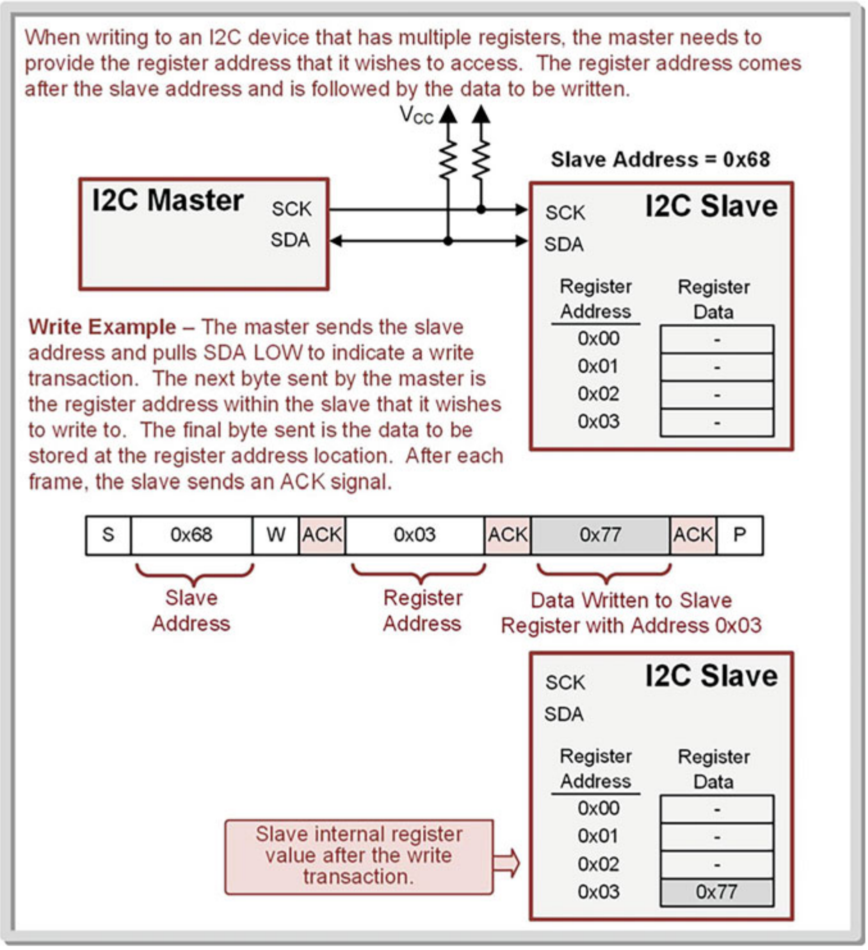
S	0x68	R	ACK	0x45	NACK	P

The NACK signal doesn't signify that the master didn't receive the data. It instead signifies to the slave not to send another byte. The NACK always comes right before the STOP condition in a read.

Master MCU Rx buffer value after the read transaction.

Fig. 14.34
I2C bus protocol when reading from a device with a single register

Many I2C devices contain blocks of registers that can be individually accessed. When an I2C device has multiple registers, each is assigned a *register address*. To access a specific register in the slave, the master needs to provide the register address in the message. For a write transaction, the master generates an I2C message that first provides the slave address, then provides the register address to access, and then provides the data to be written. After each frame, the slave sends an ACK signal. Figure 14.35 shows the I2C bus protocol for writing to a specific register within an I2C device by providing the register address as the second frame within the message.



When writing to an I2C device that has multiple registers, the master needs to provide the register address that it wishes to access. The register address comes after the slave address and is followed by the data to be written.

Write Example – The master sends the slave address and pulls SDA LOW to indicate a write transaction. The next byte sent by the master is the register address within the slave that it wishes to write to. The final byte sent is the data to be stored at the register address location. After each frame, the slave sends an ACK signal.

Register Address	Register Data
0x00	-
0x01	-
0x02	-
0x03	-

S	0x68	W	ACK	0x03	ACK	0x77	ACK	P

Register Address	Register Data
0x00	-
0x01	-
0x02	-
0x03	0x77

Slave internal register value after the write transaction.

Fig. 14.35
I2C bus protocol when writing to a device with multiple registers (single byte of data)

In situations where the slave contains multiple registers, the master can also write a block of data. This is handled by the slave by automatically incrementing its register address after each byte of data is written. The master still sends the slave address, the write signal, and the starting register address. The next byte of data that is sent goes into the first register address location. The slave then increments its register address. The next byte written by the master goes into the next register address. The slave will continue to increment its register address until it sees the STOP condition generated by the master. This allows the master to write a block of data to the registers within the slave while only providing the starting address of the register array. Figure 14.36 shows the I2C bus protocol for writing a block of data to a device with multiple registers.

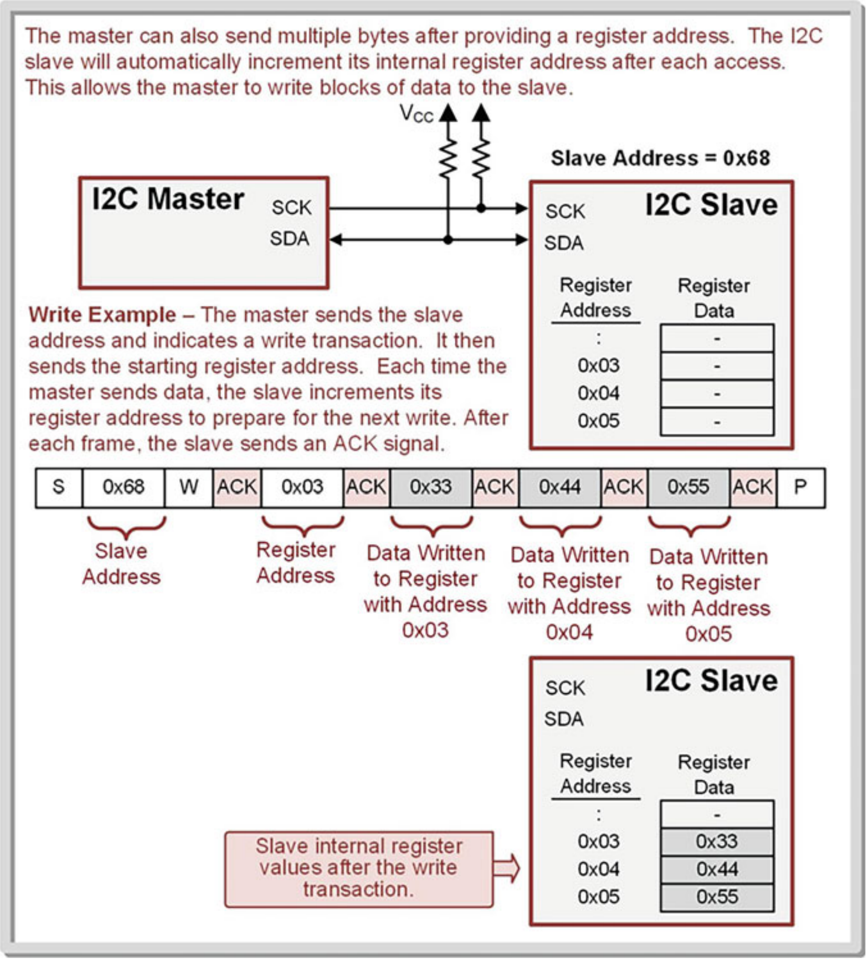
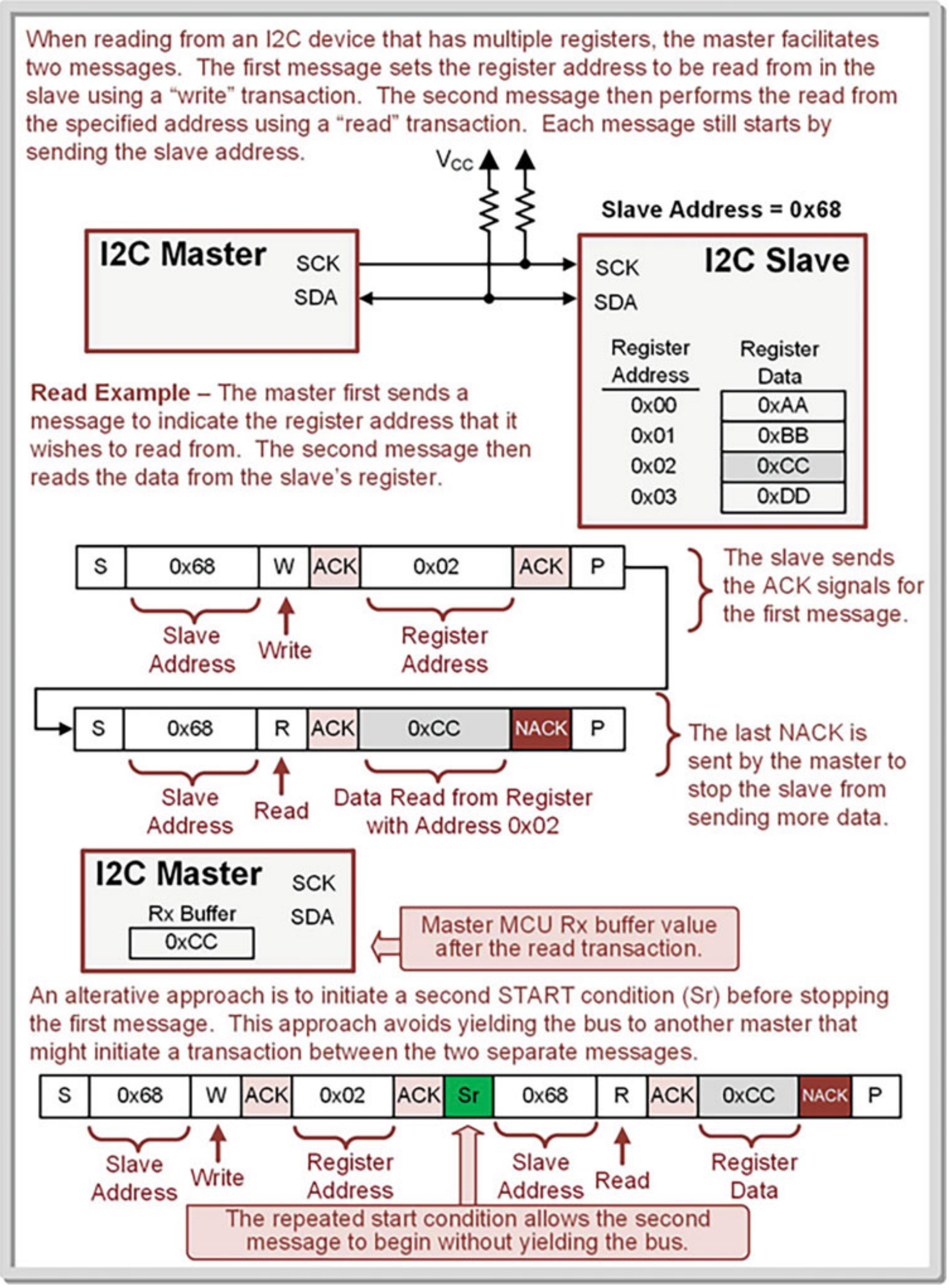

The master can also send multiple bytes after providing a register address. The I2C slave will automatically increment its internal register address after each access. This allows the master to write blocks of data to the slave.
Vcc
Slave Address = 0x68
I2C Master
SCK
SDA
I2C Slave
SCK
SDA
Register Address
Register Data
:
0x03
0x04
0x05
-
Write Example – The master sends the slave address and indicates a write transaction. It then sends the starting register address. Each time the master sends data, the slave increments its register address to prepare for the next write. After each frame, the slave sends an ACK signal.
S
0x68
W
ACK
0x03
ACK
0x33
ACK
0x44
ACK
0x55
ACK
P
Slave Address
Register Address
Data Written to Register with Address 0x03
Data Written to Register with Address 0x04
Data Written to Register with Address 0x05
I2C Slave
SCK
SDA
Register Address
Register Data
:
0x03
0x04
0x05
-
0x33
0x44
0x55
Slave internal register values after the write transaction.

Fig. 14.36
I2C bus protocol when writing to a device with multiple registers (block of data)

When reading from a device that contains multiple registers, two messages are needed. The first message sets the register address within the slave that will be read from using a write transaction. The first message puts the slave into a mode where it is expecting a second message that will read data from the register address that was just sent. The second message performs a read transaction that retrieves the data from the register address sent in the first message. An alternative approach can also be used where a second START (Sr) condition is generated before the STOP condition of the first message. This initiates the second read transaction immediately without giving up control of the bus. Figure 14.37 shows the I2C bus protocol for reading from a device with multiple registers using two messages or alternatively by producing a second start condition within a single message.



When reading from an I2C device that has multiple registers, the master facilitates two messages. The first message sets the register address to be read from in the slave using a "write" transaction. The second message then performs the read from the specified address using a "read" transaction. Each message still starts by sending the slave address.

Read Example – The master first sends a message to indicate the register address that it wishes to read from. The second message then reads the data from the slave's register.

The slave sends the ACK signals for the first message.

The last NACK is sent by the master to stop the slave from sending more data.

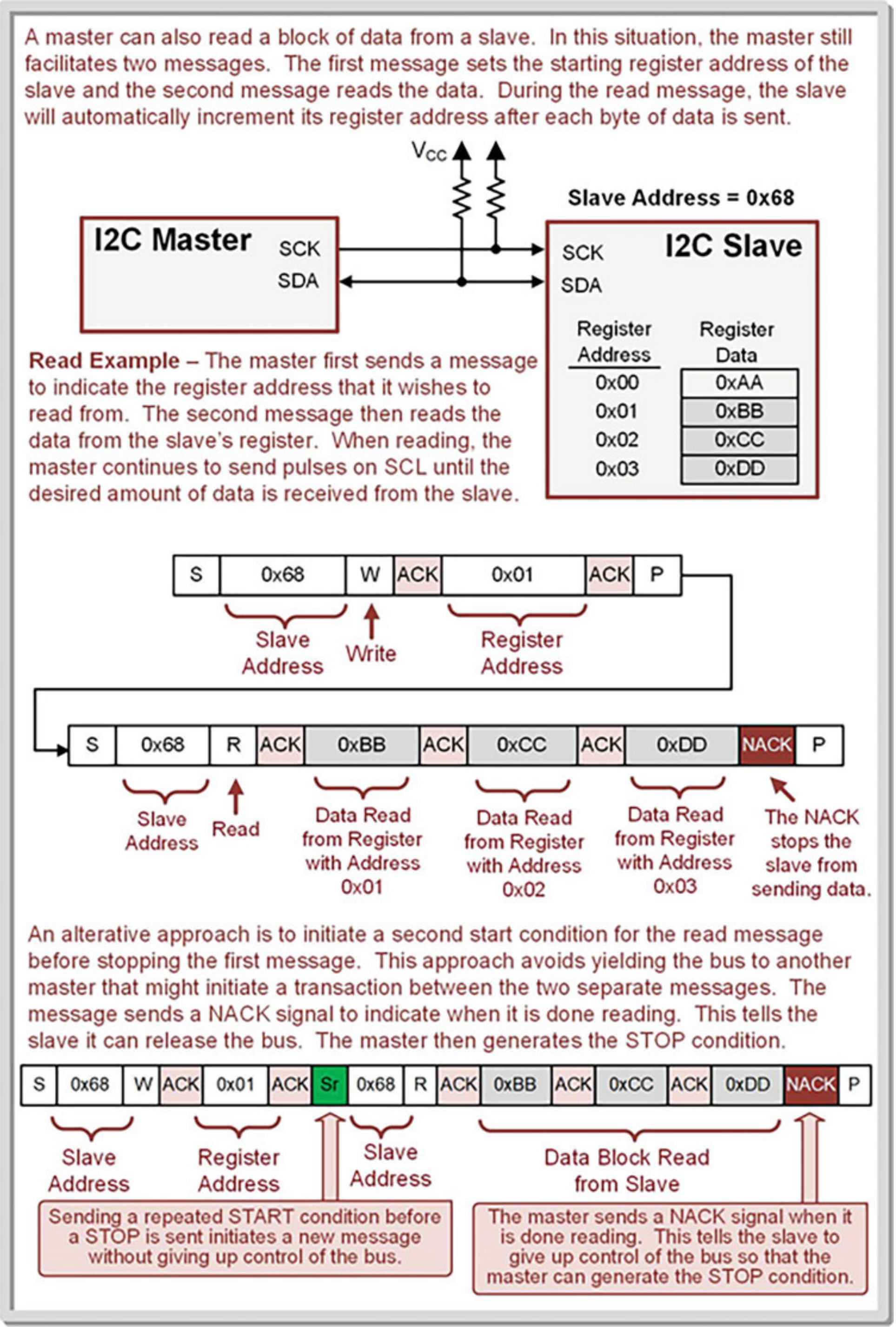
Master MCU Rx buffer value after the read transaction.

An alterative approach is to initiate a second START condition (Sr) before stopping the first message. This approach avoids yielding the bus to another master that might initiate a transaction between the two separate messages.

The repeated start condition allows the second message to begin without yielding the bus.

Fig. 14.37
I2C bus protocol when reading from a device with multiple registers (single byte of data)

The master can also read blocks of data from a slave. In this situation, the master still sends two messages, the first setting the register address and the second retrieving the data. In the second message, the master continually sends SCL pulses to read as many bytes as it wants prior to generating the STOP condition. Figure 14.38 shows the I2C bus protocol for reading a block of data from a device with multiple registers using two messages and also using a second start condition.



A master can also read a block of data from a slave. In this situation, the master still facilitates two messages. The first message sets the starting register address of the slave and the second message reads the data. During the read message, the slave will automatically increment its register address after each byte of data is sent.

Register Address	Register Data
0x00	0xAA
0x01	0xBB
0x02	0xCC
0x03	0xDD

Read Example – The master first sends a message to indicate the register address that it wishes to read from. The second message then reads the data from the slave's register. When reading, the master continues to send pulses on SCL until the desired amount of data is received from the slave.

S	0x68	W	ACK	0x01	ACK	P

S	0x68	R	ACK	0xBB	ACK	0xCC	ACK	0xDD	NACK	P

Slave Address
Read
Data Read from Register with Address 0x01
Data Read from Register with Address 0x02
Data Read from Register with Address 0x03
The NACK stops the slave from sending data.

An alterative approach is to initiate a second start condition for the read message before stopping the first message. This approach avoids yielding the bus to another master that might initiate a transaction between the two separate messages. The message sends a NACK signal to indicate when it is done reading. This tells the slave it can release the bus. The master then generates the STOP condition.

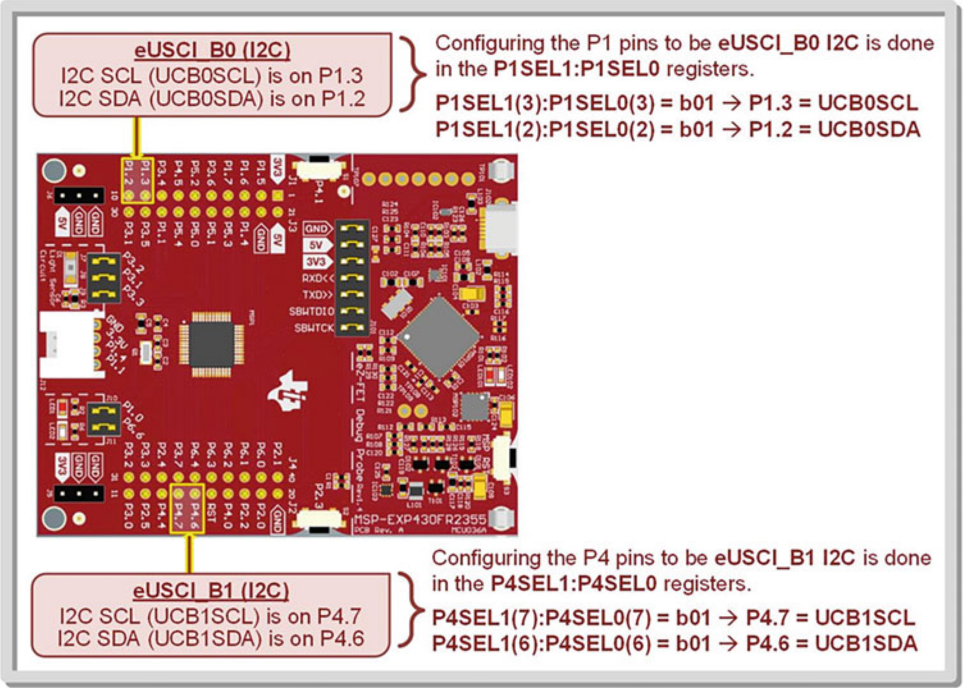
S	0x68	W	ACK	0x01	ACK	Sr	0x68	R	ACK	0xBB	ACK	0xCC	ACK	0xDD	NACK	P

Sending a repeated START condition before a STOP is sent initiates a new message without giving up control of the bus.

The master sends a NACK signal when it is done reading. This tells the slave to give up control of the bus so that the master can generate the STOP condition.

Fig. 14.38
I2C bus protocol when reading from a device with multiple registers (multiple bytes of data)

The MSP430FR2355 contains two eUSCIs that support I2C. These are eUSCI_B0 and eUSCI_B1. These two eUSCI peripherals are configurable to either support I2C or SPI. Each eUSCI has SCL and SDA signals that share pins with ports on the MCU. The eUSCI_B0 SCL/SDA pins share with port 1, bits 3 and 2, respectively. The eUSCI_B1 SCL/SDA pins share with port 4, bits 7 and 6, respectively. Figure 14.39 shows the location of the two I2C peripherals on the MSP430FR2355 LaunchPad™ board.



eUSCI_B0 (I2C)
I2C SCL (UCB0SCL) is on P1.3
I2C SDA (UCB0SDA) is on P1.2

Configuring the P1 pins to be **eUSCI_B0 I2C** is done in the **P1SEL1:P1SEL0** registers.

P1SEL1(3):P1SEL0(3) = b01 → P1.3 = UCB0SCL
P1SEL1(2):P1SEL0(2) = b01 → P1.2 = UCB0SDA

eUSCI_B1 (I2C)
I2C SCL (UCB1SCL) is on P4.7
I2C SDA (UCB1SDA) is on P4.6

Configuring the P4 pins to be **eUSCI_B1 I2C** is done in the **P4SEL1:P4SEL0** registers.

P4SEL1(7):P4SEL0(7) = b01 → P4.7 = UCB1SCL
P4SEL1(6):P4SEL0(6) = b01 → P4.6 = UCB1SDA

Fig. 14.39
eUSCI_Bx I2C pins on MSP430FR2355 LaunchPad™ board

14.3.2 I2C Master Operation on the MSP430FR2355

Let's start learning about the I2C protocol by looking at the transmit capabilities of the MSP430FR2355 in master mode. Figure 14.40 shows a general block diagram of the I2C peripheral on the MSP430FR2355.

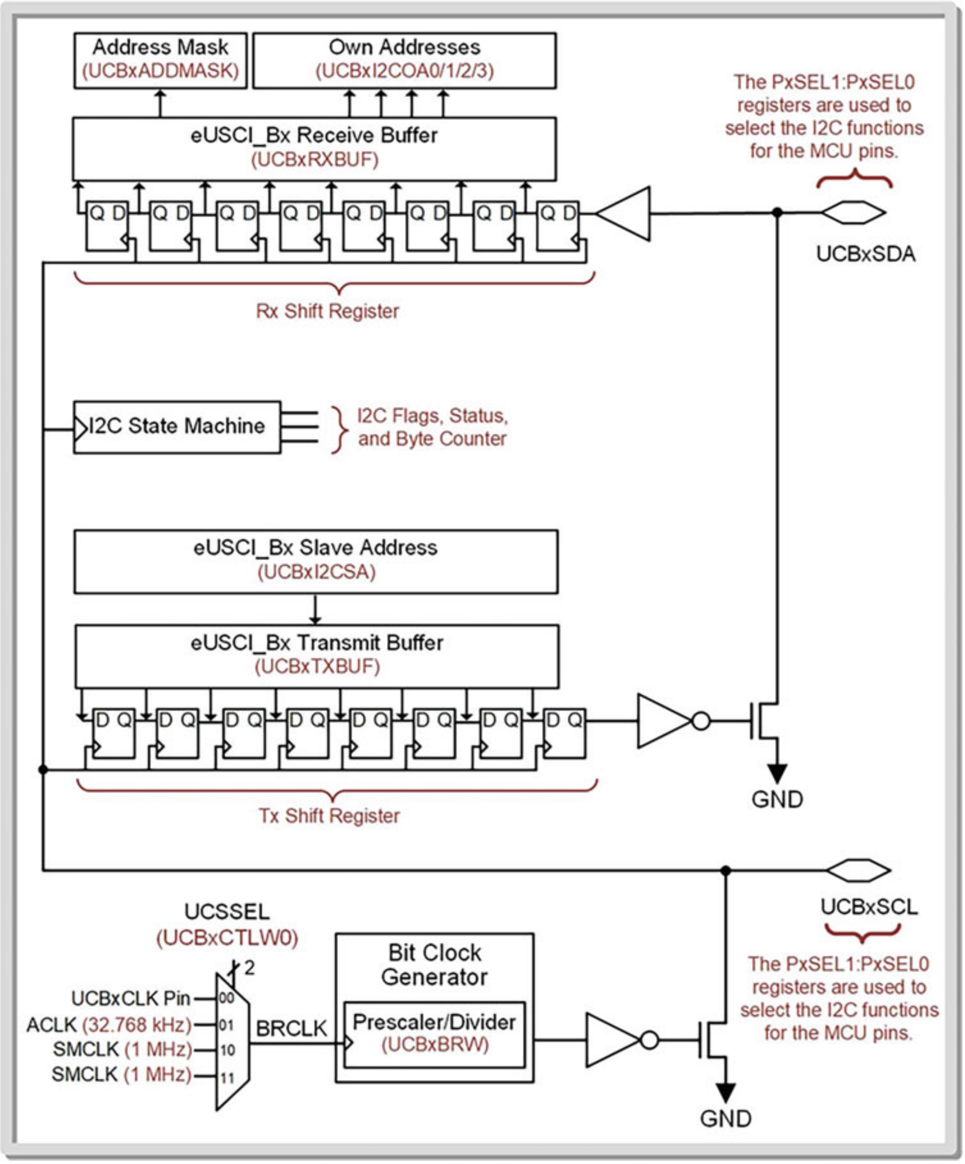


Fig. 14.40
I2C general block diagram of the eUSCI_Bx peripheral on the MSP430FR2355

The I2C system contains a variety of configuration registers that are used to trigger events, indicate when certain conditions have occurred, and hold various information such as the slave address and the data sent or received. The complete list of I2C configuration registers is below.

- *eUSCI_Bx control word 0* (UCBxCTLW0) – has different bit fields when in I2C mode
- *eUSCI_Bx control word 1* (UCBxCTLW0) – has different bit fields when in I2C mode
- *eUSCI_Bx bit rate control word* (UCBxBRW) – same function as in SPI mode

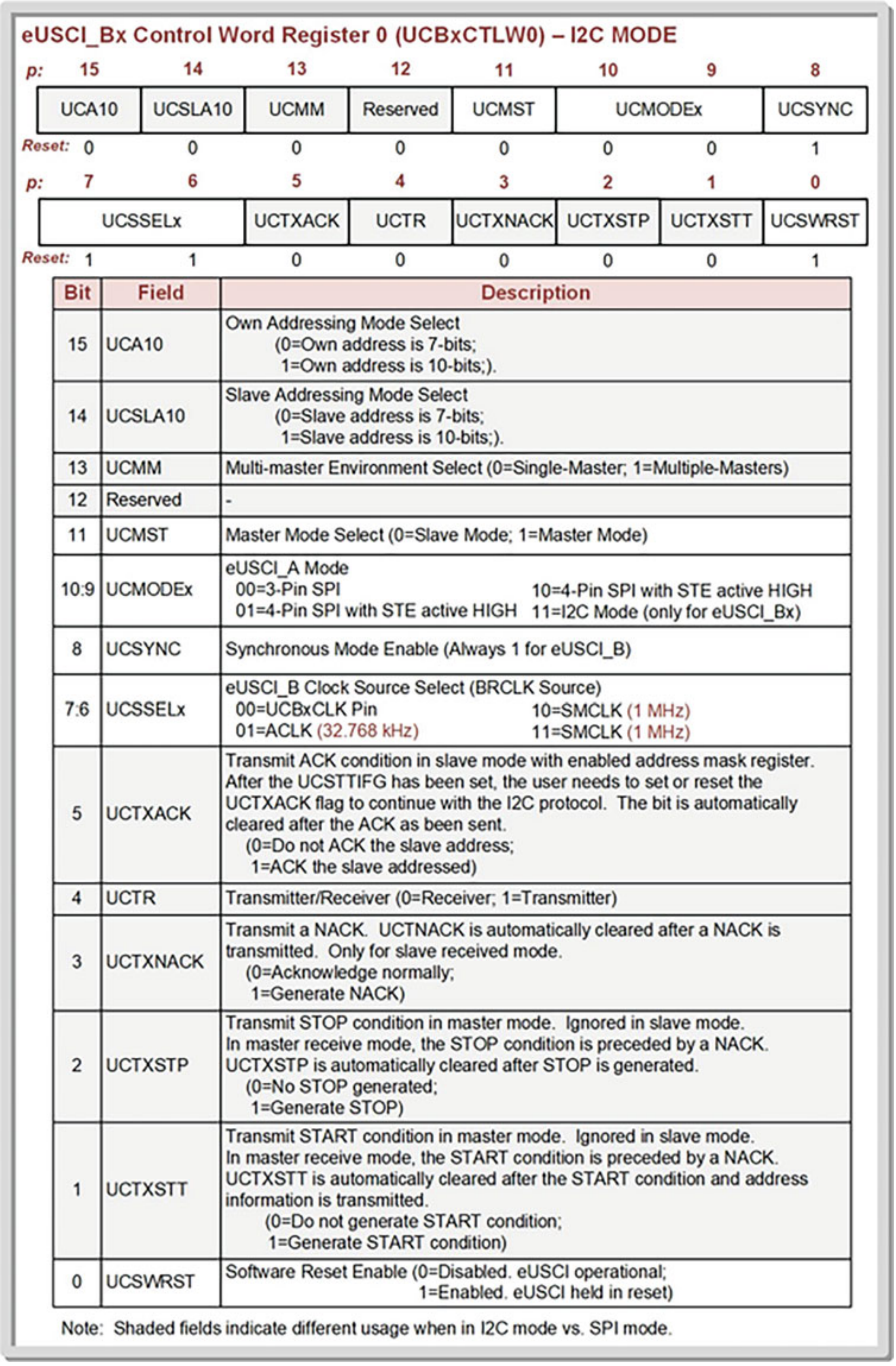
- *eUSCI_Bx status* (UCBxSTATW) – has different bit fields when in I2C mode
- *eUSCI_Bx byte counter threshold* (UCBxTBCNT)
- *eUSCI_Bx receive buffer* (UCBxRXBUF) – same function as in SPI mode
- *eUSCI_Bx transmit buffer* (UCBxTXBUF) – same function as in SPI mode
- *eUSCI_Bx I2C own address 0* (UCBxI2COA0)
- *eUSCI_Bx I2C own address 1* (UCBxI2COA1)
- *eUSCI_Bx I2C own address 2* (UCBxI2COA2)
- *eUSCI_Bx I2C own address 3* (UCBxI2COA3)
- *eUSCI_Bx received address* (UCBxADDRX)
- *eUSCI_Bx address mask* (UCBxADDMASK)
- *eUSCI_Bx I2C slave address* (UCBxI2CSA)
- *eUSCI_Bx interrupt enable* (UCBxIE) – has different bit fields when in I2C mode
- *eUSCI_Bx interrupt flag* (UCBxIFG) – has different bit fields when in I2C mode
- *eUSCI_Bx interrupt vector* (UCBxIV) – has different bit fields when in I2C mode

The recommended order of steps from the MSP430 user's guide to configure the I2C peripheral is:

1. Set the UCSWRST bit in the UCBxCTLW0 configuration register to put the eUSCI_Ax peripheral into software reset.
2. Initialize all eUSCI_Bx configuration registers.
3. Configure ports.
4. Clear UCSWRST to take the eUSCI_Ax peripheral out of reset.
5. Enable interrupts (optional) in the UCBxIE configuration register.

14.3.2.1 Writing Data as an I2C Master

The first step in setting up the I2C peripheral is to put the system into software reset to avoid erroneous data from being transmitted during setup. This is done by setting the UCSWRST bit in the eUSCI_Bx control word 0 (UCBxCTLW0) register. Upon reset, UCSWRST = 1, so eUSCI_Bx is in software reset by default; however, it is good practice to explicitly set this bit in order to ensure the system is disabled. Figure 14.41 gives the details of the UCBxCTLW0 register where the UCSWRST resides. This is the same register as is used to configure the eUSCI_Bx SPI peripherals; however, many of the fields are used for different settings when configuring the peripheral as an I2C.



eUSCI_Bx Control Word Register 0 (UCBxCTLW0) – I2C MODE

p:	15	14	13	12	11	10	9	8
	UCA10	UCSLA10	UCMM	Reserved	UCMST	UCMODEx		UCSYNC
Reset:	0	0	0	0	0	0	0	1

p:	7	6	5	4	3	2	1	0
	UCSSELx		UCTXACK	UCTR	UCTXNACK	UCTXSTP	UCTXSTT	UCSWRST
Reset:	1	1	0	0	0	0	0	1

Bit	Field	Description
15	UCA10	Own Addressing Mode Select (0=Own address is 7-bits; 1=Own address is 10-bits;).
14	UCSLA10	Slave Addressing Mode Select (0=Slave address is 7-bits; 1=Slave address is 10-bits;).
13	UCMM	Multi-master Environment Select (0=Single-Master; 1=Multiple-Masters)
12	Reserved	-
11	UCMST	Master Mode Select (0=Slave Mode; 1=Master Mode)
10:9	UCMODEx	eUSCI_A Mode 00=3-Pin SPI 10=4-Pin SPI with STE active HIGH 01=4-Pin SPI with STE active HIGH 11=I2C Mode (only for eUSCI_Bx)
8	UCSYNC	Synchronous Mode Enable (Always 1 for eUSCI_B)
7:6	UCSSELx	eUSCI_B Clock Source Select (BRCLK Source) 00=UCBxCLK Pin 10=SMCLK (1 MHz) 01=ACLK (32.768 kHz) 11=SMCLK (1 MHz)
5	UCTXACK	Transmit ACK condition in slave mode with enabled address mask register. After the UCSTTIFG has been set, the user needs to set or reset the UCTXACK flag to continue with the I2C protocol. The bit is automatically cleared after the ACK as been sent. (0=Do not ACK the slave address; 1=ACK the slave addressed)
4	UCTR	Transmitter/Receiver (0=Receiver; 1=Transmitter)
3	UCTXNACK	Transmit a NACK. UCTNACK is automatically cleared after a NACK is transmitted. Only for slave received mode. (0=Acknowledge normally; 1=Generate NACK)
2	UCTXSTP	Transmit STOP condition in master mode. Ignored in slave mode. In master receive mode, the STOP condition is preceded by a NACK. UCTXSTP is automatically cleared after STOP is generated. (0=No STOP generated; 1=Generate STOP)
1	UCTXSTT	Transmit START condition in master mode. Ignored in slave mode. In master receive mode, the START condition is preceded by a NACK. UCTXSTT is automatically cleared after the START condition and address information is transmitted. (0=Do not generate START condition; 1=Generate START condition)
0	UCSWRST	Software Reset Enable (0=Disabled. eUSCI operational; 1=Enabled. eUSCI held in reset)

Note: Shaded fields indicate different usage when in I2C mode vs. SPI mode.

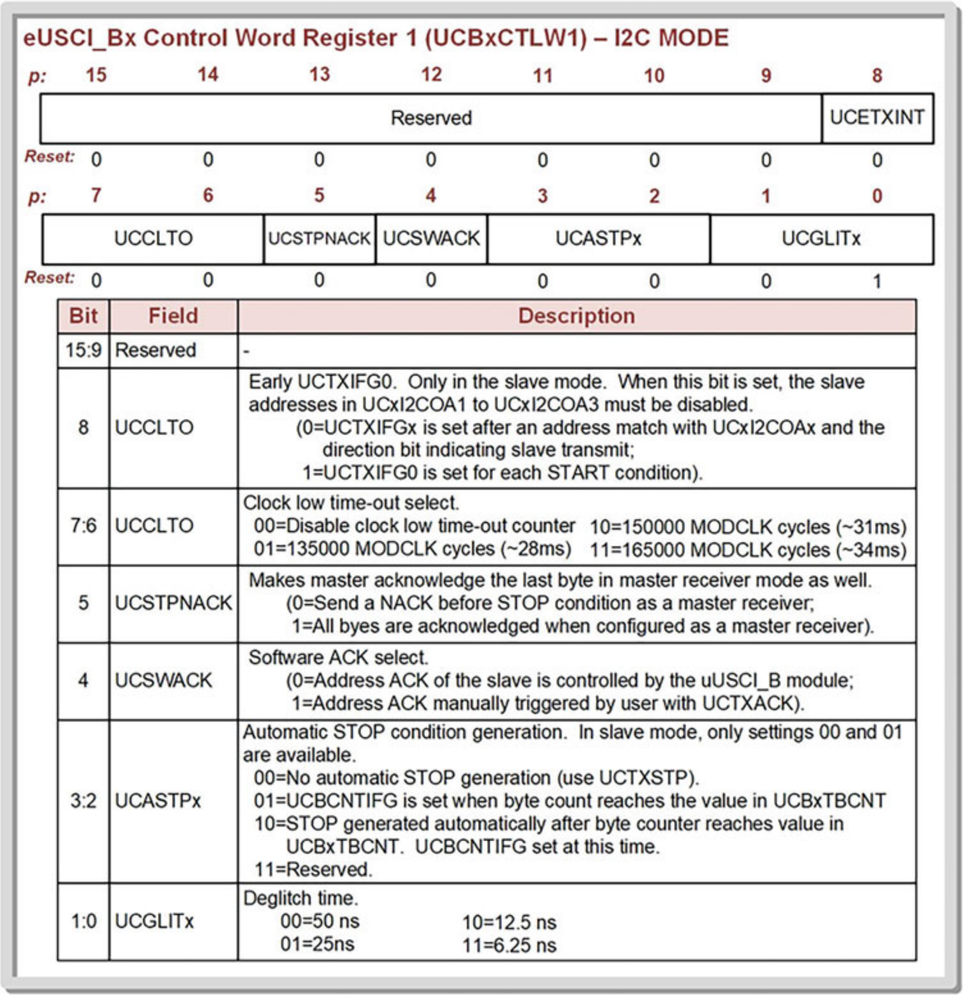
Fig. 14.41
eUSCI_Bx control word register 0 (UCBxCTLW0): I2C mode

The next configuration step that is handled in the UCBxCTLW0 register is selecting the clock for the eUSCI_Bx peripheral (i.e., BRCLK). This is done using the UCSSELx bits. When in I2C model, the default setting for UCSSELx = 11, which selects SMCLK on the MSP430FR2355. The only other choice for BRCLK is the external pin UCBxCLK (UCSSELx = 00). On the MSP430FR2355 LaunchPad™ board, we will always use the default setting of SMCLK for the BRCLK source.

The next step to set up the clock is configuring the UCBxBRW register, which is the prescaler/divider for the source clock. The value that is stored into UCBxBRW will divide the incoming clock source (i.e., BRCLK) to produce the SCL that will be used by the master and all slaves. This register is used in the same way for I2C as in SPI as a clock prescaler/divider.

Just as with the SPI interface, many of the I2C peripheral settings are configured in the UCBxCTLW0 register. The UCMODEx bits dictate whether the peripheral is set up in SPI or I2C (UCMODEx = 11 for I2C). Since only SPI and I2C are used in the eUSCI_Bx peripherals, the synchronous mode enable (UCSYNC) bit is always high. The UCMST bit in UCBxCTLW0 dictates whether the MCU will be the I2C master (UCMST = 1) or slave (UCMST = 0). The UCTR bit in UCBxCTLW0 dictates whether the MCU will be transmitting (UCTR = 1) or receiving (UCTR = 0) data. The UCA10 bit configures whether the MCU's own slave address is 7 bits or 10 bits when the MCU is the slave. The UCA10SLA bit configures whether the external slave address is 7 bits or 10 bits when the MCU is the master. There are also four bits within UCBxCTLW0 that will trigger events on the I2C bus such as generating a START condition (UCTXSTT = 1), generating a STOP condition (UCTXSTP = 1), sending an ACK (UCTXACK = 1), and sending a NACK (UCTXNACK = 1).

There are additional configuration bits for settings up a basic I2C master transmitter in the UCBxCTLW1 register. The most commonly used setting is the automatic STOP condition generation (UCASTPx). When UCASTPx = 10, the master will automatically generate the STOP condition once the desired number of data bytes have been sent or received. Figure 14.42 gives the details of the UCBxCTLW1 register where UCASTPx resides.



eUSCI_Bx Control Word Register 1 (UCBxCTLW1) – I2C MODE

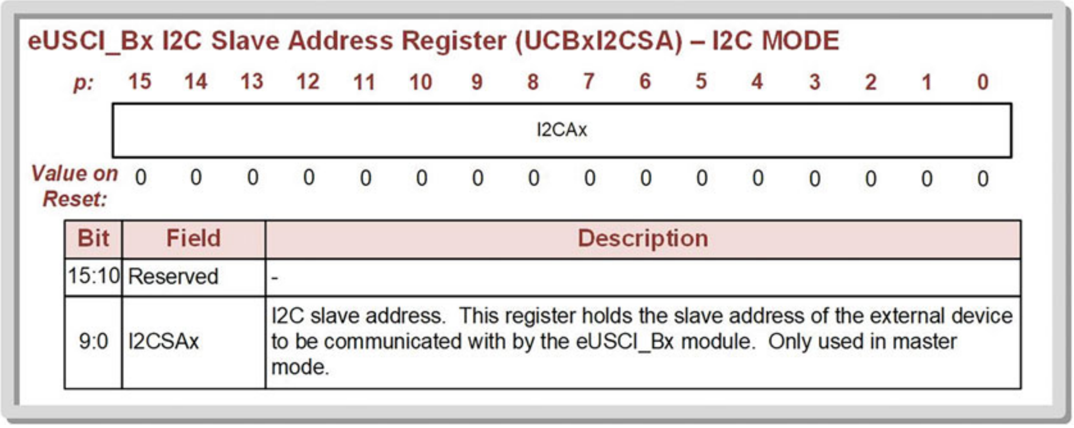
p:	15	14	13	12	11	10	9	8
	Reserved							UCETXINT
Reset:	0	0	0	0	0	0	0	0

p:	7	6	5	4	3	2	1	0
	UCCLTO		UCSTPNACK	UCSWACK	UCASTPx		UCGLITx	
Reset:	0	0	0	0	0	0	0	1

Bit	Field	Description
15:9	Reserved	-
8	UCCLTO	Early UCTXIFG0. Only in the slave mode. When this bit is set, the slave addresses in UCxI2COA1 to UCxI2COA3 must be disabled. (0=UCTXIFGx is set after an address match with UCxI2COAx and the direction bit indicating slave transmit; 1=UCTXIFG0 is set for each START condition).
7:6	UCCLTO	Clock low time-out select. 00=Disable clock low time-out counter 10=150000 MODCLK cycles (~31ms) 01=135000 MODCLK cycles (~28ms) 11=165000 MODCLK cycles (~34ms)
5	UCSTPNACK	Makes master acknowledge the last byte in master receiver mode as well. (0=Send a NACK before STOP condition as a master receiver; 1=All byes are acknowledged when configured as a master receiver).
4	UCSWACK	Software ACK select. (0=Address ACK of the slave is controlled by the uUSCI_B module; 1=Address ACK manually triggered by user with UCTXACK).
3:2	UCASTPx	Automatic STOP condition generation. In slave mode, only settings 00 and 01 are available. 00=No automatic STOP generation (use UCTXSTP). 01=UCBCNTIFG is set when byte count reaches the value in UCBxTBCNT 10=STOP generated automatically after byte counter reaches value in UCBxTBCNT. UCBCNTIFG set at this time. 11=Reserved.
1:0	UCGLITx	Deglitch time. 00=50 ns 10=12.5 ns 01=25ns 11=6.25 ns

Fig. 14.42
eUSCI_Bx control word register 1 (UCBxCTLW1): I2C mode

The slave address that the master wishes to communicate with is configured in the UCBxI2CSA register. Figure 14.43 gives the details of the UCBxI2CSA register.



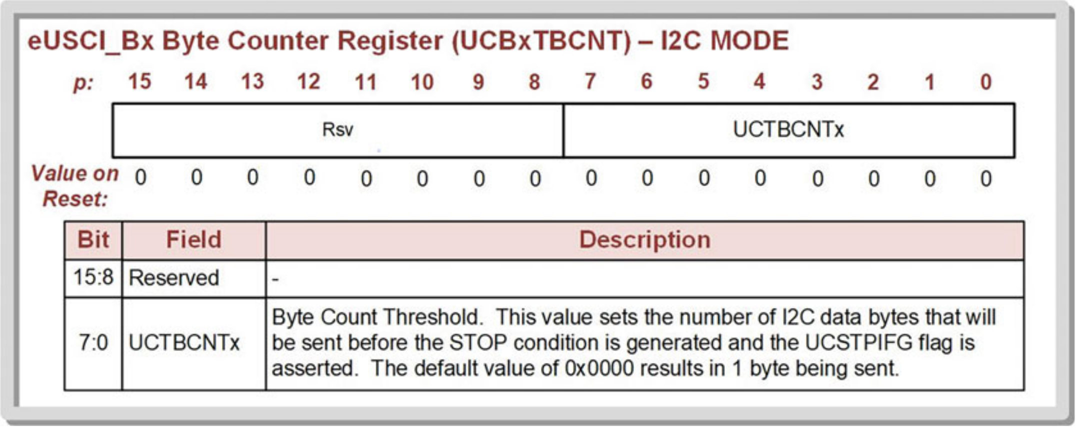
eUSCI_Bx I2C Slave Address Register (UCBxI2CSA) – I2C MODE

p:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	I2CAx															
Value on Reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	Field	Description
15:10	Reserved	-
9:0	I2CSAx	I2C slave address. This register holds the slave address of the external device to be communicated with by the eUSCI_Bx module. Only used in master mode.

Fig. 14.43
eUSCI_Bx_I2C slave address register (UCBxI2CSA)

The last configuration needed to set up a basic master write transmission is the number of bytes that are to be sent using the UCBxTBCNT register. The MCU will continue to send until the number of bytes transferred matches the value in UCBxTBCNT. After the desired number of bytes has been transferred, the master will automatically generate the STOP condition if UCASTPx = 10. Figure 14.44 gives the details of the UCBxTBCNT register.



eUSCI_Bx Byte Counter Register (UCBxTBCNT) – I2C MODE

p:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Rsv								UCTBCNTx							
Value on Reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

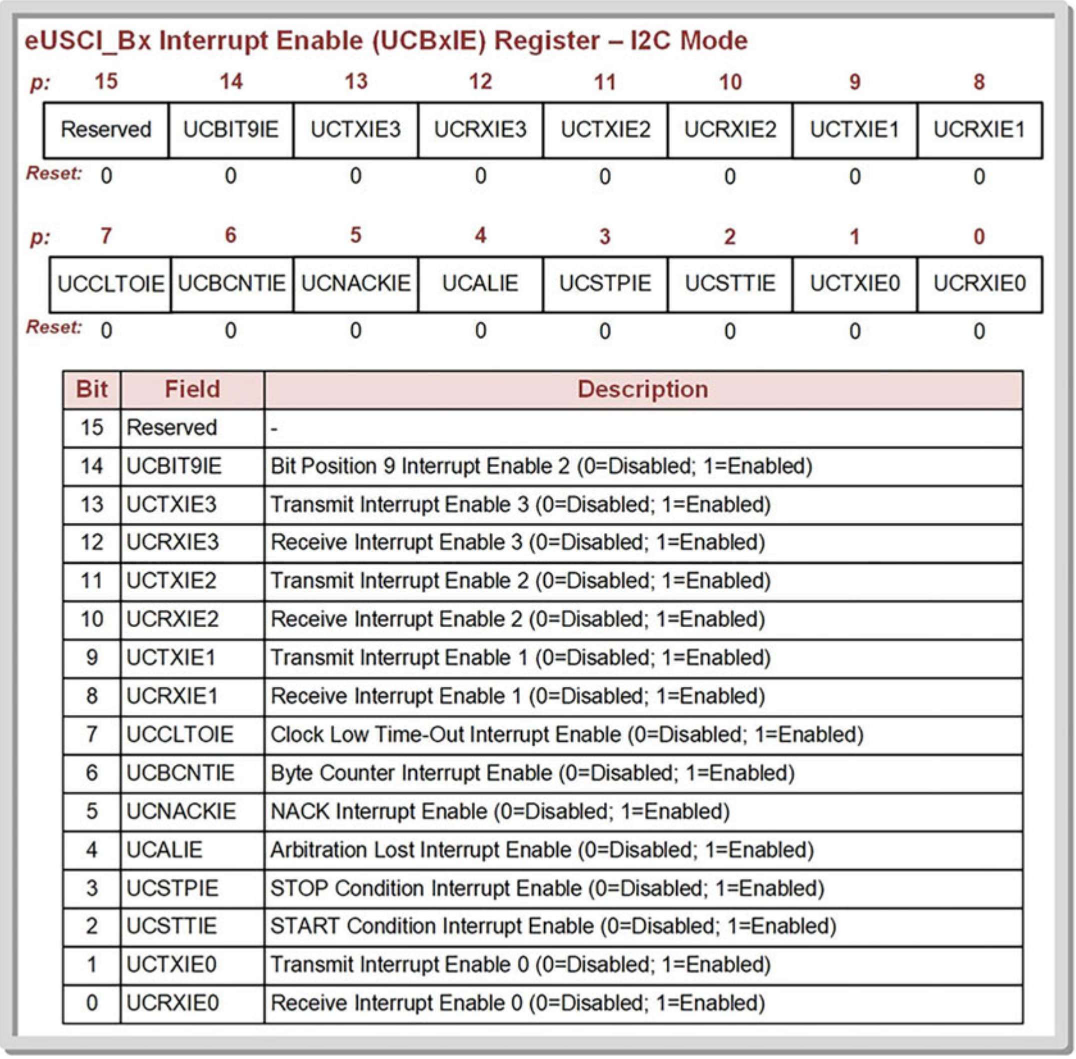
Bit	Field	Description
15:8	Reserved	-
7:0	UCTBCNTx	Byte Count Threshold. This value sets the number of I2C data bytes that will be sent before the STOP condition is generated and the UCSTPIFG flag is asserted. The default value of 0x0000 results in 1 byte being sent.

Fig. 14.44
eUSCI_Bx byte counter register (UCBxTBCNT): I2C mode

At this point, we have covered all of the eUSCI_Bx settings to configure the I2C system for a basic master write transaction. The next step is to configure the ports on the MSP430FR2355 to use the SCL and SDA. Just as in the other serial peripherals, this is done using the PxSEL1 and PxSEL0 configuration registers. The settings for these port configuration registers were given in Table 14.2 in Sect. 14.1.2.

At this point if we were using UART or SPI, the peripheral would be set up enough so that we could simply store information into the Tx buffer, and it would be shifted out; however, in I2C, the Tx buffer can only be written to within the eUSCI_Bx interrupt service routine. This ensures that the data is only shifted out after other events in the protocol have occurred, such as sending the slave address and receiving an

ACK from the slave. The MSP430 contains a large number of interrupt capabilities that help automate the process of sending and receiving portions of the I2C message. These interrupts are enabled in the UCBxIE register (shown in Fig. 14.45), have flags in the ICBxIFG register (shown in Fig. 14.46), and can be prioritized in the event of simultaneous interrupts in the UCBxIV register (shown in Fig. 14.47). Note that these three registers are the same as used when the eUSCI_Bx peripherals are configured in SPI mode, but with different bit fields.



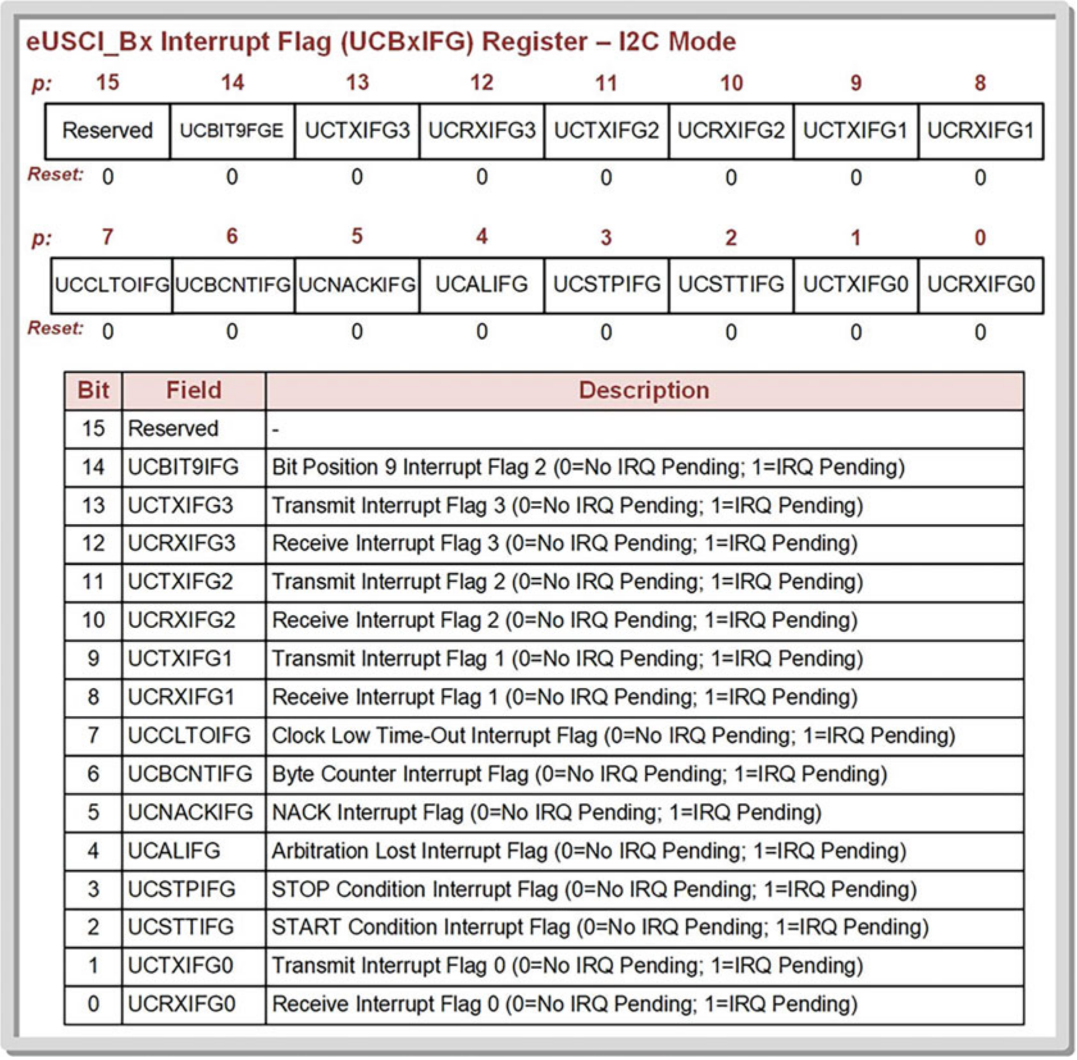
eUSCI_Bx Interrupt Enable (UCBxIE) Register – I2C Mode

p:	15	14	13	12	11	10	9	8
	Reserved	UCBIT9IE	UCTXIE3	UCRXIE3	UCTXIE2	UCRXIE2	UCTXIE1	UCRXIE1
Reset:	0	0	0	0	0	0	0	0

p:	7	6	5	4	3	2	1	0
	UCCLTOIE	UCBCNTIE	UCNACKIE	UCALIE	UCSTPIE	UCSTTIE	UCTXIE0	UCRXIE0
Reset:	0	0	0	0	0	0	0	0

Bit	Field	Description
15	Reserved	-
14	UCBIT9IE	Bit Position 9 Interrupt Enable 2 (0=Disabled; 1=Enabled)
13	UCTXIE3	Transmit Interrupt Enable 3 (0=Disabled; 1=Enabled)
12	UCRXIE3	Receive Interrupt Enable 3 (0=Disabled; 1=Enabled)
11	UCTXIE2	Transmit Interrupt Enable 2 (0=Disabled; 1=Enabled)
10	UCRXIE2	Receive Interrupt Enable 2 (0=Disabled; 1=Enabled)
9	UCTXIE1	Transmit Interrupt Enable 1 (0=Disabled; 1=Enabled)
8	UCRXIE1	Receive Interrupt Enable 1 (0=Disabled; 1=Enabled)
7	UCCLTOIE	Clock Low Time-Out Interrupt Enable (0=Disabled; 1=Enabled)
6	UCBCNTIE	Byte Counter Interrupt Enable (0=Disabled; 1=Enabled)
5	UCNACKIE	NACK Interrupt Enable (0=Disabled; 1=Enabled)
4	UCALIE	Arbitration Lost Interrupt Enable (0=Disabled; 1=Enabled)
3	UCSTPIE	STOP Condition Interrupt Enable (0=Disabled; 1=Enabled)
2	UCSTTIE	START Condition Interrupt Enable (0=Disabled; 1=Enabled)
1	UCTXIE0	Transmit Interrupt Enable 0 (0=Disabled; 1=Enabled)
0	UCRXIE0	Receive Interrupt Enable 0 (0=Disabled; 1=Enabled)

Fig. 14.45
eUSCI_Bx interrupt enable (UCBxIE) register: I2C mode



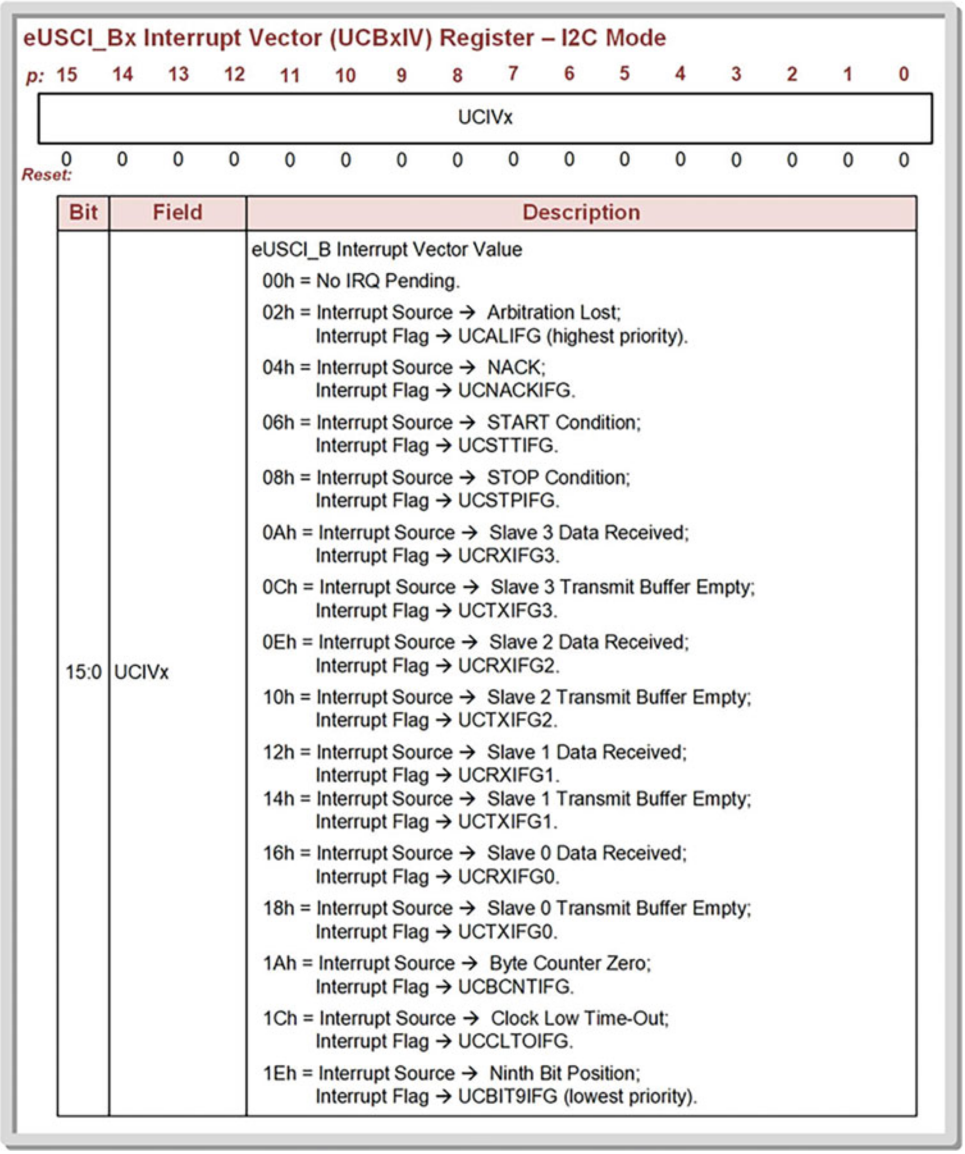
eUSCI_Bx Interrupt Flag (UCBxIFG) Register – I2C Mode

p:	15	14	13	12	11	10	9	8
	Reserved	UCBIT9FGE	UCTXIFG3	UCRXIFG3	UCTXIFG2	UCRXIFG2	UCTXIFG1	UCRXIFG1
Reset:	0	0	0	0	0	0	0	0

p:	7	6	5	4	3	2	1	0
	UCCLTOIFG	UCBCNTIFG	UCNACKIFG	UCALIFG	UCSTPIFG	UCSTTIFG	UCTXIFG0	UCRXIFG0
Reset:	0	0	0	0	0	0	0	0

Bit	Field	Description
15	Reserved	-
14	UCBIT9IFG	Bit Position 9 Interrupt Flag 2 (0=No IRQ Pending; 1=IRQ Pending)
13	UCTXIFG3	Transmit Interrupt Flag 3 (0=No IRQ Pending; 1=IRQ Pending)
12	UCRXIFG3	Receive Interrupt Flag 3 (0=No IRQ Pending; 1=IRQ Pending)
11	UCTXIFG2	Transmit Interrupt Flag 2 (0=No IRQ Pending; 1=IRQ Pending)
10	UCRXIFG2	Receive Interrupt Flag 2 (0=No IRQ Pending; 1=IRQ Pending)
9	UCTXIFG1	Transmit Interrupt Flag 1 (0=No IRQ Pending; 1=IRQ Pending)
8	UCRXIFG1	Receive Interrupt Flag 1 (0=No IRQ Pending; 1=IRQ Pending)
7	UCCLTOIFG	Clock Low Time-Out Interrupt Flag (0=No IRQ Pending; 1=IRQ Pending)
6	UCBCNTIFG	Byte Counter Interrupt Flag (0=No IRQ Pending; 1=IRQ Pending)
5	UCNACKIFG	NACK Interrupt Flag (0=No IRQ Pending; 1=IRQ Pending)
4	UCALIFG	Arbitration Lost Interrupt Flag (0=No IRQ Pending; 1=IRQ Pending)
3	UCSTPIFG	STOP Condition Interrupt Flag (0=No IRQ Pending; 1=IRQ Pending)
2	UCSTTIFG	START Condition Interrupt Flag (0=No IRQ Pending; 1=IRQ Pending)
1	UCTXIFG0	Transmit Interrupt Flag 0 (0=No IRQ Pending; 1=IRQ Pending)
0	UCRXIFG0	Receive Interrupt Flag 0 (0=No IRQ Pending; 1=IRQ Pending)

Fig. 14.46
eUSCI_Bx interrupt flag (UCBxIFG) register: I2C mode



eUSCI_Bx Interrupt Vector (UCBxIV) Register – I2C Mode

p:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	UCIVx															
Reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	Field	Description
15:0	UCIVx	eUSCI_B Interrupt Vector Value 00h = No IRQ Pending. 02h = Interrupt Source → Arbitration Lost; Interrupt Flag → UCALIFG (highest priority). 04h = Interrupt Source → NACK; Interrupt Flag → UCNACKIFG. 06h = Interrupt Source → START Condition; Interrupt Flag → UCSTTIFG. 08h = Interrupt Source → STOP Condition; Interrupt Flag → UCSTPIFG. 0Ah = Interrupt Source → Slave 3 Data Received; Interrupt Flag → UCRXIFG3. 0Ch = Interrupt Source → Slave 3 Transmit Buffer Empty; Interrupt Flag → UCTXIFG3. 0Eh = Interrupt Source → Slave 2 Data Received; Interrupt Flag → UCRXIFG2. 10h = Interrupt Source → Slave 2 Transmit Buffer Empty; Interrupt Flag → UCTXIFG2. 12h = Interrupt Source → Slave 1 Data Received; Interrupt Flag → UCRXIFG1. 14h = Interrupt Source → Slave 1 Transmit Buffer Empty; Interrupt Flag → UCTXIFG1. 16h = Interrupt Source → Slave 0 Data Received; Interrupt Flag → UCRXIFG0. 18h = Interrupt Source → Slave 0 Transmit Buffer Empty; Interrupt Flag → UCTXIFG0. 1Ah = Interrupt Source → Byte Counter Zero; Interrupt Flag → UCBCNTIFG. 1Ch = Interrupt Source → Clock Low Time-Out; Interrupt Flag → UCCLTOIFG. 1Eh = Interrupt Source → Ninth Bit Position; Interrupt Flag → UCBIT9IFG (lowest priority).

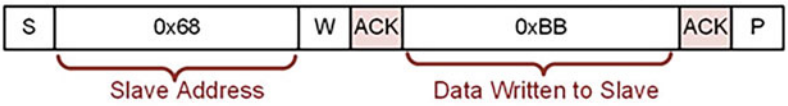
Fig. 14.47
eUSCI_Bx interrupt vector (UCBxIE) register: I2C mode

At this point we have enough background information to design our first I2C program. Let's design a program that will configure the eUSCI_B0 peripheral as an I2C master and transmit one byte to a slave. The address of the slave will be 0 × 68 and is set in the UCB0I2CSA register during initialization. We will write to the slave as if it only has one internal register, so we won't provide a register address. Our program will use a while() loop to generate a START condition periodically by asserting the UCTXSTT bit within the UCB0CTL0 register, followed by a delay. When UCTXSTT is asserted, the MSP430 will generate the START condition and automatically send out the slave address held in UCB0I2CSA. It will then wait for an ACK from the slave. Once an ACK is received, the UCTXFLG0 will be asserted and an eUSCI_B0 interrupt will be triggered. Within the eUSCI_B0 service routine, we will write data to the Tx

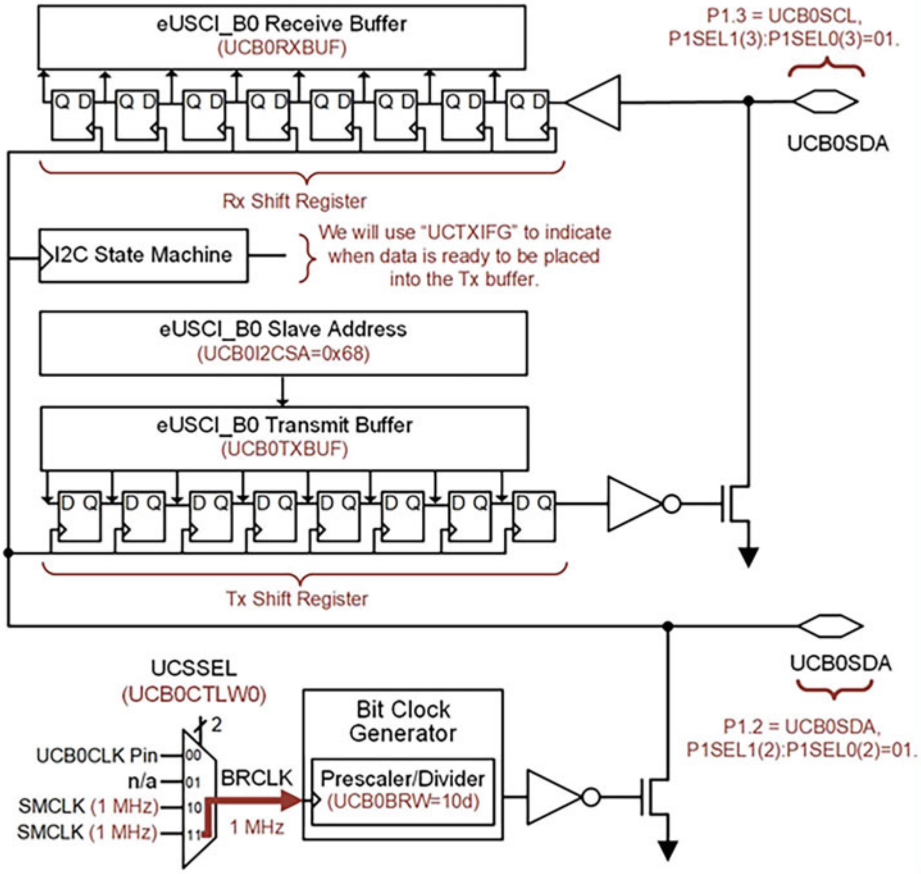
Buffer (UCTXBUF). We will use the automatic STOP generation feature of the MSP430 (UCASTP = 10) in order to end the message after the number of bytes sent matches the value in UCB0TBCNT. By configuring UCB0TBCNT = 1, the message will only send one byte and then stop. Follow the design in Examples 14.36, 14.37, and 14.38 to see how to use the I2C peripheral in master transmit mode to send one byte to a slave[13].

EXAMPLE: TRANSMITTING ONE BYTE FROM AN I2C MASTER (PART 1)

Let's configure the eUSCI_B0 peripheral as an I2C master and continually transmit the value **0xBB**. The receiving slave device in this example is the *Adafruit PCF8523 Real Time Clock* with a hardcoded slave address of **0x68**. This is the I2C message we are trying to create:



Note that to run this example a slave device needs to be connected to the LaunchPad™ board to provide the pull-up resistors and the ACK signals. For eUSCI_B0, SCL is on P1.3 and SDA is on P1.2. We will observe SCL and SDA using a logic analyzer. We will use SMCLK as the source clock (BRCLK) and divide it down by 10 to produce an SCL=100 kHz. We will use automatic STOP generation and UCBTCNT=1 to handle stopping the message after 0xBB has been sent. The following is the eUSCI_B0 setup for this example.

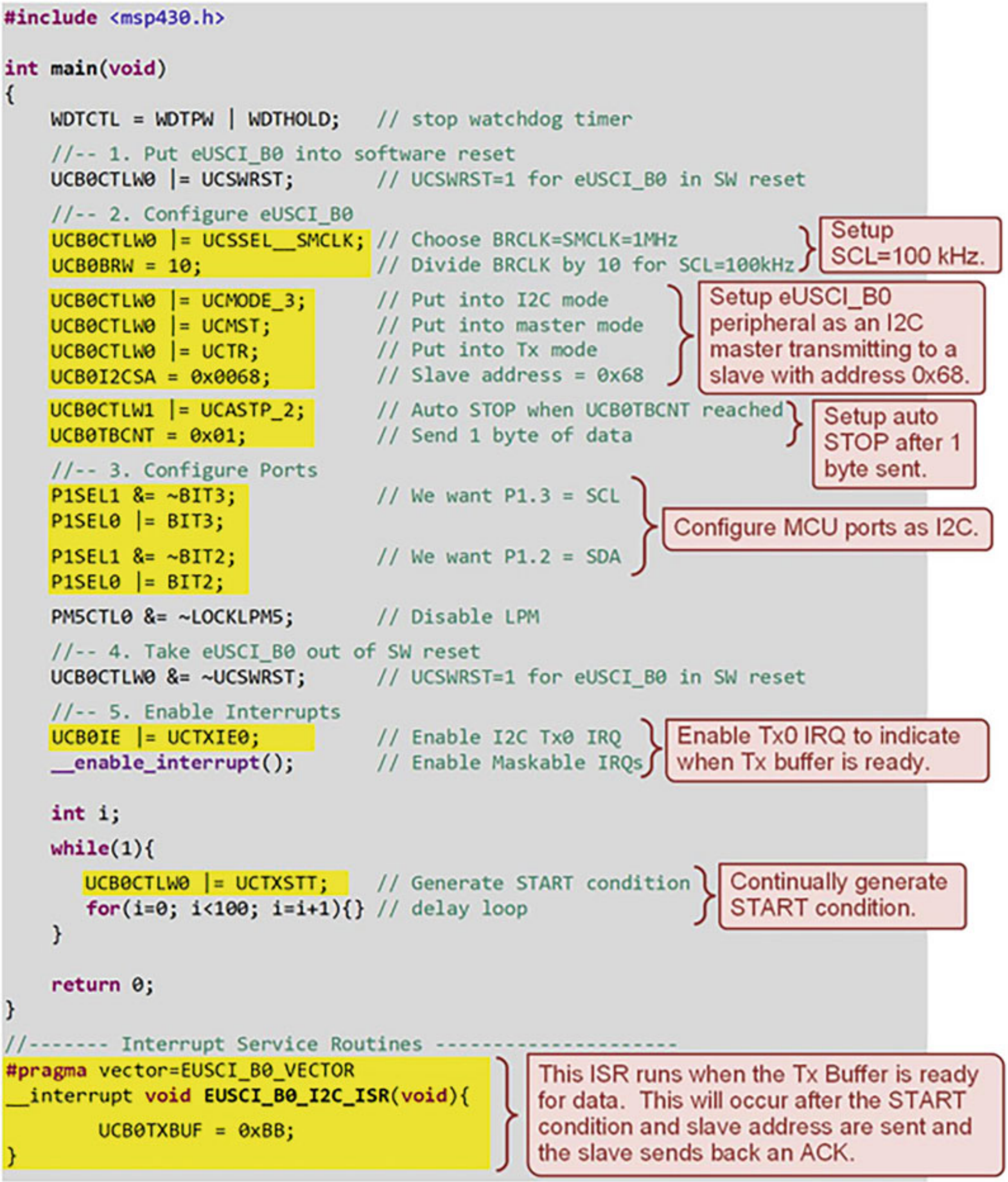


Example 14.36
Transmitting one byte from an I2C master (part 1)

Example: Transmitting One Byte From an I2C Master (Part 2)

1) In CCS, create a new C/C++ Empty Project (with main.c) titled:
C_I2C_Tx1_Master_Write_1Byte.

2) Type in the following code in main.c after the statement to stop the watchdog timer.



```c
#include <msp430.h>

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

    //-- 1. Put eUSCI_B0 into software reset
    UCB0CTLW0 |= UCSWRST;       // UCSWRST=1 for eUSCI_B0 in SW reset

    //-- 2. Configure eUSCI_B0
    UCB0CTLW0 |= UCSSEL__SMCLK; // Choose BRCLK=SMCLK=1MHz
    UCB0BRW = 10;               // Divide BRCLK by 10 for SCL=100kHz

    UCB0CTLW0 |= UCMODE_3;      // Put into I2C mode
    UCB0CTLW0 |= UCMST;         // Put into master mode
    UCB0CTLW0 |= UCTR;          // Put into Tx mode
    UCB0I2CSA = 0x0068;         // Slave address = 0x68

    UCB0CTLW1 |= UCASTP_2;      // Auto STOP when UCB0TBCNT reached
    UCB0TBCNT = 0x01;           // Send 1 byte of data

    //-- 3. Configure Ports
    P1SEL1 &= ~BIT3;            // We want P1.3 = SCL
    P1SEL0 |= BIT3;

    P1SEL1 &= ~BIT2;            // We want P1.2 = SDA
    P1SEL0 |= BIT2;

    PM5CTL0 &= ~LOCKLPM5;       // Disable LPM

    //-- 4. Take eUSCI_B0 out of SW reset
    UCB0CTLW0 &= ~UCSWRST;      // UCSWRST=1 for eUSCI_B0 in SW reset

    //-- 5. Enable Interrupts
    UCB0IE |= UCTXIE0;          // Enable I2C Tx0 IRQ
    __enable_interrupt();       // Enable Maskable IRQs

    int i;
    while(1){
        UCB0CTLW0 |= UCTXSTT;   // Generate START condition
        for(i=0; i<100; i=i+1){} // delay loop
    }

    return 0;
}
//------- Interrupt Service Routines ---------------------
#pragma vector=EUSCI_B0_VECTOR
__interrupt void EUSCI_B0_I2C_ISR(void){
        UCB0TXBUF = 0xBB;
}
```

Setup SCL=100 kHz.

Setup eUSCI_B0 peripheral as an I2C master transmitting to a slave with address 0x68.

Setup auto STOP after 1 byte sent.

Configure MCU ports as I2C.

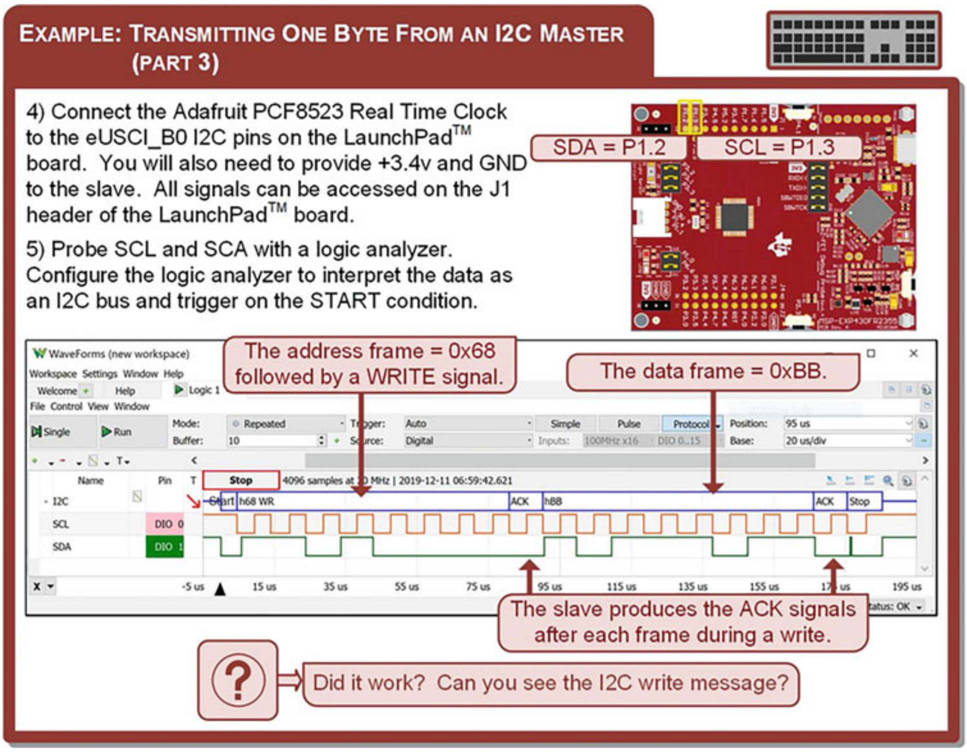
Enable Tx0 IRQ to indicate when Tx buffer is ready.

Continually generate START condition.

This ISR runs when the Tx Buffer is ready for data. This will occur after the START condition and slave address are sent and the slave sends back an ACK.

3) Save and debug your program. → We'll connect the logic analyzer next.

Example 14.37
Transmitting one byte from an I2C master (part 2)



EXAMPLE: TRANSMITTING ONE BYTE FROM AN I2C MASTER (PART 3)

4) Connect the Adafruit PCF8523 Real Time Clock to the eUSCI_B0 I2C pins on the LaunchPad™ board. You will also need to provide +3.4v and GND to the slave. All signals can be accessed on the J1 header of the LaunchPad™ board.

5) Probe SCL and SCA with a logic analyzer. Configure the logic analyzer to interpret the data as an I2C bus and trigger on the START condition.

SDA = P1.2

SCL = P1.3

The address frame = 0x68 followed by a WRITE signal.

The data frame = 0xBB.

WaveForms (new workspace)

Mode: Repeated

Buffer: 10

Trigger: Auto

Source: Digital

Inputs: 100MHz x16 DIO 0..15

Position: 95 us

Base: 20 us/div

4096 samples at 10 MHz | 2019-12-11 06:59:42.621

Name	Pin	
I2C		Start h68 WR ACK hBB ACK Stop
SCL	DIO 0	
SDA	DIO 1	

Status: OK

The slave produces the ACK signals after each frame during a write.

Did it work? Can you see the I2C write message?

Example 14.38
Transmitting one byte from an I2C master (part 3)

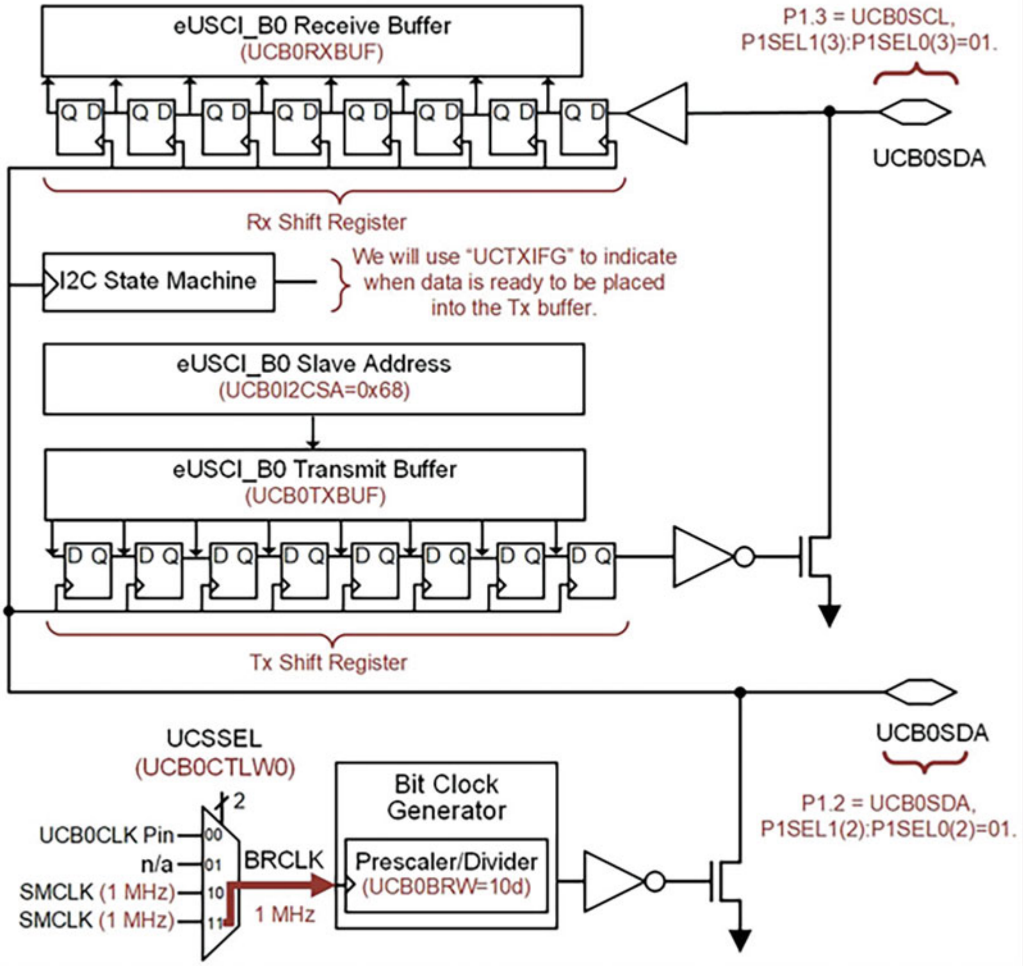
Now let's look at a program that will configure the MSP430FR2355 as an I2C master in transmit mode, but this time send out a register address followed by three bytes of data to write. In this program, we will create an array that holds four bytes, the register address followed by the three bytes of data. In order to tell the peripheral to send four bytes before generating the STOP condition, we will set UCB0TCNT = 4. Each time a byte of data is ready to be put into the Tx buffer for transmit, the UCTXIFG0 will assert and the eUSCI_B0 interrupt will trigger. Within the ISR, we will insert logic to store the next byte of data in our array into the Tx buffer. Follow the design in Examples 14.39, 14.40, and 14.41 to see how to use the I2C peripheral in master transmit mode to send the slave register address followed by three bytes of data.

Example: Transmitting a Register Address and Three Bytes of Data From an I2C Master (Part 1)

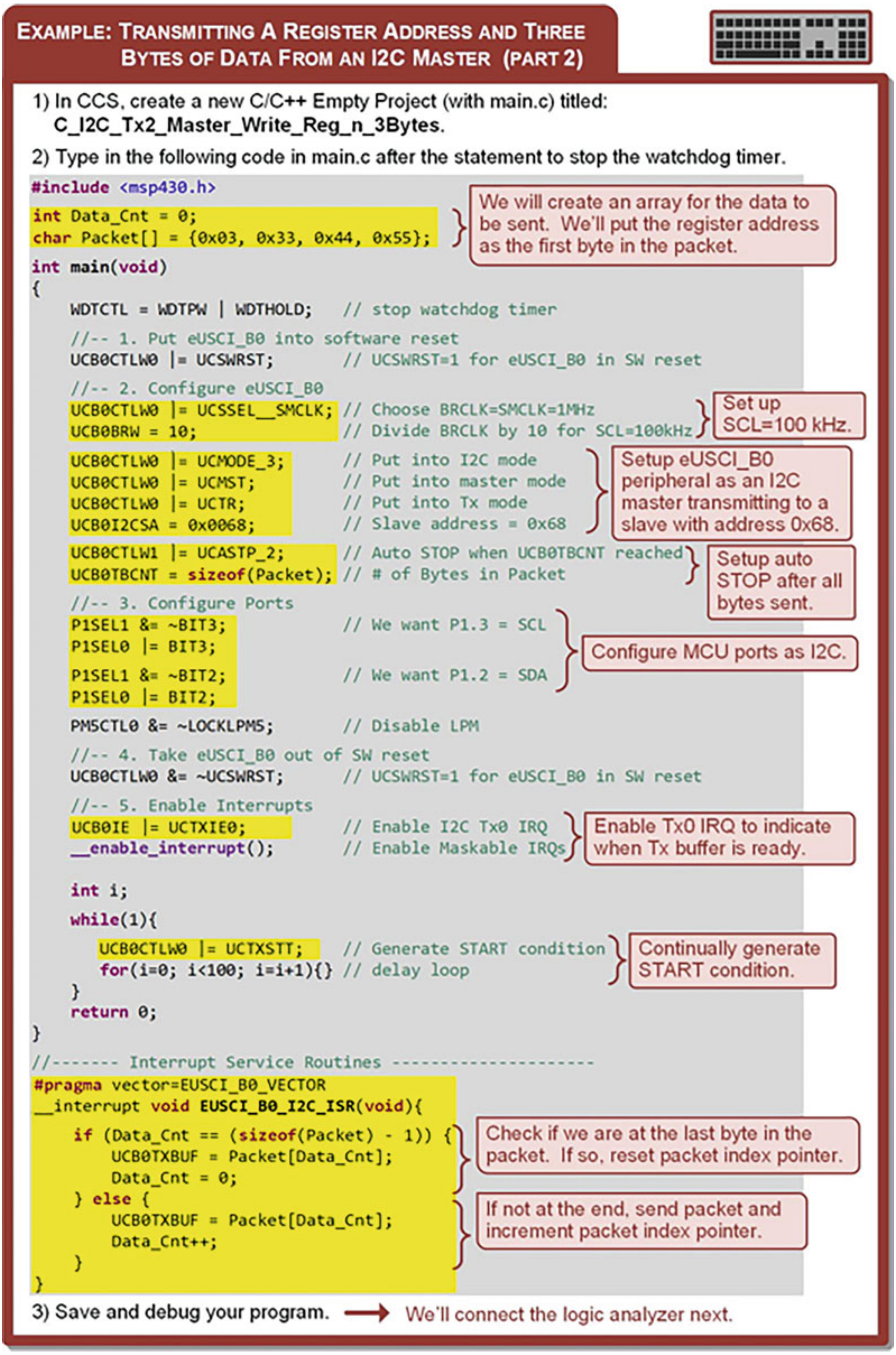
Let's configure the eUSCI_B0 peripheral as an I2C master and transmit 3x bytes of data **{0x33, 0x44, 0x55}** to a slave starting at the slave's register address **0x03**. The receiving slave in this example is the *Adafruit PCF8523 Real Time Clock* with a hard-coded slave address of **0x68**. This is the I2C message we are trying to create:

S	0x68	W	ACK	0x03	ACK	0x33	ACK	0x44	ACK	0x55	ACK	P
	Slave Address			Register Address		Data Written to Register with Address 0x03		Data Written to Register with Address 0x04		Data Written to Register with Address 0x05		

The following is the eUSCI_B0 setup for this example.



Example 14.39
Transmitting a register address and three bytes of data from an I2C master (part 1)



Example: Transmitting a Register Address and Three Bytes of Data From an I2C Master (Part 2)

1) In CCS, create a new C/C++ Empty Project (with main.c) titled:
C_I2C_Tx2_Master_Write_Reg_n_3Bytes.

2) Type in the following code in main.c after the statement to stop the watchdog timer.

```
#include <msp430.h>

int Data_Cnt = 0;
char Packet[] = {0x03, 0x33, 0x44, 0x55};

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

    //-- 1. Put eUSCI_B0 into software reset
    UCB0CTLW0 |= UCSWRST;       // UCSWRST=1 for eUSCI_B0 in SW reset

    //-- 2. Configure eUSCI_B0
    UCB0CTLW0 |= UCSSEL__SMCLK; // Choose BRCLK=SMCLK=1MHz
    UCB0BRW = 10;               // Divide BRCLK by 10 for SCL=100kHz

    UCB0CTLW0 |= UCMODE_3;      // Put into I2C mode
    UCB0CTLW0 |= UCMST;         // Put into master mode
    UCB0CTLW0 |= UCTR;          // Put into Tx mode
    UCB0I2CSA = 0x0068;         // Slave address = 0x68

    UCB0CTLW1 |= UCASTP_2;      // Auto STOP when UCB0TBCNT reached
    UCB0TBCNT = sizeof(Packet); // # of Bytes in Packet

    //-- 3. Configure Ports
    P1SEL1 &= ~BIT3;            // We want P1.3 = SCL
    P1SEL0 |= BIT3;

    P1SEL1 &= ~BIT2;            // We want P1.2 = SDA
    P1SEL0 |= BIT2;

    PM5CTL0 &= ~LOCKLPM5;       // Disable LPM

    //-- 4. Take eUSCI_B0 out of SW reset
    UCB0CTLW0 &= ~UCSWRST;      // UCSWRST=1 for eUSCI_B0 in SW reset

    //-- 5. Enable Interrupts
    UCB0IE |= UCTXIE0;          // Enable I2C Tx0 IRQ
    __enable_interrupt();       // Enable Maskable IRQs

    int i;
    while(1){
        UCB0CTLW0 |= UCTXSTT;   // Generate START condition
        for(i=0; i<100; i=i+1){} // delay loop
    }
    return 0;
}
//------- Interrupt Service Routines --------------------
#pragma vector=EUSCI_B0_VECTOR
__interrupt void EUSCI_B0_I2C_ISR(void){
    if (Data_Cnt == (sizeof(Packet) - 1)) {
        UCB0TXBUF = Packet[Data_Cnt];
        Data_Cnt = 0;
    } else {
        UCB0TXBUF = Packet[Data_Cnt];
        Data_Cnt++;
    }
}
```

We will create an array for the data to be sent. We'll put the register address as the first byte in the packet.

Set up SCL=100 kHz.

Setup eUSCI_B0 peripheral as an I2C master transmitting to a slave with address 0x68.

Setup auto STOP after all bytes sent.

Configure MCU ports as I2C.

Enable Tx0 IRQ to indicate when Tx buffer is ready.

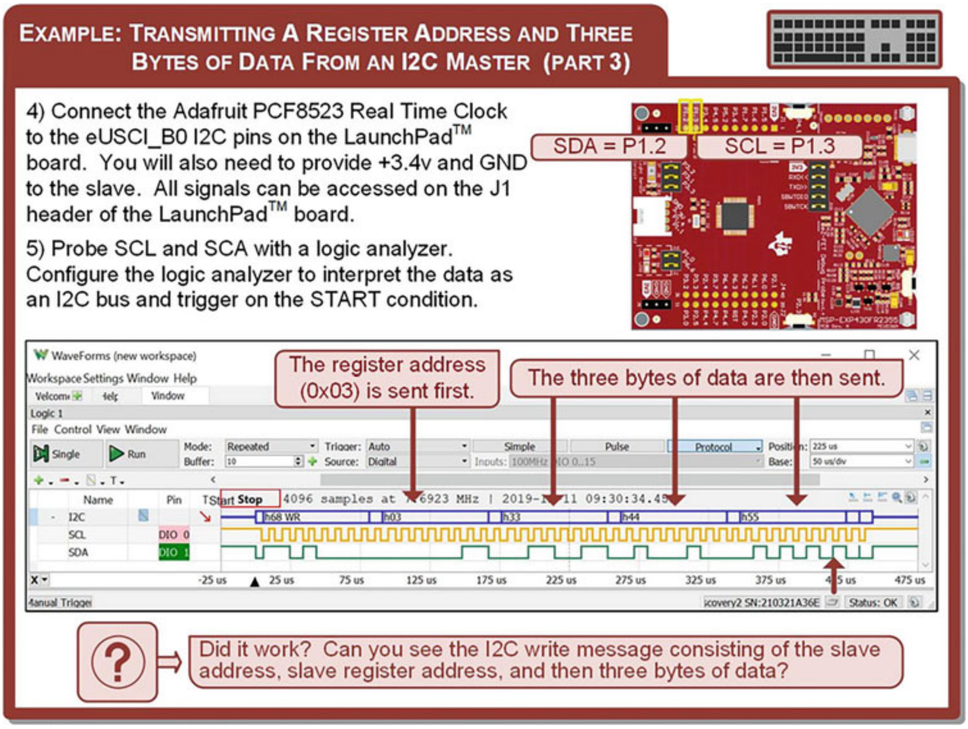
Continually generate START condition.

Check if we are at the last byte in the packet. If so, reset packet index pointer.

If not at the end, send packet and increment packet index pointer.

3) Save and debug your program. → We'll connect the logic analyzer next.

Example 14.40
Transmitting a register address and three bytes of data from an I2C master (part 2)



EXAMPLE: TRANSMITTING A REGISTER ADDRESS AND THREE BYTES OF DATA FROM AN I2C MASTER (PART 3)

4) Connect the Adafruit PCF8523 Real Time Clock to the eUSCI_B0 I2C pins on the LaunchPad™ board. You will also need to provide +3.4v and GND to the slave. All signals can be accessed on the J1 header of the LaunchPad™ board.

5) Probe SCL and SCA with a logic analyzer. Configure the logic analyzer to interpret the data as an I2C bus and trigger on the START condition.

The register address (0x03) is sent first.

The three bytes of data are then sent.

Did it work? Can you see the I2C write message consisting of the slave address, slave register address, and then three bytes of data?

Example 14.41
Transmitting a register address and three bytes of data from an I2C master (part 3)

14.3.2.2 Reading Data as an I2C Master

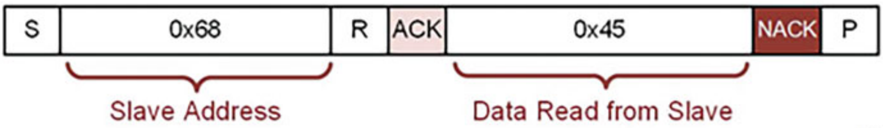
When reading from a slave, we follow many of the initialization steps as when transmitting. We still configure the speed of SCL using UCB0BRW, put the peripheral into I2C mode using UCMODEx, and make it a master using UCMST. We still put the slave address to communicate with into UCB0I2CA and indicate the number of bytes that will be transferred before an automatic STOP condition using UC0TBCNT. The only difference during eUSCI_B0 initialization for reading is to put the peripheral into receive mode using UCTR = 0.

When reading, the UCRXIFG0 will be asserted when the slave sends back a byte of data and it arrives in the Rx buffer. This flag will trigger an eUSCI_B0 interrupt. Within the eUSCI_B0 service routine, we will simply store the Rx buffer value into a variable. During read mode, the master will produce a NACK signal after the last byte in the transmission has been received as dictated by the value in UC0TBCNT.

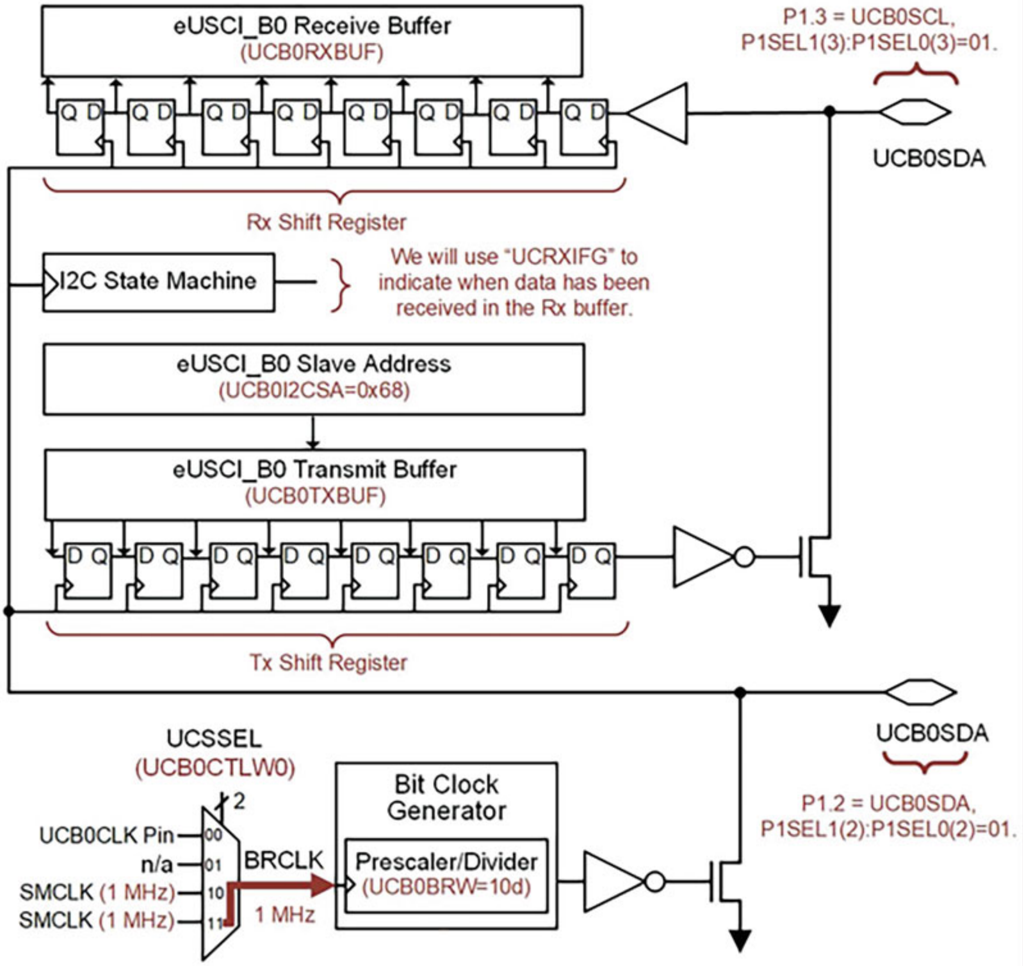
Let's look at a program that will continually read one byte of data from a slave with a slave address of 0 × 68. In this program we will continually generate a START condition by setting UCTXSTT within the main loop. We will use an Rx interrupt service routine to read the value received in the Rx buffer. Follow the design in Examples 14.42, 14.43, and 14.44 to see how to use the I2C peripheral in master receive mode to read a single byte of data from a slave.

Example: Receiving One Byte From an I2C Slave (Part 1)

Let's configure the eUSCI_B0 peripheral as an I2C master and continually read a single byte from a slave. The transmitting slave device in this example is the *Adafruit PCF8523 Real Time Clock* with a hard-coded slave address of **0x68**. This is the I2C message we are trying to create:



Note that to run this example, a slave device needs to be connected to the LaunchPad™ board to provide the pull-up resistors and the ACK signals. After the master generates the START condition and sends the slave address, the slave will send an ACK and then send back its data. The master will put out a NACK to indicate to the slave not to send any more data. The master then generates the STOP condition. The following is the eUSCI_B0 setup for this example.

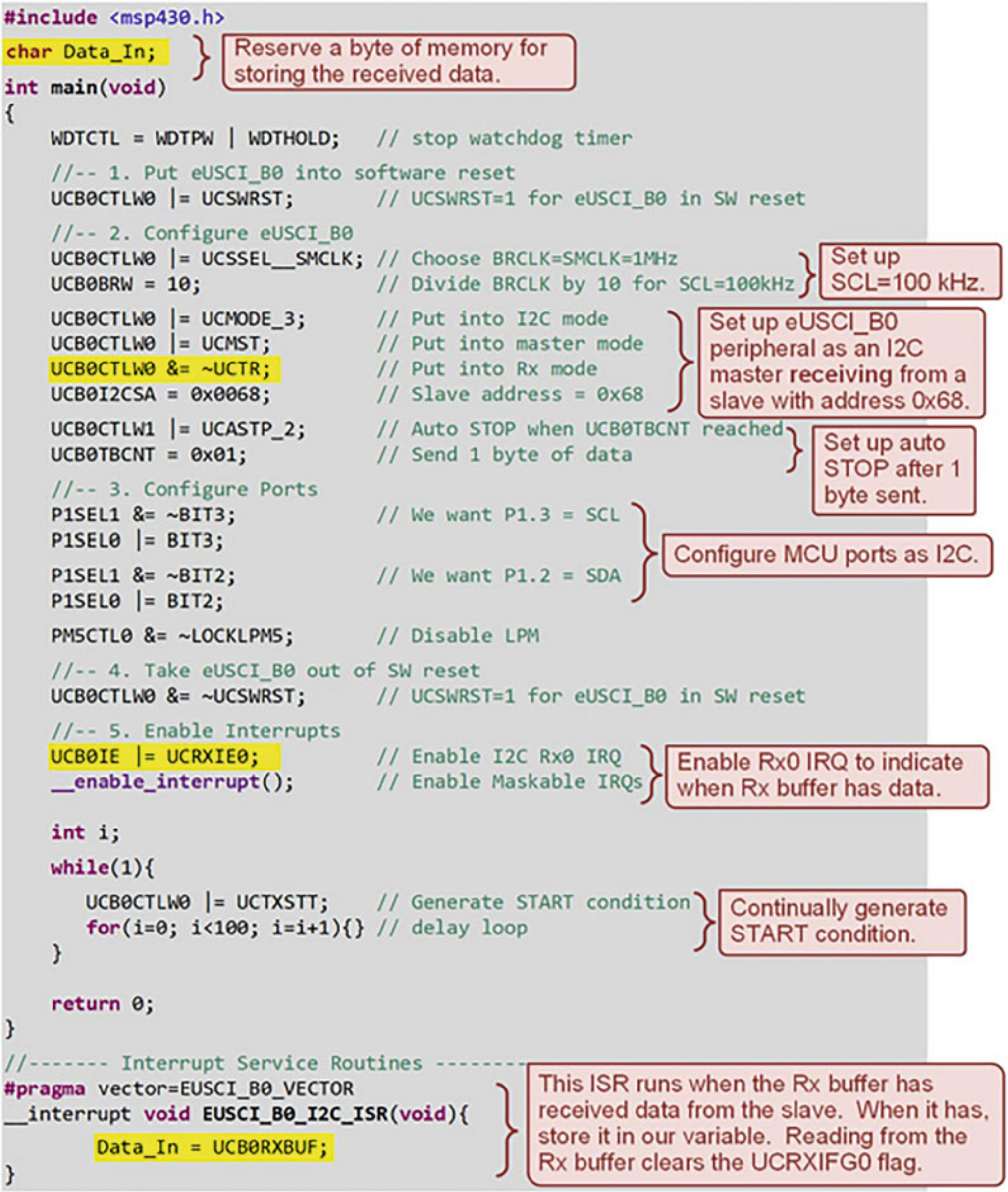


Example 14.42
Receiving one byte from an I2C slave (part 1)

Example: Receiving One Byte From an I2C Slave (Part 2)

1) In CCS, create a new C/C++ Empty Project (with main.c) titled:
C_I2C_Rx1_Master_Write_1Byte.

2) Type in the following code in main.c after the statement to stop the watchdog timer.



```
#include <msp430.h>
char Data_In;
int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;     // stop watchdog timer
    //-- 1. Put eUSCI_B0 into software reset
    UCB0CTLW0 |= UCSWRST;         // UCSWRST=1 for eUSCI_B0 in SW reset
    //-- 2. Configure eUSCI_B0
    UCB0CTLW0 |= UCSSEL__SMCLK;   // Choose BRCLK=SMCLK=1MHz
    UCB0BRW = 10;                 // Divide BRCLK by 10 for SCL=100kHz

    UCB0CTLW0 |= UCMODE_3;        // Put into I2C mode
    UCB0CTLW0 |= UCMST;           // Put into master mode
    UCB0CTLW0 &= ~UCTR;           // Put into Rx mode
    UCB0I2CSA = 0x0068;           // Slave address = 0x68

    UCB0CTLW1 |= UCASTP_2;        // Auto STOP when UCB0TBCNT reached
    UCB0TBCNT = 0x01;             // Send 1 byte of data
    //-- 3. Configure Ports
    P1SEL1 &= ~BIT3;              // We want P1.3 = SCL
    P1SEL0 |= BIT3;

    P1SEL1 &= ~BIT2;              // We want P1.2 = SDA
    P1SEL0 |= BIT2;

    PM5CTL0 &= ~LOCKLPM5;         // Disable LPM
    //-- 4. Take eUSCI_B0 out of SW reset
    UCB0CTLW0 &= ~UCSWRST;        // UCSWRST=1 for eUSCI_B0 in SW reset
    //-- 5. Enable Interrupts
    UCB0IE |= UCRXIE0;            // Enable I2C Rx0 IRQ
    __enable_interrupt();         // Enable Maskable IRQs

    int i;
    while(1){
        UCB0CTLW0 |= UCTXSTT;     // Generate START condition
        for(i=0; i<100; i=i+1){}  // delay loop
    }

    return 0;
}
//------- Interrupt Service Routines --------
#pragma vector=EUSCI_B0_VECTOR
__interrupt void EUSCI_B0_I2C_ISR(void){
        Data_In = UCB0RXBUF;
}
```

Reserve a byte of memory for storing the received data.

Set up SCL=100 kHz.

Set up eUSCI_B0 peripheral as an I2C master receiving from a slave with address 0x68.

Set up auto STOP after 1 byte sent.

Configure MCU ports as I2C.

Enable Rx0 IRQ to indicate when Rx buffer has data.

Continually generate START condition.

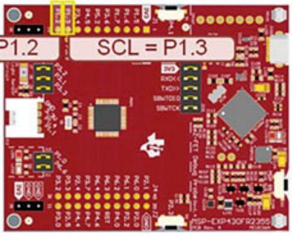
This ISR runs when the Rx buffer has received data from the slave. When it has, store it in our variable. Reading from the Rx buffer clears the UCRXIFG0 flag.

3) Save and debug your program. → We'll connect the logic analyzer next.

Example 14.43
Receiving one byte from an I2C slave (part 2)

EXAMPLE: RECEIVING ONE BYTE FROM AN I2C SLAVE (PART 3)

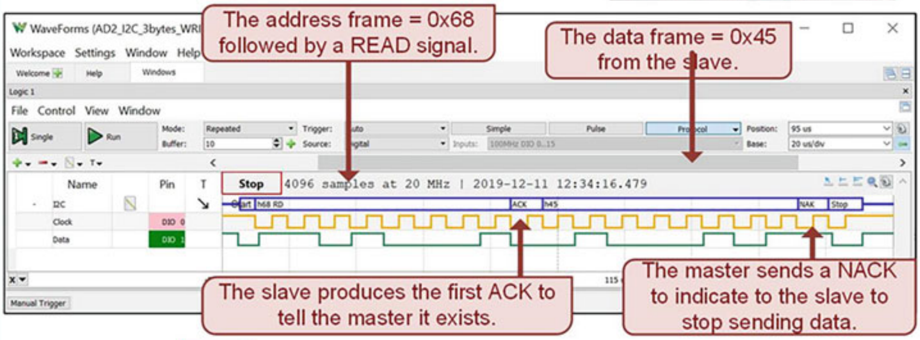
4) Connect the Adafruit PCF8523 Real Time Clock to the eUSCI_B0 I2C pins on the LaunchPad^{TM} board. You will also need to provide +3.4v and GND to the slave. All signals can be accessed on the J1 header of the LaunchPad^{TM} board.



SDA = P1.2

SCL = P1.3

5) Probe SCL and SCA with a logic analyzer. Configure the logic analyzer to interpret the data as an I2C bus and trigger on the START condition.



The address frame = 0x68 followed by a READ signal.

The data frame = 0x45 from the slave.

The slave produces the first ACK to tell the master it exists.

The master sends a NACK to indicate to the slave to stop sending data.

Did it work? Can you see the I2C read message?

Example 14.44
Receiving one byte from an I2C slave (part 3)

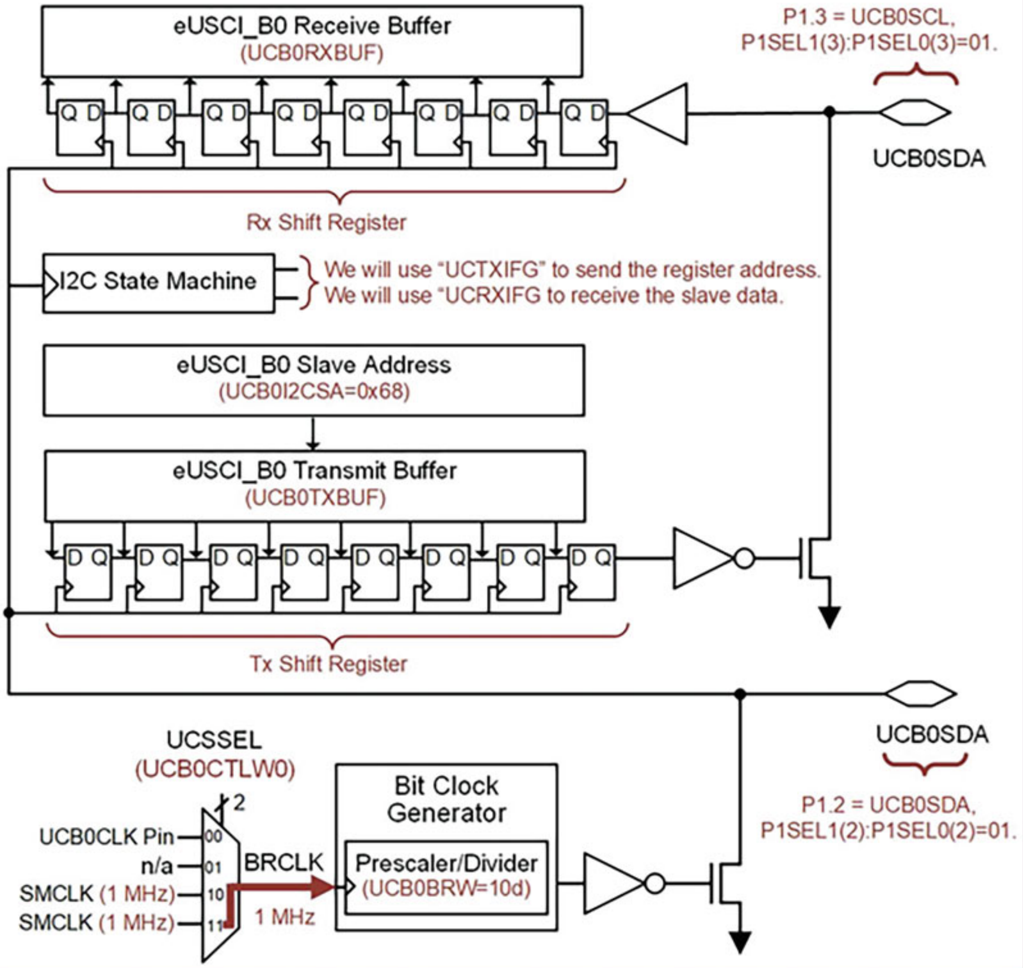
Now let’s look at an example of reading from a specific register address within the slave device. Recall that this is accomplished by sending two messages; the first is a write message that provides the register address in the data frame; the second is a read message that transfers the data in the provided register address from the slave to the master. There are a variety of ways to accomplish this design. The example that will be presented generates the start conditions for the two messages in the main while() loop and then allows the eUSCI_B0 interrupt service routine to handle transmitting the register address during the write message and receiving the data during the read message. When using this approach, there needs to be functionality in-between the two start conditions that waits for the prior message to complete before the next message can be sent. This is accomplished by polling the STOP flag (UCSTPIFG) in the UCB0IFG register. This flag will assert once the stop bit has been generated for a complete message. Follow the design in Examples 14.45, 14.46, and 14.47 to experiment with using two messages to read from a specific register address within a slave.

Example: Reading From a Specific Slave Register Address Using Two I2C Messages (part 1)

Let's configure the eUSCI_B0 peripheral as an I2C master and continually read from register address 0x03 in the slave. This will require two messages. The first will provide the register address using a write message. The second will read the value at the register address provided using a read message. The slave device in this example is the *Adafruit PCF8523 Real Time Clock* with a hard-coded slave address of **0x68**. This is the I2C message we are trying to create:

S	0x68	W	ACK	0x03	ACK	P	S	0x68	R	ACK	0xD0	NACK	P
	Slave Address	Write		Register Address				Slave Address	Read		Register Data		

The following is the eUSCI_B0 setup for this example.

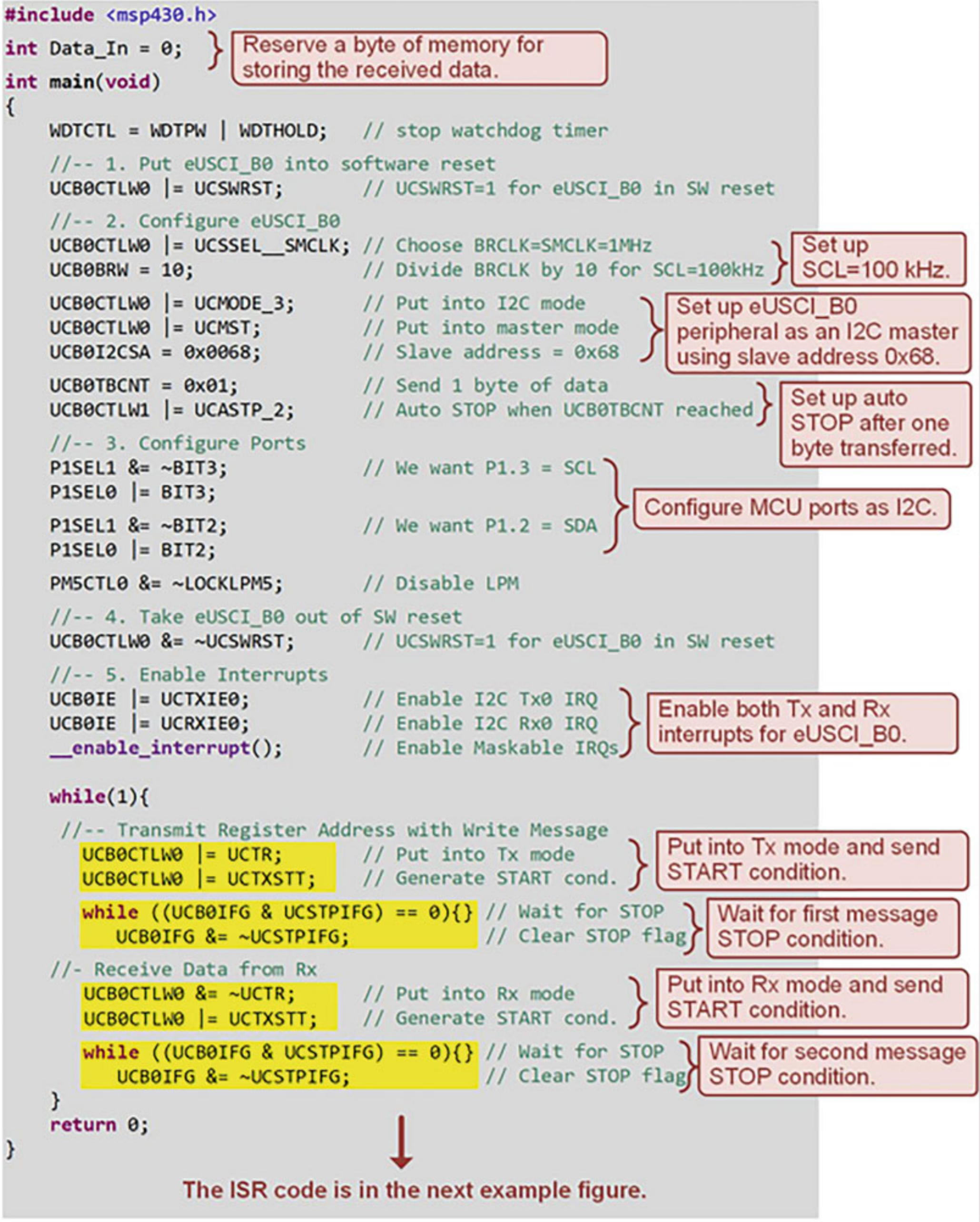


Example 14.45
Reading from a specific slave register address using two I2C messages (part 1)

Example: Reading From a Specific Slave Register Address Using Two I2C Messages (part 2)

1) In CCS, create a new C/C++ Empty Project (with main.c) titled:
C_I2C_Rx2_Master_Read_From_Reg_Addr.

2) Type in the following code in main.c after the statement to stop the watchdog timer.



```
#include <msp430.h>
int Data_In = 0;
int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;    // stop watchdog timer
    //-- 1. Put eUSCI_B0 into software reset
    UCB0CTLW0 |= UCSWRST;        // UCSWRST=1 for eUSCI_B0 in SW reset
    //-- 2. Configure eUSCI_B0
    UCB0CTLW0 |= UCSSEL__SMCLK; // Choose BRCLK=SMCLK=1MHz
    UCB0BRW = 10;                // Divide BRCLK by 10 for SCL=100kHz
    UCB0CTLW0 |= UCMODE_3;       // Put into I2C mode
    UCB0CTLW0 |= UCMST;          // Put into master mode
    UCB0I2CSA = 0x0068;          // Slave address = 0x68
    UCB0TBCNT = 0x01;            // Send 1 byte of data
    UCB0CTLW1 |= UCASTP_2;       // Auto STOP when UCB0TBCNT reached
    //-- 3. Configure Ports
    P1SEL1 &= ~BIT3;             // We want P1.3 = SCL
    P1SEL0 |= BIT3;
    P1SEL1 &= ~BIT2;             // We want P1.2 = SDA
    P1SEL0 |= BIT2;
    PM5CTL0 &= ~LOCKLPM5;        // Disable LPM
    //-- 4. Take eUSCI_B0 out of SW reset
    UCB0CTLW0 &= ~UCSWRST;       // UCSWRST=1 for eUSCI_B0 in SW reset
    //-- 5. Enable Interrupts
    UCB0IE |= UCTXIE0;           // Enable I2C Tx0 IRQ
    UCB0IE |= UCRXIE0;           // Enable I2C Rx0 IRQ
    __enable_interrupt();        // Enable Maskable IRQs

    while(1){
     //-- Transmit Register Address with Write Message
       UCB0CTLW0 |= UCTR;        // Put into Tx mode
       UCB0CTLW0 |= UCTXSTT;     // Generate START cond.
       while ((UCB0IFG & UCSTPIFG) == 0){} // Wait for STOP
          UCB0IFG &= ~UCSTPIFG;            // Clear STOP flag
    //- Receive Data from Rx
       UCB0CTLW0 &= ~UCTR;       // Put into Rx mode
       UCB0CTLW0 |= UCTXSTT;     // Generate START cond.
       while ((UCB0IFG & UCSTPIFG) == 0){} // Wait for STOP
          UCB0IFG &= ~UCSTPIFG;            // Clear STOP flag
    }
    return 0;
}
```

Reserve a byte of memory for storing the received data.
Set up SCL=100 kHz.
Set up eUSCI_B0 peripheral as an I2C master using slave address 0x68.
Set up auto STOP after one byte transferred.
Configure MCU ports as I2C.
Enable both Tx and Rx interrupts for eUSCI_B0.
Put into Tx mode and send START condition.
Wait for first message STOP condition.
Put into Rx mode and send START condition.
Wait for second message STOP condition.
The ISR code is in the next example figure.

Example 14.46
Reading from a specific slave register address using two I2C messages (part 2)

Example: Reading From a Specific Slave Register Address Using Two I2C Messages (part 3)

```
//------- Interrupt Service Routines ---------------------
#pragma vector=EUSCI_B0_VECTOR
__interrupt void EUSCI_B0_I2C_ISR(void){

  switch(UCB0IV){
    case 0x16:                      // ID 16: RXIFG0
        Data_In = UCB0RXBUF;        // Retrieve data
        break;

      case 0x18:                    // ID 18: TXIFG0
        UCB0TXBUF = 0x03;           // Send Reg Addr
        break;

      default:
        break;
  }
}
```

We will use UCB0IV to determine which IRQ flag has triggered.

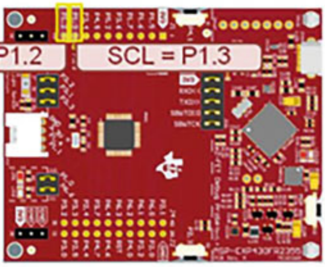
The Rx flag asserts during the second message when data comes back from the slave.

The Tx flag asserts during the first message when the register address can be sent.

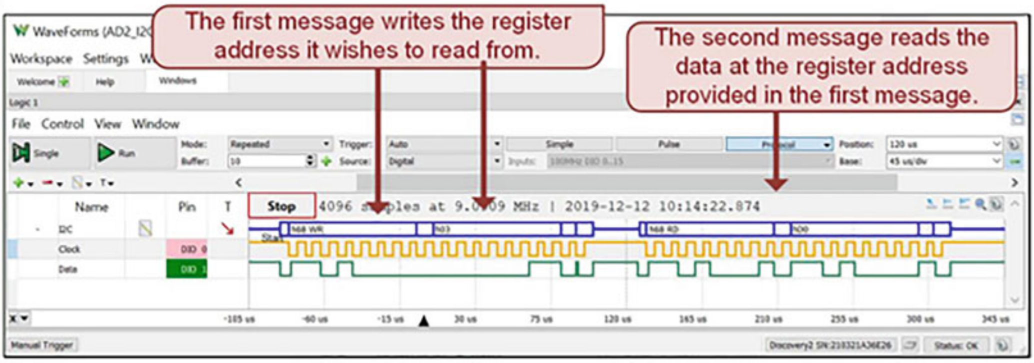
3) Save and debug your program.

4) Connect the Adafruit PCF8523 Real Time Clock to the eUSCI_B0 I2C pins on the LaunchPad™ board. You will also need to provide +3.4v and GND to the slave. All signals can be accessed on the J1 header of the LaunchPad™ board.

5) Probe SCL and SCA with a logic analyzer. Configure the logic analyzer to interpret the data as an I2C bus and trigger on the slave address of 0x68.


SDA = P1.2

SCL = P1.3


The first message writes the register address it wishes to read from.

The second message reads the data at the register address provided in the first message.

Mode: Repeated
Buffer: 10
Trigger: Auto
Source: Digital
Position: 120 us
Base: 45 us/div

4096 samples at 9.0909 MHz | 2019-12-12 10:14:22.874

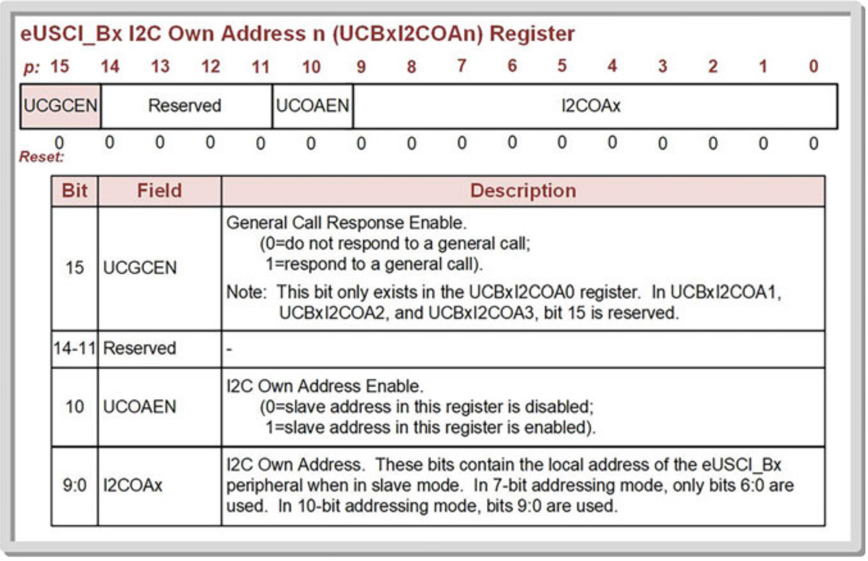
?

Did it work? Can you generate two I2C messages? Register address 0x03 on the PCF8523 is the "seconds" field. If you run the logic analyzer continuously you will see the value increment every second.

Example 14.47
Reading from a specific slave register address using two I2C messages (part 3)

14.3.3 I2C Slave Operation on the MSP430FR2355

While most of the time an MCU acts as the I2C master, the MSP430FR2355 also can be configured as an I2C slave. This allows other I2C masters to use some of the capabilities on the MCU such as its timers, its ADC, or any of its other peripherals. Configuring the MSP430FR2355 as a slave is accomplished using UCMODx = 11 to put the peripheral into I2C mode and UCMST = 0 to configure it as a slave. In slave mode, the MCU does not produce SCL, so no configuration of UCSSELx or UCBRx is needed. The MCU is first put into receiver mode using UCTR = 0 in order to receive the I2C slave address that the master sends out to all slaves. The MSP430FR2355 supports up to four separate and user-programmable slave addresses. Each of these slave addresses contains independent interrupts flags for both Tx and Rx. The slave address values are stored in the *eUSCI_Bx I2C Own Address n* (UCBxI2COAn) registers by the user during initialization. The four specific register names in the MSP430FR2355 are UCBxI2CA3, UCBxI2CA2, UCBxI2CA1, and UCBxI2CA0. Each of these registers contains an *Own Address Enable* (UCOAEN) bit in position 10 that must be asserted if the slave address register is to be active. The UCBxI2CA0 register is unique in that its 15th position is the *General Call Response Enable* (ECGEN) for the MCU's entire slave system. The 15th positions in the other three I2C own address registers are reserved. Figure 14.48 gives the details of the UCBxI2COAn registers.



eUSCI_Bx I2C Own Address n (UCBxI2COAn) Register

p:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	UCGCEN	Reserved				UCOAEN	I2COAx									
Reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	Field	Description
15	UCGCEN	General Call Response Enable. (0=do not respond to a general call; 1=respond to a general call). Note: This bit only exists in the UCBxI2COA0 register. In UCBxI2COA1, UCBxI2COA2, and UCBxI2COA3, bit 15 is reserved.
14-11	Reserved	-
10	UCOAEN	I2C Own Address Enable. (0=slave address in this register is disabled; 1=slave address in this register is enabled).
9:0	I2COAx	I2C Own Address. These bits contain the local address of the eUSCI_Bx peripheral when in slave mode. In 7-bit addressing mode, only bits 6:0 are used. In 10-bit addressing mode, bits 9:0 are used.

Fig. 14.48
eUSCI_Bx I2C own address n (UCBxI2COAn) register

In slave mode, if the I2C slave address sent by a master matches any of the values in an enabled UCBxI2COAn register, an ACK will be automatically sent and the UCSTTIFG flag is set. Whether the MCU is supposed to transmit or receive is configured automatically by setting the corresponding UCTXIFG or UCRXIFG flag in the slave (i.e., there is no need to manually configure UCTR). The user simply needs to read or write to the Tx or Rx buffers within the eUSCI_B service routine depending on the type of data transfer that is being requested.

For a slave transmit, data is put into the Tx buffer and automatically shifted out. The master will send back an ACK if it wants to receive another byte. This ACK will trigger another transmit interrupt so that the MCU can send more data. If the master is done receiving data, it will send a NACK followed by the STOP condition to end the message.

For a slave receive, the master will shift data into the Rx shift register. When complete, the data will be transferred to the Rx buffer and the UCRXIFG is asserted indicating that it is ready to be transferred into an internal variable. The slave will then send an ACK signal back to the master.

Concept Check

CC14.3 What is the most common mistake that beginners make when implementing an I2C bus for the first time that prevents the bus from working?

A) They forget to put the pull-up resistors on SCL and SDA.

B) The answer is "A".

Summary

- ❖ Serial communication links allow data to be transmitted as a series of bits across a single line. This reduces the number of pins needed compared to parallel communication, which sends each bit on its own line.
- ❖ The MSP430FR2355 supports three built-in serial communication standards: UART, SPI, and I2C.
- ❖ The MSP430FR2355 contains four serial communication peripherals called eUSCI_A0, eUSCI_A1, eUSCI_B0, and eUSCI_B1.
- ❖ The MSP430FR2355 eUSCI_Ax peripherals can be configured to operate in UART or SPI mode.
- ❖ The MSP430FR2355 eUSCI_Bx peripherals can be configured to operate in I2C or SPI mode.
- ❖ A UART interface sends data in an asynchronous manner. This reduces the number of signals needed in the link by not passing a clock between the transmitter and receiver.
- ❖ Before a UART begins transmitting data, the Tx and Rx are configured to have the same data frame characteristics and the baud rate.
- ❖ The baud rate describes the number of times per second that the line can change states. On the MSP430, the only two states supported are HIGH and LOW. This means the baud rate is also the number bits that can be sent per second (i.e., the bit rate).
- ❖ When two devices are connected for serial communication, it is called a *link*.
- ❖ A simplex link contains one wire and only supports serial communication in one direction.
- ❖ A full-duplex link contains two dedicated wires: one for transmitting from device A to B and one for transmitting from device B to A.
- ❖ A half-duplex link shares a single line but supports bidirectional communication. In this type of link, the two devices must share the line. Half-duplex links require an arbitration scheme to avoid having two devices transmitting at the same time.
- ❖ UART framing is the term that describes the format of the serial bit sequence. UART frames are typically 8 bits long with one start bit and one stop bit and send the LSB first. Other UART frame options are available such as 7-bit packets, sending the MSB first, adding a second stop bit, adding an error checking parity bit, and adding an address bit.
- ❖ The UART transmitter and receiver use shift registers to send and receive the serial information. When transmitting, data is written into a Tx buffer. The Tx buffer is then transferred into the Tx shift register and shifted out. When receiving, data is shifted into the Rx shift register. It is then transferred into a Rx buffer.
- ❖ A UART receiver recovers the information sent by the receiver by oversampling the incoming data. The sampling sequence begins when the receiver sees the start bit. It then counts the number of samples until it reaches a point where it knows the entire

frame has been received. The most common oversampling ratio is 16×. The Tx shift register is clocked at the link's baud rate. The Rx shift register is clocked at the oversampling ratio (i.e., 16 × BR).

- UART is a communication scheme. When logic levels are applied to the logic states, the scheme becomes a standard. Standards allow separate devices to design circuitry and programs that will interface with other devices using the same standard. Two common UART standards are TTL and RS-232.
- The MSP430FR2355 has circuitry to generate a clock to produce the desired baud rate.
- The source of the clock system can be chosen between an internal pin, ACLK, and SMCLK.
- The baud rate generator circuit contains a prescaler block followed by a modulator block. The prescaler block divides the UART source clock to get close to the desired baud rate. The modulator blocks then apply incremental adjustments to the clock divider ratio in order to compensate for any additional error.
- The MSP430FR2355 contains a variety of configuration registers to set up the UART. When setting up the UART, the recommended sequences of steps is: (1) put the system into software reset using UCSWRST; (2) initialize the eUSCI registers; (3) configure the ports; (4) take the system out software reset using UCSWRST; and (5) enable interrupts (optional).
- When using the eUSCI peripherals, the PxSEL1:PxSEL0 registers are used to select the serial communication peripheral for the pin's function.
- The MSP430FR2355 provides interrupts to indicate when data has been transmitted or received.
- A *terminal* window provides a way to transmit and receive standard I/O through the computer's serial port. By default, data is encoded in ASCII when using a terminal. ASCII assigns a unique 8-bit code to every symbol in the American written language. The data type “char” will automatically store the ASCII code for the symbol provided.
- The SPI protocol provides a way to transmit serial data in a synchronous manner.
- The SPI protocol uses the concept of a master and a slave where the master provides the clock for every device in the system.
- The SPI serial clock is called SCLK. By default, data is transmitted by the master on the rising edge of SCLK and latched by the slave(s) on the falling edge.
- The SPI master data out is denoted SIMO for “slave in, master out.” Sometimes this is also referred to as MISO, for “master out, slave in.”
- The SPI master data in is denoted SOMI for “slave out, master in.” Sometimes this is also referred to as MOSI, for “master in, slave out.”
- In SPI three-wire mode, a single master and single slave communicate using SCLK, SIMO, and SOMI.
- The SPI system can also use a slave transmit enable (STE) line in order to activate individual slaves for communication. Sometimes this is also referred to as slave select (SS).
- When a SPI system uses an STE line, it is called *four-wire mode*.
- The SPI system uses shift registers on the Tx and Rx to transfer data. A common SCLK provided by the master will clock all shift registers in the system.
- Multiple SPI slaves can be configured in as a *bus*, in which the SIMO and SOMI lines are shared among all devices and each slave it enabled with its own STE.
- Multiple SPI slaves can also be configured in a *daisy-chain* configuration in which a continuous shift register loop is created among the master and all slaves. Each time an SCLK pulses, data is shifted around the loop. The master must count the number of SCLKs that have occurred in order to track where in the loop the current data resides.
- The MSP430FR2355 SPI system generates SCLK using a clock generation circuit. The source of the clock system can be chosen between an internal pin, ACLK, and SMCLK. The clock generation circuit also contains a programmable prescaler that divides down the source clock to provide slower bit rates.
- For the master to receive data, it must generate the SCLKs to shift data into its receive buffer. This is accomplished by writing data (dummy or real) into its own Tx buffer. Writing to the master Tx buffer initiates a transfer and generates the SCLK for all devices.
- For a slave to transmit data, it stores information into its Tx buffer and then must wait for

the master to provide pulses on SCLK to shift the data out.

- The SPI system provides interrupts to indicate if the data has been shifted out (TXIFG) or if new data has arrived (RXIFG).
- I2C is a serial interface that is implemented with a two-wire, half-duplex bus. I2C supports multiple masters and multiple slaves.
- I2C provides synchronous communication over a two-wire bus. The two signals are SCL (serial clock) and SDA (serial data).
- The MSP430FR2355 eUSCI_Bx peripherals can be configured in either SPI or I2C mode. This means the MCU supports up to two I2C systems.
- The I2C bus is driven with open-drain transmitters. An open-drain transmitter can pull the signal line LOW but cannot drive it HIGH. To produce a logic HIGH on the signal, a pull-up resistor to V_{CC} is needed on both SCL and SDA.
- If no devices are driving the bus, it is *idle* and the pull-up resistors pull both SCL and SDA HIGH.
- At any given time, only one master can be controlling the bus. When the bus is being used, it is *busy*.
- Each slave on the bus has a unique slave address that the master uses to request communication.
- I2C transmits information in messages. A message is contained between START and STOP conditions, each generated by the master. The master produces SCL for the message.
- The first frame within an I2C message is the slave address followed by a read/write bit. If any slave on the bus has the address that the master is sending, it will send back an ACK signal indicating that it exists and is ready.
- Each bit in the message is given a period count value for documentation purposes. If a slave sends an ACK to the slave address, it occurs in period 9.
- If a slave sends an ACK in period 9, the next frame in the message is data. If the transaction is a write, data is transmitted by the master to the slave. If the transaction is a read, data is received by the master from the slave.
- When in transmit mode, the slave sends an ACK after each data byte is received from the master. The master can send multiple bytes of data to the slave in one message.
- When receiving, the master sends an ACK after each data byte is received with the exception of the last data byte. On the last data byte transferred in the message, the master sends a NACK to tell the slave to stop transferring data. The master then generates the STOP condition.
- I2C devices typically have multiple registers that hold various information. In these devices, continually reading from the slave will result in the device incrementing through its register addresses automatically to sequentially output all of its register data.
- A master can also access a specific slave register address. For a write transaction, the master will generate the START condition, send the slave address, wait for the slave ACK, and then send the register address followed by the data to write.
- For an I2C read transaction from a specific slave register address, two messages are needed. In the two-message approach, the first message is a write transaction, and the master sends the register address it wishes to access in the data field. The second message is a read, and the specified register address sends back its information.
- Accessing a specific register can also be done using a single message and a repeated start condition. The message begins with a write transaction and provides the register address. But instead of generating the STOP condition, the master generates a second START condition and begins a new read transaction. This approach avoids a second master from taking over the bus during the communication, which might occur when using the two-message approach.

Exercise Problems

Section 14.1: Universal Asynchronous Receiver/Transmitter (UART)

14.1.1 What is an advantage of using the UART interface for serial communication?

14.1.2 What is a disadvantage of using the UART interface for serial communication?

14.1.3 Choose the baud rate in the following list that is not commonly used in UART communication: 9600, 38,400, 115,200, 1 M.

14.1.4 What value would you put into UCSSEL if you wanted to select ACLK as the BRCLK?

14.1.5 What value would you put into UCSSEL if you wanted to select SMCLK as the BRCLK?

14.1.6 You are setting up a baud rate of 57,600 on the MSP430FR2355 with a BRCLK=SMCLK. What value do you need to load into UCAxBRW to set the prescaler?

14.1.7 You are setting up a baud rate of 57,600 on the MSP430FR2355 with a BRCLK=SMCLK. What value do you need to load into UCAxMCTL to set the second stage modulation?

14.1.8 You receive a UART bitstream of 0b01000010 its LSB arriving first. After putting the bitstream back into its MSB first order, what the ASCII character that was received?

14.1.9 You receive a UART bitstream of 0b01000110 its LSB arriving first. After putting the bitstream back into its MSB first order, what the ASCII character that was received?

14.1.10 Why is an Rx interrupt critical for UART operation?

Section 14.2: Serial Peripheral Interface (SPI)

14.2.1 What is an advantage of using the SPI protocol for serial communication?

14.2.2 What is a disadvantage of using the SPI protocol for serial communication?

14.2.3 What is the SPI mode called when only a single master and single slave are connected and an STE signal is not used?

14.2.4 What is the SPI mode called when the STE signal is used as a slave enable?

14.2.5 What is the SPI link configuration called when the SIMO and SOMI lines are shared across all slaves and each slave has its own STE enable line?

14.2.6 What is the SPI link configuration called when the SIMO and SOMI lines are connected to form a continuous data loop that the serial information is shifted around?

14.2.7 What is the fastest SCLK that can be generated on the MSP430FR2355 LaunchPad™ board?

14.2.8 What value would you put into UCAxBRW if you wanted to generate an SCLK with a frequency of 500 kHz with a BRCLK=SMCLK?

14.2.9 What value would you put into UCAxBRW if you wanted to generate an SCLK with a frequency of 25 kHz with a BRCLK=SMCLK?

14.2.10 If the master is going to send out a 16-byte packet of information with the default framing characteristics, how many SCLK pulses will it need to generate to transfer this amount of data?

Section 14.3: Inter-integrated Circuit (I2C) Bus

14.3.1 What external circuitry needs to be connected to an I2C bus due to the nature of its open-drain transmitter architecture?

14.3.2 What logic level is the open-drain transmitter able to drive an I2C signal to?

14.3.3 What is the first thing that is sent over the I2C bus after the master generates the START condition?

14.3.4 What signal is sent in period 9 if a slave on the I2C bus has the same slave address as was sent by the master in periods 1→7?

14.3.5 What signal is sent in period 9 if there isn't a slave on the I2C bus that has the same slave address as was sent by the master in periods 1→7?

14.3.6 What field is sent in period 8 of an I2C message?

14.3.7 If the master is sending data to a slave, what does the slave send back after each byte is successfully received?

14.3.8 If the master is sending data to a slave, what does the slave send back if the data was not successfully received or wasn't understood?

14.3.9 If a master is reading from a slave and wants to tell it to stop sending data, what does the master do?

14.3.10 How many separate slave addresses can the MSP430FR2355 support when configured in slave mode?

Chapter 15: Analog-to-Digital Converters

This chapter looks at the *analog to digital conversion* (ADC) capability of the MSP430 [1–3]. ADCs allow embedded computers to convert analog voltages into digital values so that they can be monitored in real-time and trigger responses. The MSP430FR2355 contains a 12-bit ADC core that can perform conversions on up to 16 user-selected inputs. This chapter gives an overview of the ADC peripheral on the MSP430FR2355 and how to design a program in C to use it.

Learning Outcomes—After completing this chapter, you will be able to:

15.1 Describe the basic concept of operation of an analog-to-digital converter.
15.2 Design programs that use the MSP430FR2355's ADC peripheral in C.

15.1 Analog-to-Digital Converters

An analog-to-digital converter (ADC or A2D) is a circuit that takes in an analog voltage and produces a digital representation of its value. ADCs consist of two stages, a *sample-and-hold* stage and a *conversion* stage. When the sample-and-hold stage is activated, it makes momentary contact with the input signal and allows it to charge a capacitor within the sample circuit. The goal of this momentary contact is to charge the capacitor to the same voltage as the input. The sample-and-hold circuitry is designed so that this can be accomplished very quickly so that it can disconnect from the input signal as soon as possible to avoid altering its signal integrity. The action of duplicating the voltage value of the input signal is called a *sample*. A capacitor is able to hold this voltage for a brief amount of time, thus providing the *hold* functionality of the sample-and-hold circuit. The conversion stage then produces a digital value that represents the analog value held on the capacitor.

The conversion of the analog sample into a digital number is called *digitizing*, *quantizing*, or *discretizing* the value. All of these terms refer to how a continuous analog signal is converted into a set of discrete digital numbers. An ADC has an analog input range that it digitizes across. The range is provided to the ADC using two inputs, the *voltage reference high* (V_{R+}) and the *voltage reference low* (V_{R-}).

ADCs can be configured to sample periodically. The speed at which samples are collected is called the *sample rate* and typically has units of *kilo samples per second* (ksps) or *mega samples per second* (Msps). If the sample rate is sufficiently faster than the frequency of the input signal, then an accurate representation of the input signal can be reconstructed using the samples. The *Nyquist-Shannon sampling theorem* from communication theory states that if the sample rate is at least twice as fast as the frequency of the input signal, then the input signal can be reconstructed. Most ADCs run at sample rates that are much higher than the minimum rate dictated by the Nyquist-Shannon sampling theorem. Figure 15.1 shows the concept of operation of a basic ADC.

B. J. LaMeres, *Embedded Systems Design using the MSP430FR2355 LaunchPad™*,
https://doi.org/10.1007/978-3-031-20888-1_15

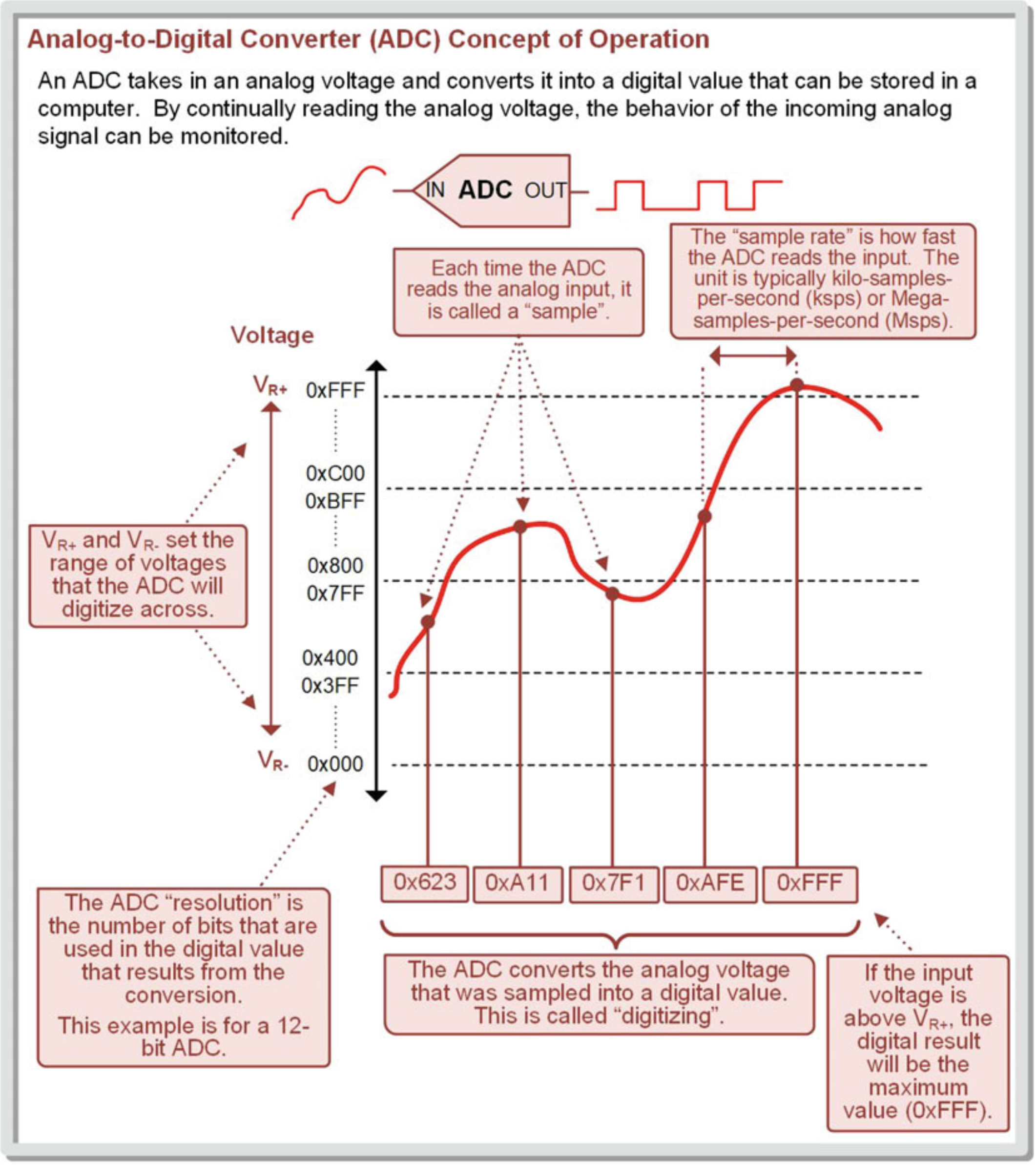

Analog-to-Digital Converter (ADC) Concept of Operation
An ADC takes in an analog voltage and converts it into a digital value that can be stored in a computer. By continually reading the analog voltage, the behavior of the incoming analog signal can be monitored.
IN
ADC
OUT
Each time the ADC reads the analog input, it is called a "sample".
The "sample rate" is how fast the ADC reads the input. The unit is typically kilo-samples-per-second (ksps) or Mega-samples-per-second (Msps).
Voltage
V_R+
0xFFF
0xC00
0xBFF
0x800
0x7FF
0x400
0x3FF
V_R-
0x000
V_R+ and V_R- set the range of voltages that the ADC will digitize across.
0x623
0xA11
0x7F1
0xAFE
0xFFF
The ADC "resolution" is the number of bits that are used in the digital value that results from the conversion.
This example is for a 12-bit ADC.
The ADC converts the analog voltage that was sampled into a digital value. This is called "digitizing".
If the input voltage is above V_R+, the digital result will be the maximum value (0xFFF).

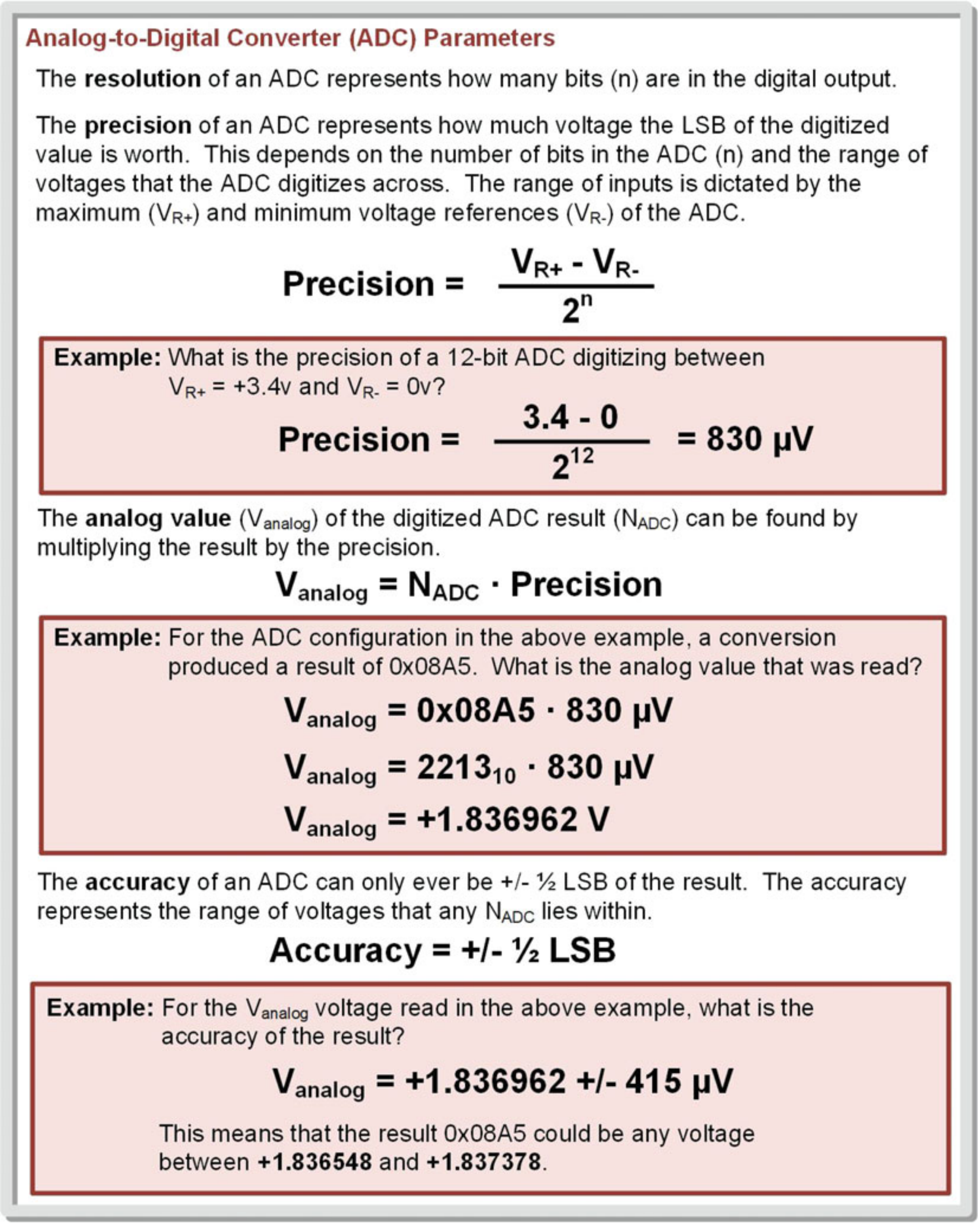
Fig. 15.1
Analog-to-digital converter (ADC) concept of operation

The *resolution* of an ADC refers to how many bits wide (n) the digital output value is. Common resolution values in MCUs are 8-bit, 10-bit, and 12-bit. The larger the resolution, the more accurate the conversion of the input signal is.

The *precision* of an ADC is the smallest voltage that the LSB of the digitized number can represent. The resolution is found by dividing the input voltage range by 2^n (i.e., $(V_{R+} - V_{R-})/2^n$).

The actual analog voltage (V_{analog}) that the ADC result represents can be found by multiplying the digital result (N_{ADC}) by the precision (i.e., $V_{analog} = N_{ADC} \cdot \text{precision}$).

The *accuracy* of an ADC states how close its digital output is to the actual input signal voltage. An ADC will only ever be able to get within ± ½ LSB of the original input signal due to the way that the input range is divided into discrete values that are 1 LSB apart from each other. The accuracy of a sample gives a range of voltages that the final digital output lies within (i.e., $V_{\text{analog}} = N_{\text{ADC}} \cdot \text{resolution} \pm \frac{1}{2}\,\text{LSB}$). Figure 15.2 gives a summary of the key ADC parameters.



Analog-to-Digital Converter (ADC) Parameters

The **resolution** of an ADC represents how many bits (n) are in the digital output.

The **precision** of an ADC represents how much voltage the LSB of the digitized value is worth. This depends on the number of bits in the ADC (n) and the range of voltages that the ADC digitizes across. The range of inputs is dictated by the maximum (V_{R+}) and minimum voltage references (V_{R-}) of the ADC.

$$\textbf{Precision} = \frac{V_{R+} - V_{R-}}{2^n}$$

Example: What is the precision of a 12-bit ADC digitizing between V_{R+} = +3.4v and V_{R-} = 0v?

$$\textbf{Precision} = \frac{3.4 - 0}{2^{12}} = 830\ \mu V$$

The **analog value** (V_{analog}) of the digitized ADC result (N_{ADC}) can be found by multiplying the result by the precision.

$$V_{\text{analog}} = N_{\text{ADC}} \cdot \textbf{Precision}$$

Example: For the ADC configuration in the above example, a conversion produced a result of 0x08A5. What is the analog value that was read?

$$V_{\text{analog}} = \text{0x08A5} \cdot 830\ \mu V$$

$$V_{\text{analog}} = 2213_{10} \cdot 830\ \mu V$$

$$V_{\text{analog}} = +1.836962\ V$$

The **accuracy** of an ADC can only ever be +/- ½ LSB of the result. The accuracy represents the range of voltages that any N_{ADC} lies within.

$$\textbf{Accuracy} = +/- \tfrac{1}{2}\ \textbf{LSB}$$

Example: For the V_{analog} voltage read in the above example, what is the accuracy of the result?

$$V_{\text{analog}} = +1.836962 +/- 415\ \mu V$$

This means that the result 0x08A5 could be any voltage between **+1.836548** and **+1.837378**.

Fig. 15.2
Analog-to-digital converter (ADC) parameters

CONCEPT CHECK

CC15.1 If precision is important, why not just make an ADC with a massive amount of resolution such as 256-bits?

A) The primary limit to the number of bits in an ADC is electrical noise. At some point, the neighboring digital codes represent analog values that are so close together that the electrical noise that exists in every system is much larger than the precision and the further digitizing is pointless.

15.2 ADC Operation on the MSP430FR2355

The MSP430FR2355 contains an ADC core with selectable resolution (8-bit, 10-bit, or 12-bit). The ADC can be driven with 1 of 16 inputs that are selected using an analog multiplexer that sits in front of the ADC core. The ADC clock source is selectable with two stages of programmable dividers/prescalers. The voltage range of the ADC is also programmable with options of using the power supply (VCC) and GND (VSS), input pins, or a variety of internally generated voltages. The ADC peripheral also has a large number of programmable options that dictate its behavior in addition to six interrupts that can be triggered upon certain conditions. Figure 15.3 shows a simplified block diagram of the ADC peripheral on the MSP430FR2355.

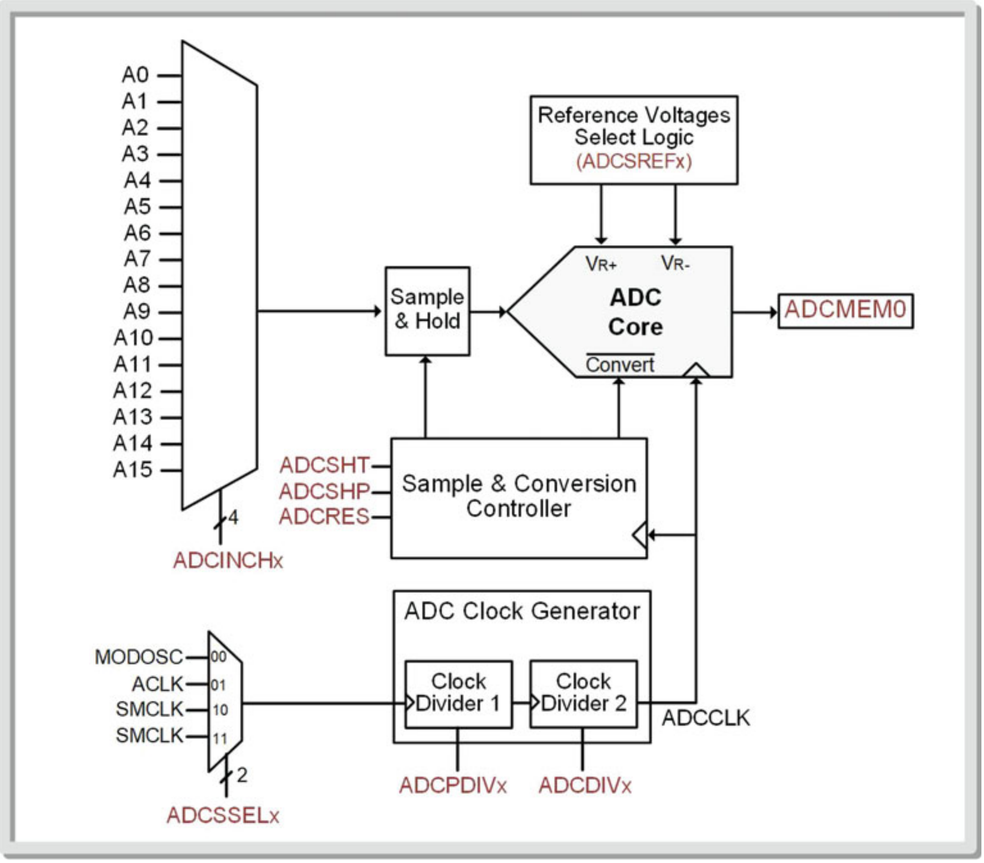
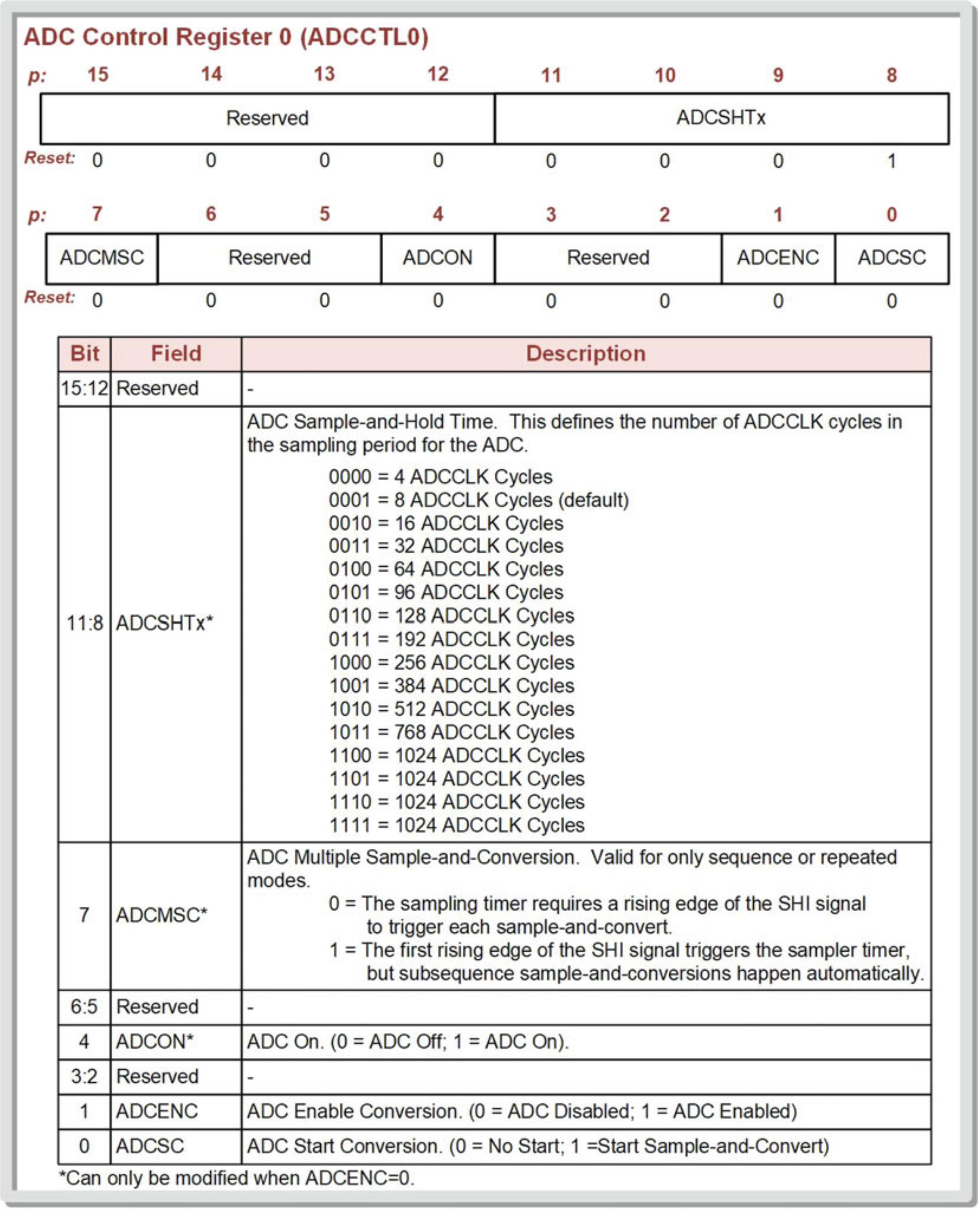


Fig. 15.3
Block diagram of ADC peripheral on the MSP430FR2355

The basic use model for the ADC peripheral is that it is first configured using a set of registers; then the conversion is started by the user (or by the successful completion of a prior conversion), and then the result of the conversion can be read from the ADC's conversion memory register. Flags can be used to track the status of the conversion and trigger interrupts to react to various states of the conversion (i.e., conversion complete). The complete list of ADC registers on the MSP430FR2355 MCU is as follows:

- *ADC control 0 (ADCCTL0) register*
- *ADC control 1 (ADCCTL1) register*
- *ADC control 2 (ADCCTL2) register*
- *ADC memory control (ADCMCTL0) register*
- *ADC conversion memory (ADCMEM0) register*
- *ADC interrupt enable (ADCIE) register*
- *ADC interrupt flag (ADCIFG) register*
- *ADC interrupt vector (ADCIV) register*
- *ADC window comparator low threshold (ADCLO) register*
- *ADC window comparator high threshold (ADCHI) register*.

The ADC is configured using four main registers: ADCCTL0, ADCCTL1, ADCCTL2, and ADCMCTL0. We will go through the settings in each of these registers that are needed to get a basic ADC program working. The ADCCTL0 register contains the settings for the number of ADCCLK cycles to use during the conversion (ADCSHTx), how the ADC is triggered (ADCMSC), turning the ADC on (ADCON), enabling the conversion (ADCENC), and starting a conversion (ADCSC). The ADCENC and ADCSC bits are used to start a conversion by asserting them simultaneously. All other settings are done in the initialization portion of the program. Figure 15.4 gives the details of ADCCTL0.



ADC Control Register 0 (ADCCTL0)

p:	15	14	13	12	11	10	9	8
	Reserved				ADCSHTx			
Reset:	0	0	0	0	0	0	0	1

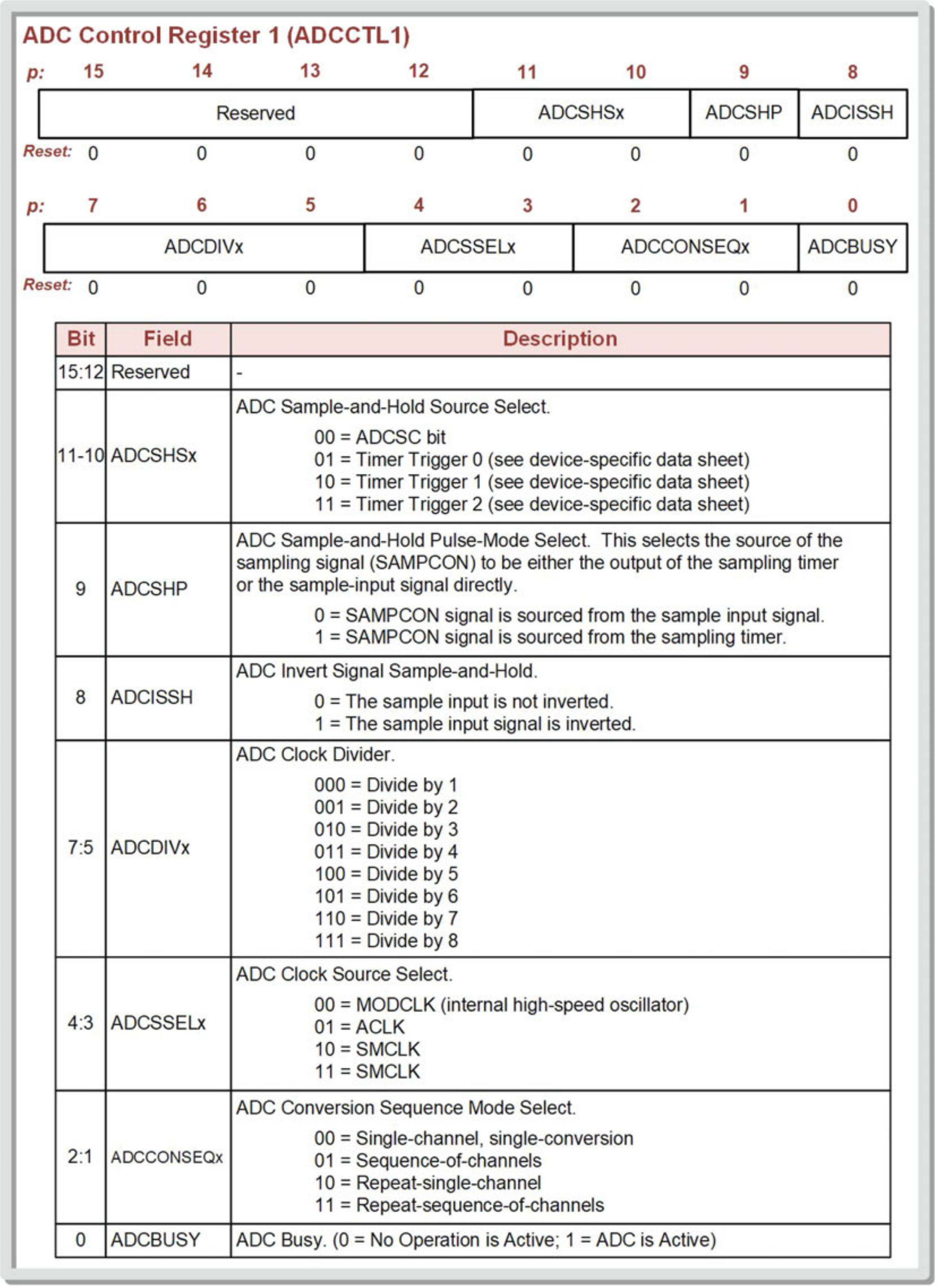
p:	7	6	5	4	3	2	1	0
	ADCMSC	Reserved		ADCON	Reserved		ADCENC	ADCSC
Reset:	0	0	0	0	0	0	0	0

Bit	Field	Description
15:12	Reserved	-
11:8	ADCSHTx*	ADC Sample-and-Hold Time. This defines the number of ADCCLK cycles in the sampling period for the ADC. 0000 = 4 ADCCLK Cycles 0001 = 8 ADCCLK Cycles (default) 0010 = 16 ADCCLK Cycles 0011 = 32 ADCCLK Cycles 0100 = 64 ADCCLK Cycles 0101 = 96 ADCCLK Cycles 0110 = 128 ADCCLK Cycles 0111 = 192 ADCCLK Cycles 1000 = 256 ADCCLK Cycles 1001 = 384 ADCCLK Cycles 1010 = 512 ADCCLK Cycles 1011 = 768 ADCCLK Cycles 1100 = 1024 ADCCLK Cycles 1101 = 1024 ADCCLK Cycles 1110 = 1024 ADCCLK Cycles 1111 = 1024 ADCCLK Cycles
7	ADCMSC*	ADC Multiple Sample-and-Conversion. Valid for only sequence or repeated modes. 0 = The sampling timer requires a rising edge of the SHI signal to trigger each sample-and-convert. 1 = The first rising edge of the SHI signal triggers the sampler timer, but subsequence sample-and-conversions happen automatically.
6:5	Reserved	-
4	ADCON*	ADC On. (0 = ADC Off; 1 = ADC On).
3:2	Reserved	-
1	ADCENC	ADC Enable Conversion. (0 = ADC Disabled; 1 = ADC Enabled)
0	ADCSC	ADC Start Conversion. (0 = No Start; 1 =Start Sample-and-Convert)

*Can only be modified when ADCENC=0.

Fig. 15.4
ADC control register 0 (ADCCTL0)

The ADCCTL1 register contains settings for the source of the sample trigger (ADCSHSx), an option for running the sampler in pulse mode (ADCSHP), an input inversion option (ADCISSH), settings for the second clock divider stage (ADCDIVx), the ADC clock source (ADCSSELx), whether the conversion is to run once or multiple times upon a trigger (ADCCONSEQx), and a status flag indicating the conversion is busy (ADCBSUSY). Figure 15.5 gives the details of ADCCTL1.



ADC Control Register 1 (ADCCTL1)

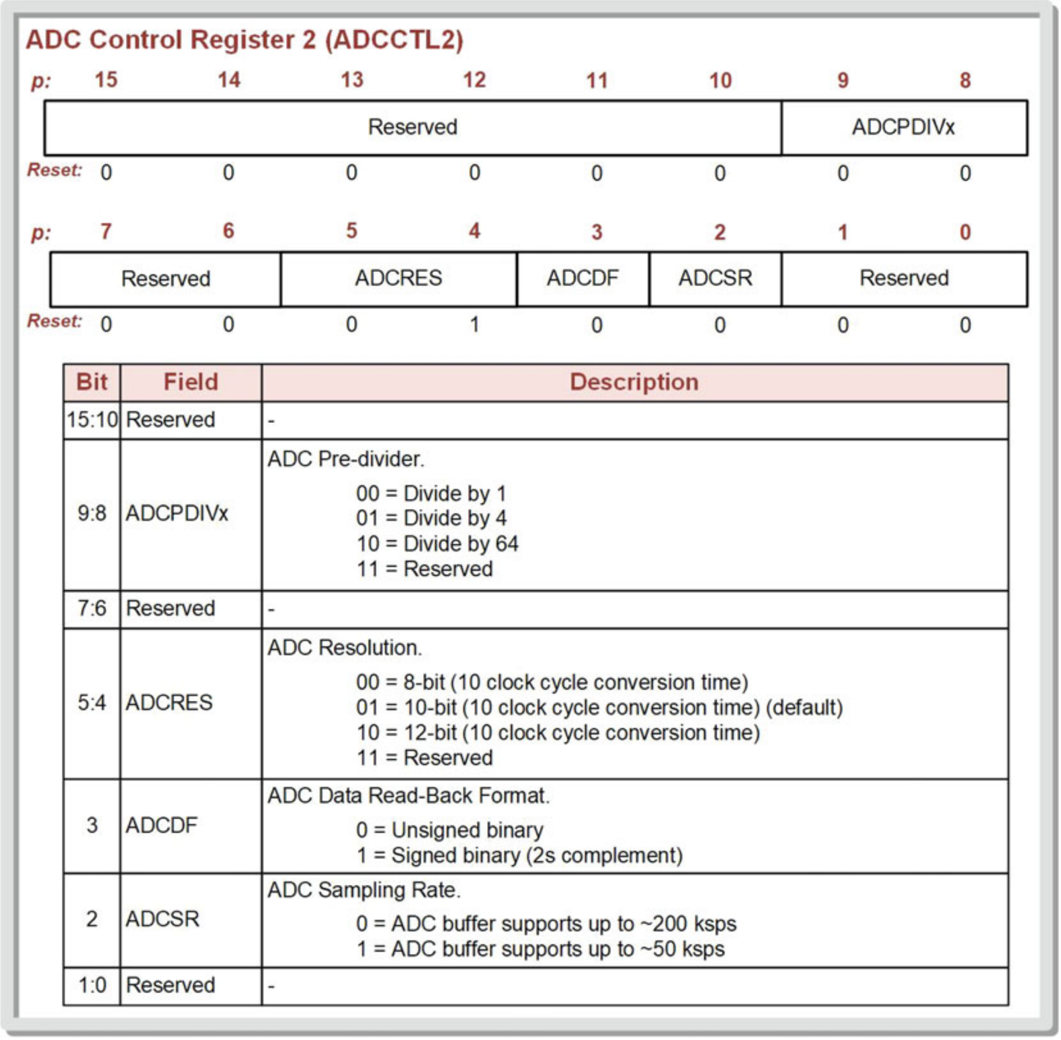
p:	15	14	13	12	11	10	9	8
	Reserved				ADCSHSx		ADCSHP	ADCISSH
Reset:	0	0	0	0	0	0	0	0

p:	7	6	5	4	3	2	1	0
	ADCDIVx			ADCSSELx		ADCCONSEQx		ADCBUSY
Reset:	0	0	0	0	0	0	0	0

Bit	Field	Description
15:12	Reserved	-
11-10	ADCSHSx	ADC Sample-and-Hold Source Select. 00 = ADCSC bit 01 = Timer Trigger 0 (see device-specific data sheet) 10 = Timer Trigger 1 (see device-specific data sheet) 11 = Timer Trigger 2 (see device-specific data sheet)
9	ADCSHP	ADC Sample-and-Hold Pulse-Mode Select. This selects the source of the sampling signal (SAMPCON) to be either the output of the sampling timer or the sample-input signal directly. 0 = SAMPCON signal is sourced from the sample input signal. 1 = SAMPCON signal is sourced from the sampling timer.
8	ADCISSH	ADC Invert Signal Sample-and-Hold. 0 = The sample input is not inverted. 1 = The sample input signal is inverted.
7:5	ADCDIVx	ADC Clock Divider. 000 = Divide by 1 001 = Divide by 2 010 = Divide by 3 011 = Divide by 4 100 = Divide by 5 101 = Divide by 6 110 = Divide by 7 111 = Divide by 8
4:3	ADCSSELx	ADC Clock Source Select. 00 = MODCLK (internal high-speed oscillator) 01 = ACLK 10 = SMCLK 11 = SMCLK
2:1	ADCCONSEQx	ADC Conversion Sequence Mode Select. 00 = Single-channel, single-conversion 01 = Sequence-of-channels 10 = Repeat-single-channel 11 = Repeat-sequence-of-channels
0	ADCBUSY	ADC Busy. (0 = No Operation is Active; 1 = ADC is Active)

Fig. 15.5
ADC control register 1 (ADCCTL1)

The ADCCTL2 register contains settings for the first clock divider stage (ADCPDIVx), the resolution of the ADC (ADCRES), the data format of the result (ADCDF), and the range for the anticipated sample rate (ADCSR). Note that the default setting for ADCRES is 01 (10 bits). If this setting is to be changed, the LSB of ADCRES needs to be cleared before the new settings are written. Figure 15.6 gives the details of ADCCTL2.



ADC Control Register 2 (ADCCTL2)

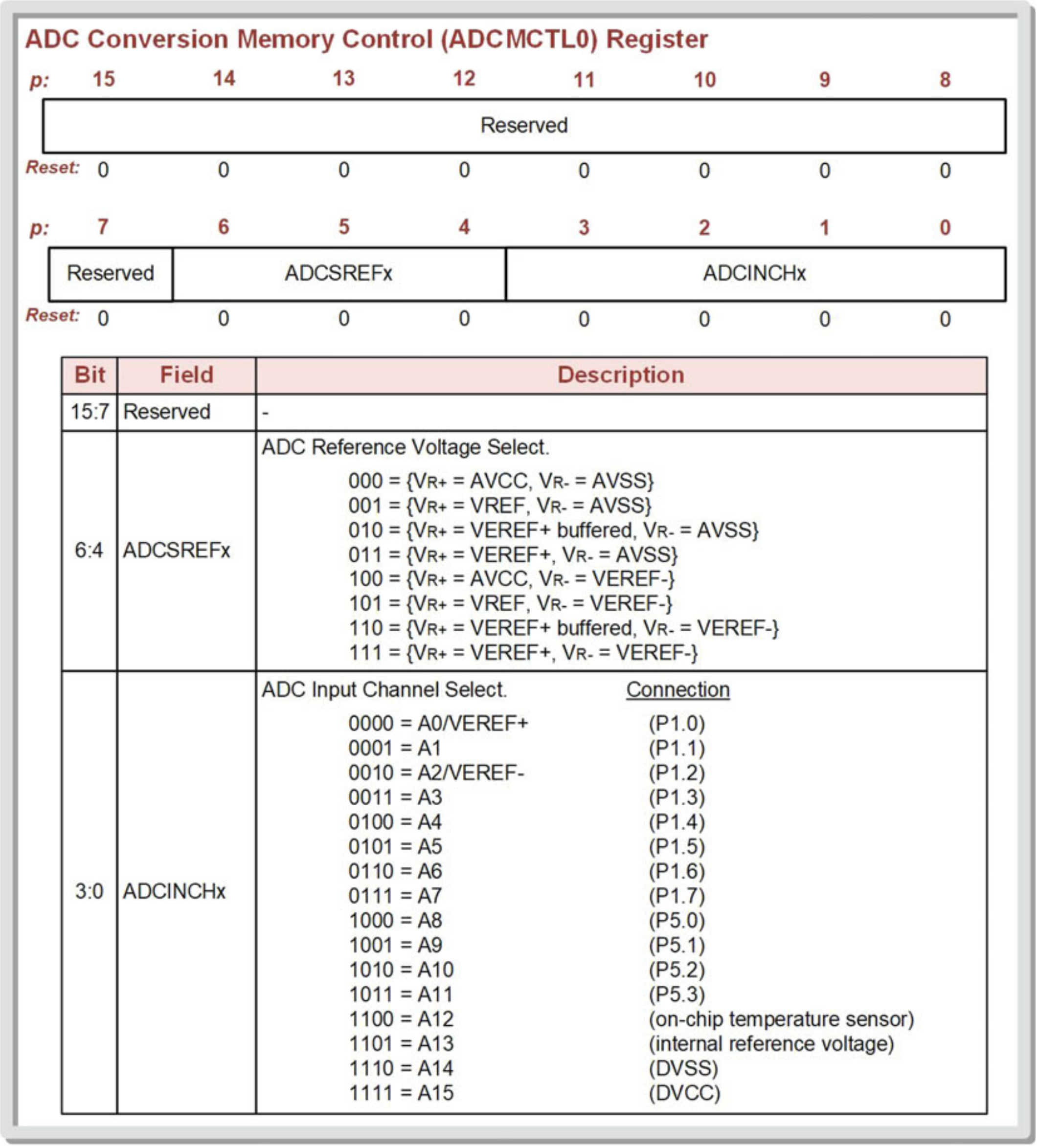
p:	15	14	13	12	11	10	9	8
	Reserved						ADCPDIVx	
Reset:	0	0	0	0	0	0	0	0

p:	7	6	5	4	3	2	1	0
	Reserved		ADCRES		ADCDF	ADCSR	Reserved	
Reset:	0	0	0	1	0	0	0	0

Bit	Field	Description
15:10	Reserved	-
9:8	ADCPDIVx	ADC Pre-divider. 00 = Divide by 1 01 = Divide by 4 10 = Divide by 64 11 = Reserved
7:6	Reserved	-
5:4	ADCRES	ADC Resolution. 00 = 8-bit (10 clock cycle conversion time) 01 = 10-bit (10 clock cycle conversion time) (default) 10 = 12-bit (10 clock cycle conversion time) 11 = Reserved
3	ADCDF	ADC Data Read-Back Format. 0 = Unsigned binary 1 = Signed binary (2s complement)
2	ADCSR	ADC Sampling Rate. 0 = ADC buffer supports up to ~200 ksps 1 = ADC buffer supports up to ~50 ksps
1:0	Reserved	-

Fig. 15.6
ADC control register 2 (ADCCTL2)

The ADCMCTL0 register contains settings for the reference voltage selection (ADCSREFx) and the ADC input channel that will be routed to the sample-and-hold stage (ADCINCHx). For the ADCSREFx settings, “VREF” refers to the internal reference voltages that the MCU can produce while “VEREF+/–” refers to external pins. Also, “AVCC” refers to the MCU power supply (+3.4 V) while “AVSS” refers to the MCU ground (0 V). The ADCINCHx setting allows 1 of 16 different input channels to be chosen. Of these 16 channels, 12 can come from MCU port pins. If any of these 12 port pins are to be used, they must also be configured for ADC usage using PxSEL1:PxSEL0. The other four options for ADC input channels include an internal temperature sensor, the internal reference voltage, the power supply (VCC), and GND (VSS). Figure 15.7 gives the details of ADCMCTL0.



ADC Conversion Memory Control (ADCMCTL0) Register

p:	15	14	13	12	11	10	9	8
	Reserved							
Reset:	0	0	0	0	0	0	0	0

p:	7	6	5	4	3	2	1	0
	Reserved	ADCSREFx			ADCINCHx			
Reset:	0	0	0	0	0	0	0	0

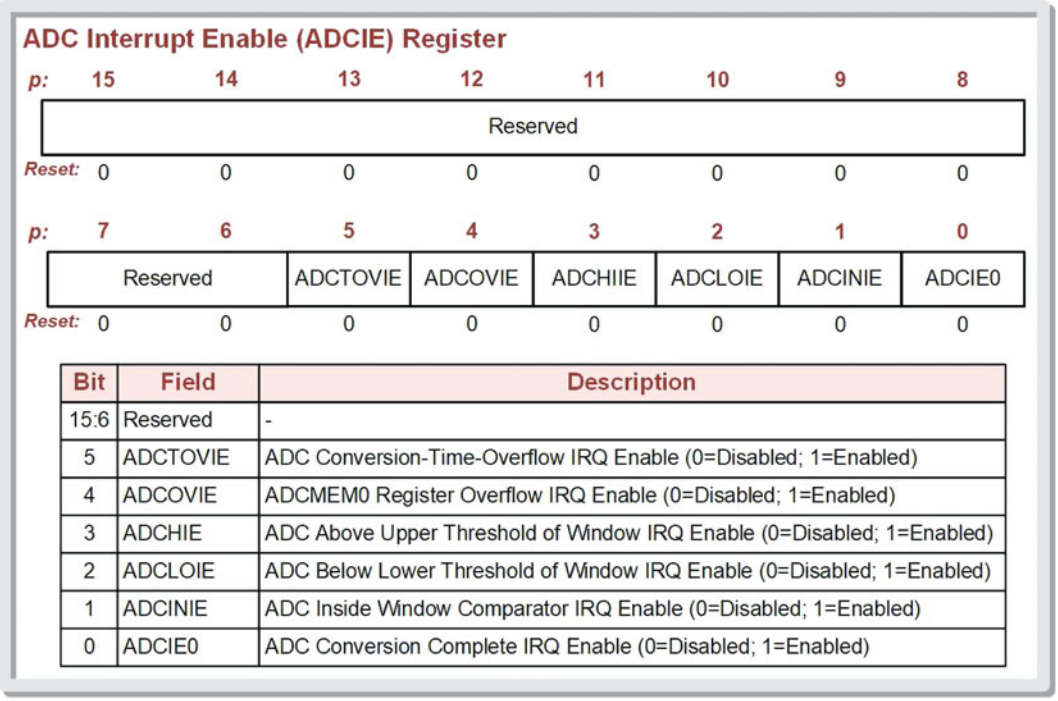
Bit	Field	Description
15:7	Reserved	-

6:4	ADCSREFx	ADC Reference Voltage Select.
		000 = {V_{R+} = AVCC, V_{R-} = AVSS}
		001 = {V_{R+} = VREF, V_{R-} = AVSS}
		010 = {V_{R+} = VEREF+ buffered, V_{R-} = AVSS}
		011 = {V_{R+} = VEREF+, V_{R-} = AVSS}
		100 = {V_{R+} = AVCC, V_{R-} = VEREF-}
		101 = {V_{R+} = VREF, V_{R-} = VEREF-}
		110 = {V_{R+} = VEREF+ buffered, V_{R-} = VEREF-}
		111 = {V_{R+} = VEREF+, V_{R-} = VEREF-}

3:0	ADCINCHx	ADC Input Channel Select. Connection 0000 = A0/VEREF+ (P1.0) 0001 = A1 (P1.1) 0010 = A2/VEREF- (P1.2) 0011 = A3 (P1.3) 0100 = A4 (P1.4) 0101 = A5 (P1.5) 0110 = A6 (P1.6) 0111 = A7 (P1.7) 1000 = A8 (P5.0) 1001 = A9 (P5.1) 1010 = A10 (P5.2) 1011 = A11 (P5.3) 1100 = A12 (on-chip temperature sensor) 1101 = A13 (internal reference voltage) 1110 = A14 (DVSS) 1111 = A15 (DVCC)

Fig. 15.7
ADC conversion memory control (ADCMCTL0) register

The ADC peripheral contains six interrupt flags that can be used to monitor the status of the conversion. There is an interrupt flag that will trigger when a conversion is complete (ADCIFG0). There are three interrupts that can be used with a *threshold window* feature where the ADC watches for whether the input is within the window (ADCINIFG), below the window (ADCLOIFG), or above the window (ADCHIIFG). There is also one interrupt to indicate if ADCMEM0 has been written to before the last conversion result has been read (ADCOVIFG). Finally, there is one interrupt to indicate that a new conversion is triggered before the current conversion has completed (ADCTOVIFG). These interrupts are maskable and are locally enabled within the ADCIE register (shown in Fig. 15.8). The flags for these ADC interrupts are held in the ADCIFG register (shown in Fig. 15.9). The ADCIV provides unique codes for simultaneous interrupts that can be used to set the priority of the response (Fig. 15.10). The ADC has one interrupt vector address.



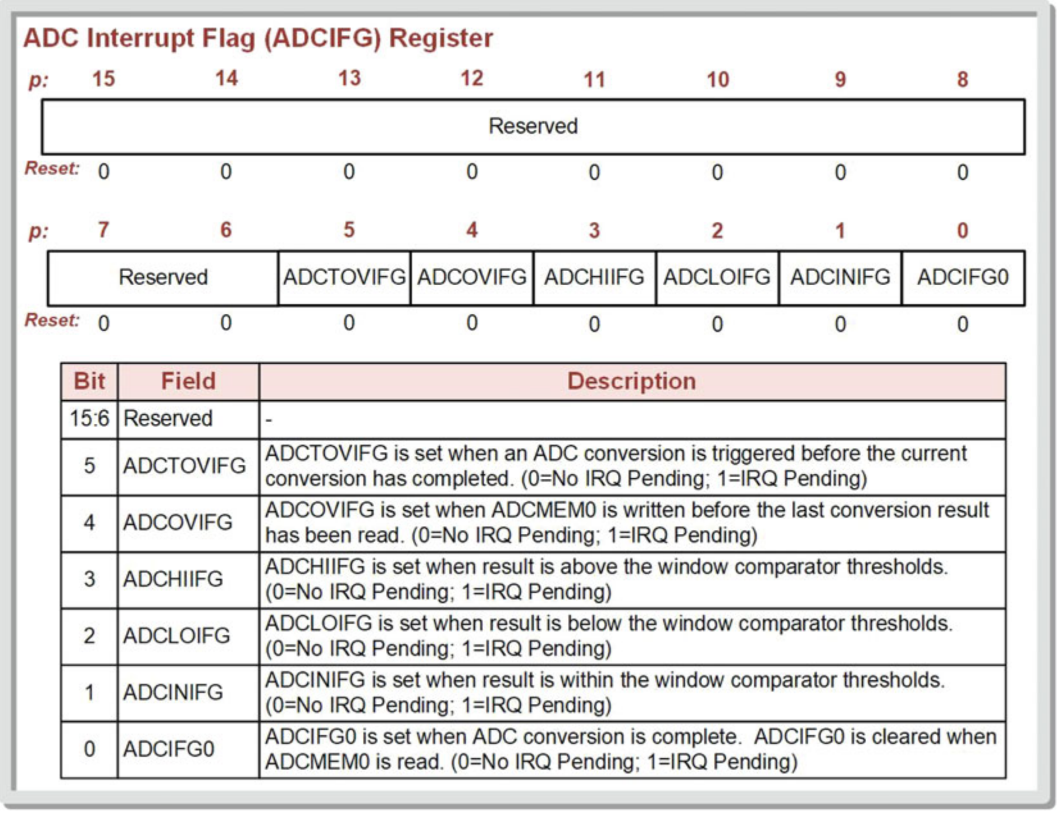
ADC Interrupt Enable (ADCIE) Register

p:	15	14	13	12	11	10	9	8
	Reserved							
Reset:	0	0	0	0	0	0	0	0

p:	7	6	5	4	3	2	1	0
	Reserved		ADCTOVIE	ADCOVIE	ADCHIIE	ADCLOIE	ADCINIE	ADCIE0
Reset:	0	0	0	0	0	0	0	0

Bit	Field	Description
15:6	Reserved	-
5	ADCTOVIE	ADC Conversion-Time-Overflow IRQ Enable (0=Disabled; 1=Enabled)
4	ADCOVIE	ADCMEM0 Register Overflow IRQ Enable (0=Disabled; 1=Enabled)
3	ADCHIE	ADC Above Upper Threshold of Window IRQ Enable (0=Disabled; 1=Enabled)
2	ADCLOIE	ADC Below Lower Threshold of Window IRQ Enable (0=Disabled; 1=Enabled)
1	ADCINIE	ADC Inside Window Comparator IRQ Enable (0=Disabled; 1=Enabled)
0	ADCIE0	ADC Conversion Complete IRQ Enable (0=Disabled; 1=Enabled)

Fig. 15.8
ADC interrupt enable (ADCIE) register



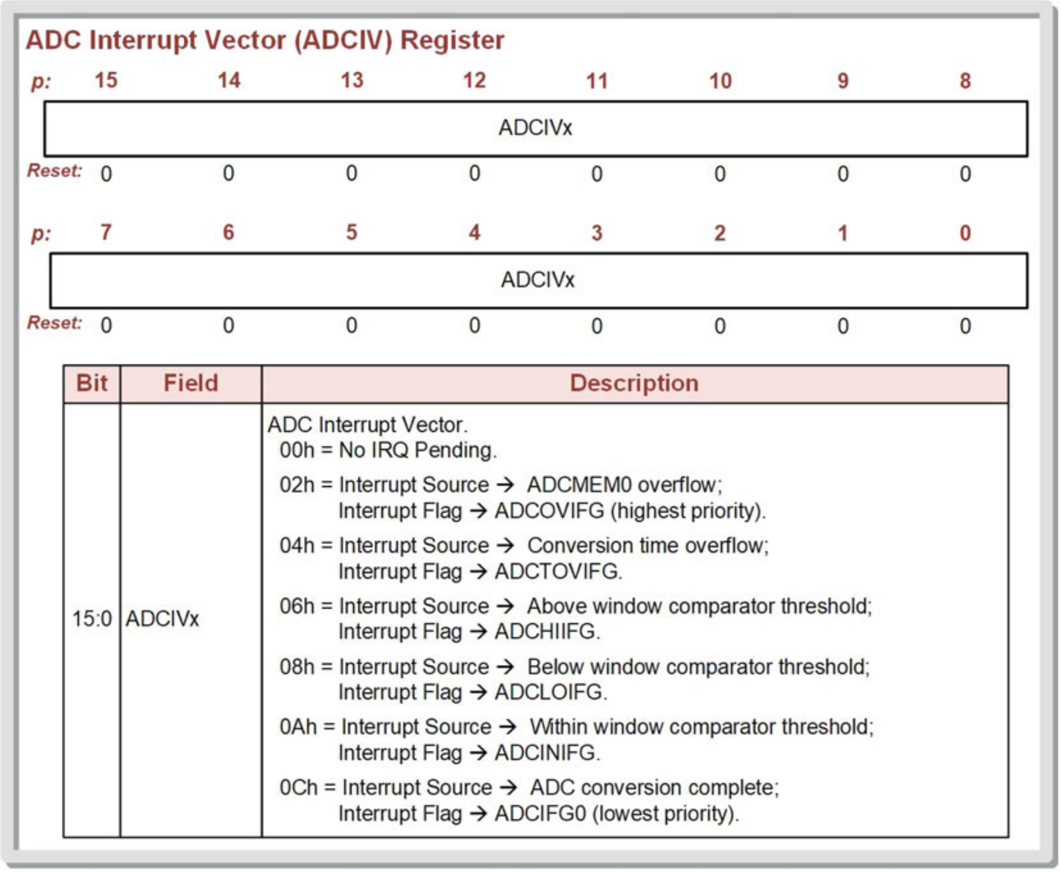
ADC Interrupt Flag (ADCIFG) Register

p:	15	14	13	12	11	10	9	8
	Reserved							
Reset:	0	0	0	0	0	0	0	0

p:	7	6	5	4	3	2	1	0
	Reserved		ADCTOVIFG	ADCOVIFG	ADCHIIFG	ADCLOIFG	ADCINIFG	ADCIFG0
Reset:	0	0	0	0	0	0	0	0

Bit	Field	Description
15:6	Reserved	-
5	ADCTOVIFG	ADCTOVIFG is set when an ADC conversion is triggered before the current conversion has completed. (0=No IRQ Pending; 1=IRQ Pending)
4	ADCOVIFG	ADCOVIFG is set when ADCMEM0 is written before the last conversion result has been read. (0=No IRQ Pending; 1=IRQ Pending)
3	ADCHIIFG	ADCHIIFG is set when result is above the window comparator thresholds. (0=No IRQ Pending; 1=IRQ Pending)
2	ADCLOIFG	ADCLOIFG is set when result is below the window comparator thresholds. (0=No IRQ Pending; 1=IRQ Pending)
1	ADCINIFG	ADCINIFG is set when result is within the window comparator thresholds. (0=No IRQ Pending; 1=IRQ Pending)
0	ADCIFG0	ADCIFG0 is set when ADC conversion is complete. ADCIFG0 is cleared when ADCMEM0 is read. (0=No IRQ Pending; 1=IRQ Pending)

Fig. 15.9
ADC interrupt flag (ADCIFG) register



ADC Interrupt Vector (ADCIV) Register

p:	15	14	13	12	11	10	9	8
	ADCIVx							
Reset:	0	0	0	0	0	0	0	0

p:	7	6	5	4	3	2	1	0
	ADCIVx							
Reset:	0	0	0	0	0	0	0	0

Bit	Field	Description
15:0	ADCIVx	ADC Interrupt Vector. 00h = No IRQ Pending. 02h = Interrupt Source → ADCMEM0 overflow; Interrupt Flag → ADCOVIFG (highest priority). 04h = Interrupt Source → Conversion time overflow; Interrupt Flag → ADCTOVIFG. 06h = Interrupt Source → Above window comparator threshold; Interrupt Flag → ADCHIIFG. 08h = Interrupt Source → Below window comparator threshold; Interrupt Flag → ADCLOIFG. 0Ah = Interrupt Source → Within window comparator threshold; Interrupt Flag → ADCINIFG. 0Ch = Interrupt Source → ADC conversion complete; Interrupt Flag → ADCIFG0 (lowest priority).

Fig. 15.10
ADC interrupt vector (ADCIV) register

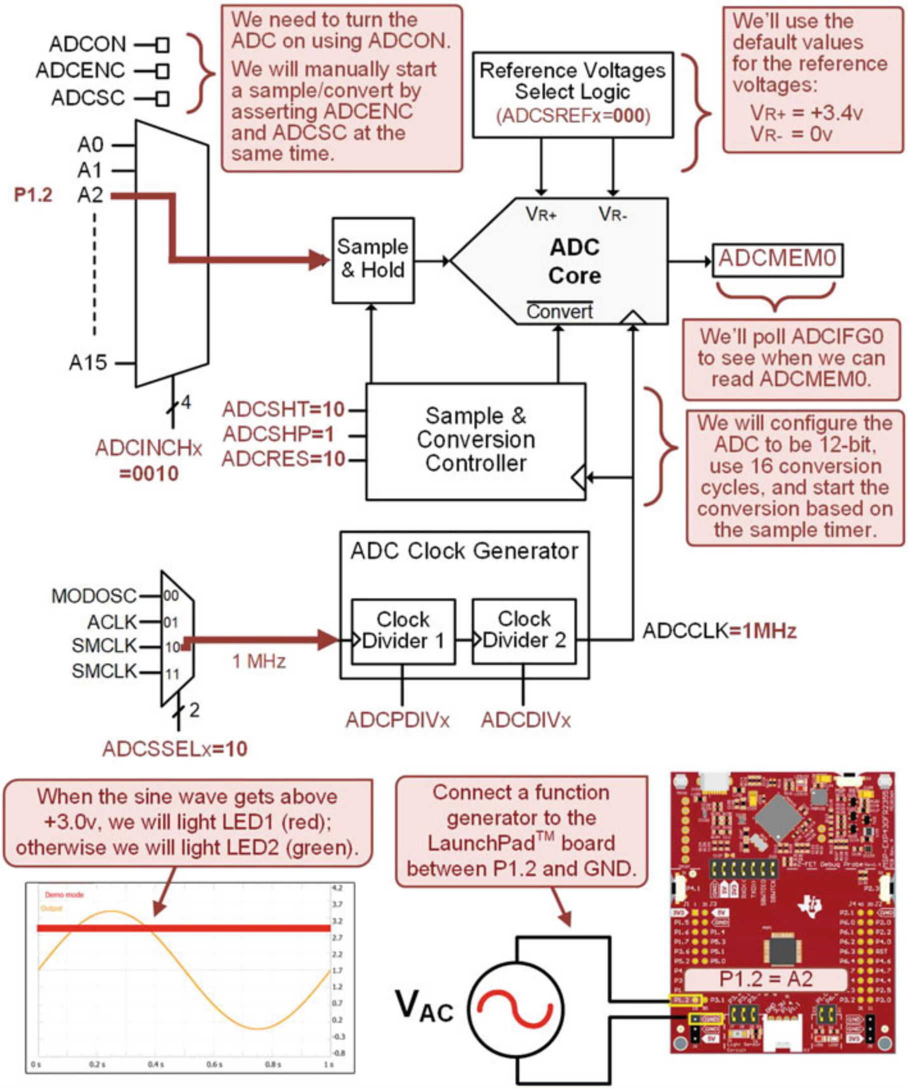
Let's now design a program to use the ADC on the MSP430FR2355 to digitize an analog voltage on P1.2. This pin can be driven on the LaunchPad™ board using a function generator. We will drive a sine wave voltage into P1.2 that goes from 0 to +3.4 V. We will continually digitize this signal and assert the LEDs depending on its value. When P1.2 is below +3.0 V, LED2 (green) will be asserted and LED1 will be off. When P1.2 is above +3.0 V, LED1 (red) will be asserted and LED2 will be off.

The first step to set up the ADC for this design is to use the P1SEL1:P1SEL0 registers to configure P1.2 to use its analog function (A2). Within the ADCCTL0 register, we will set the number of conversion cycles to 16 using ADCSHTand turn the ADC on using ADCON. Within ADCCTL0 we do not need to use ADCMSC because we are not going to use repeated mode. Within ADCCCTL1 we will select SMCLK as the ADC clock source using ADSSELx and select the sample timer as the source for the sample trigger using (ADCSHP). We will use the default values for ADCSHS (use ADCSC to trigger sample), ADCISSH (no inversion), and ADCDIVx (no clock division). We will also accept the default value for ADCCONSEQx, which is to configure the ADC for a single-channel, single-conversion that is triggered by user. This means each conversion will need to be manually started. We will not use ADCBUSY as we will monitor the status of the conversion using the ADCIFG0. Within ADCCTL2 we will set the resolution to 12-bits using ADCRES. We will use the default values for ADCPDIVx (no clock division), ADCDF (data formatted as unsigned binary), and ADCSR (support for sample rates up to ~200 ksps). Within ADCMCTL0 we will set the ADC input channel to A2. We will accept the default values for ADCSREFx, which uses $V_{R+} = \text{VCC}$ and $V_{R-} = 0$ V.

Within the main program loop, we will start the conversion by simultaneously asserting ADCENC and ADCSC in the ADCCTL0 register. After the conversion starts, we will wait in a polling loop until the conversion is finished by monitoring ADCIFG0. After the conversion completes, we will read the result from the ADCMEM0. Reading ADCMEM0 will clear ADCIFG0. We will then have logic to set the LEDs according to the ADC result. Examples 15.1 and 15.2 show this program.

Example: Reading an Analog Voltage With the ADC Using Polling to Monitor Conversion-Complete (Part 1)

We are going to set up the ADC on the LaunchPad™ board to read an analog voltage on P1.2. We will drive P1.2 with a sine wave voltage from a function generator. The sine wave will have an amplitude of 1.7v, an offset of 1.7v, and frequency of 1Hz. These settings will produce an analog voltage that will cycle between 0v and +3.4v every second. When the voltage is **below +3.0v**, we will light up LED2 (green) on the LaunchPad™ board. When the voltage is **above +3.0v**, we will light up LED1 (red) on the LaunchPad™ board. This program emulates an "over-voltage monitor" application. The following is the setup for this example.



We need to turn the ADC on using ADCON. We will manually start a sample/convert by asserting ADCENC and ADCSC at the same time.

We'll use the default values for the reference voltages: VR+ = +3.4v VR- = 0v

We'll poll ADCIFG0 to see when we can read ADCMEM0.

We will configure the ADC to be 12-bit, use 16 conversion cycles, and start the conversion based on the sample timer.

When the sine wave gets above +3.0v, we will light LED1 (red); otherwise we will light LED2 (green).

Connect a function generator to the LaunchPad™ board between P1.2 and GND.

Example 15.1
Reading an analog voltage with the ADC using polling to monitor conversion-complete (part 1)

Example: Reading an Analog Voltage With the ADC Using Polling to Monitor Conversion-Complete (Part 2)

1) In CCS, create a new C/C++ Empty Project (with main.c) titled: **C_ADC_Sampling_P1.2_Polling**.

2) Type in the following code in main.c after the statement to stop the watchdog timer.

```c
#include <msp430.h>
unsigned int ADC_Value;
int main(void)
{
    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer

    //-- Configure Ports
    P1DIR |= BIT0;              // Config P1.0 (LED1) as output
    P6DIR |= BIT6;              // Config P6.6 (LED2) as output

    P1SEL1  |= BIT2;            // Configure P1.2 Pin for A2
    P1SEL0  |= BIT2;

    PM5CTL0 &= ~LOCKLPM5;       // Turn on GPIO

    //-- Configure ADC
    ADCCTL0 &= ~ADCSHT;         // Clear ADCSHT from def. of ADCSHT=01
    ADCCTL0 |= ADCSHT_2;        // Conversion Cycles = 16 (ADCSHT=10)
    ADCCTL0 |= ADCON;           // Turn ADC ON

    ADCCTL1 |= ADCSSEL_2;       // ADC Clock Source = SMCLK
    ADCCTL1 |= ADCSHP;          // Sample signal source = sampling timer

    ADCCTL2 &= ~ADCRES;         // Clear ADCRES from def. of ADCRES=01
    ADCCTL2 |= ADCRES_2;        // Resolution = 12-bit (ADCRES=10)

    ADCMCTL0 |= ADCINCH_2;      // ADC Input Channel = A2 (P1.2)

    while(1)
    {
        ADCCTL0 |= ADCENC | ADCSC;          // Enable and Start conversion
        while((ADCIFG & ADCIFG0) == 0){}    // wait for conv. complete
        ADC_Value = ADCMEM0;                // Read ADC result

        if(ADC_Value > 3613){ // If (A2 > 3v)
            P1OUT |= BIT0;    //    LED1=ON (red)
            P6OUT &= ~BIT6;   //    LED2=OFF
        } else {              // If (A2 < 3v)
            P1OUT &= ~BIT0;   //    LED1=OFF
            P6OUT |= BIT6;    //    LED2=ON (green)
        }
    }
    return 0;
}
```

- Change mode for P1.2 to ADC channel A2.

- Conversion cycles = 16.
- ADC = on.

- Use SMCLK.
- Sample timer.

- 12-bit resolution

- Send A2 to ADC.

- Start ADC.

- Wait.

- Read result.

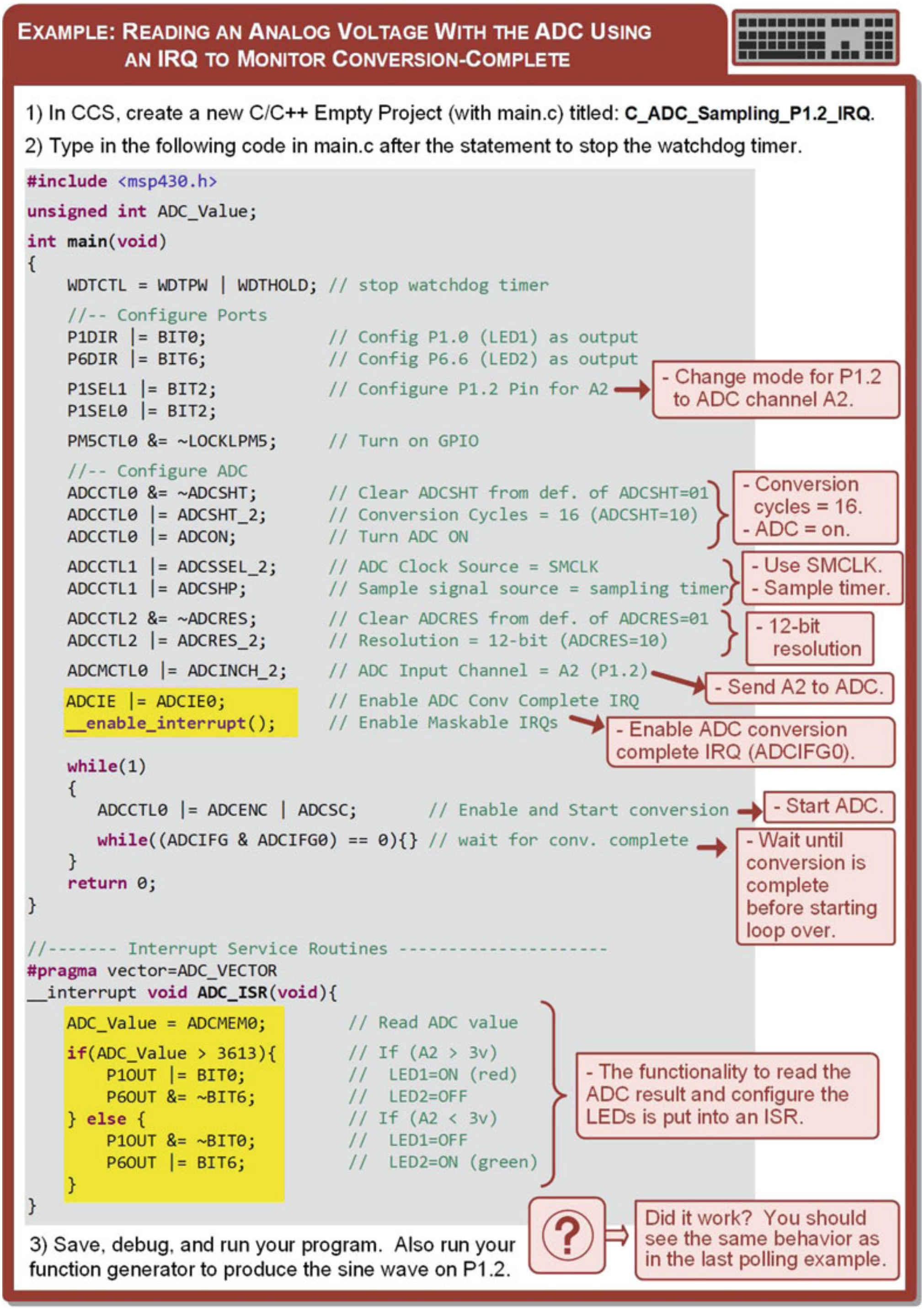
- Check the value that was read and assert / de assert the LEDs accordingly.

3) Save, debug, and run your program. Also run your function generator to produce the sine wave on P1.2.

Did it work? You should see the green LED on most of the time. But when the input signal gets above +3.0v during the top part of the sine wave, the red LED will turn on and the green LED will turn off.

Example 15.2
Reading an analog voltage with the ADC using polling to monitor conversion-complete (part 2)

Let's now look at how we can use an ADCIFG0 interrupt to read ADCMEM0 when the conversion completes. Example 15.3 shows the program to accomplish this. Note that this code uses the setup shown in Example 15.1.



EXAMPLE: READING AN ANALOG VOLTAGE WITH THE ADC USING AN IRQ TO MONITOR CONVERSION-COMPLETE

1) In CCS, create a new C/C++ Empty Project (with main.c) titled: **C_ADC_Sampling_P1.2_IRQ**.

2) Type in the following code in main.c after the statement to stop the watchdog timer.

```
#include <msp430.h>
unsigned int ADC_Value;
int main(void)
{
    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer
    //-- Configure Ports
    P1DIR  |= BIT0;             // Config P1.0 (LED1) as output
    P6DIR  |= BIT6;             // Config P6.6 (LED2) as output

    P1SEL1  |= BIT2;            // Configure P1.2 Pin for A2
    P1SEL0  |= BIT2;

    PM5CTL0 &= ~LOCKLPM5;       // Turn on GPIO

    //-- Configure ADC
    ADCCTL0 &= ~ADCSHT;         // Clear ADCSHT from def. of ADCSHT=01
    ADCCTL0 |= ADCSHT_2;        // Conversion Cycles = 16 (ADCSHT=10)
    ADCCTL0 |= ADCON;           // Turn ADC ON

    ADCCTL1 |= ADCSSEL_2;       // ADC Clock Source = SMCLK
    ADCCTL1 |= ADCSHP;          // Sample signal source = sampling timer

    ADCCTL2 &= ~ADCRES;         // Clear ADCRES from def. of ADCRES=01
    ADCCTL2 |= ADCRES_2;        // Resolution = 12-bit (ADCRES=10)

    ADCMCTL0 |= ADCINCH_2;      // ADC Input Channel = A2 (P1.2)

    ADCIE |= ADCIE0;            // Enable ADC Conv Complete IRQ
    __enable_interrupt();       // Enable Maskable IRQs

    while(1)
    {
        ADCCTL0 |= ADCENC | ADCSC;        // Enable and Start conversion
        while((ADCIFG & ADCIFG0) == 0){} // wait for conv. complete
    }
    return 0;
}

//------- Interrupt Service Routines ---------------------
#pragma vector=ADC_VECTOR
__interrupt void ADC_ISR(void){
    ADC_Value = ADCMEM0;          // Read ADC value
    if(ADC_Value > 3613){         // If (A2 > 3v)
        P1OUT |= BIT0;            //   LED1=ON (red)
        P6OUT &= ~BIT6;           //   LED2=OFF
    } else {                      // If (A2 < 3v)
        P1OUT &= ~BIT0;           //   LED1=OFF
        P6OUT |= BIT6;            //   LED2=ON (green)
    }
}
```

- Change mode for P1.2 to ADC channel A2.

- Conversion cycles = 16.
- ADC = on.

- Use SMCLK.
- Sample timer.

- 12-bit resolution

- Send A2 to ADC.

- Enable ADC conversion complete IRQ (ADCIFG0).

- Start ADC.

- Wait until conversion is complete before starting loop over.

- The functionality to read the ADC result and configure the LEDs is put into an ISR.

3) Save, debug, and run your program. Also run your function generator to produce the sine wave on P1.2.

? ⇨ Did it work? You should see the same behavior as in the last polling example.

Example 15.3
Reading an analog voltage with the ADC using an IRQ to monitor conversion-complete

Concept Check

CC15.2 Why does the MSP430FR2355 have resolution settings for 8-bit and 10-bit? Isn't the 12-bit setting always better?

A) The lower resolution settings don't take as long to perform the conversion, so they are faster.

B) Some applications have electrical noise that make the lower bits of the ADC result meaningless since they are always measuring noise.

C) The 8-bit setting can be used in applications where only one byte of data is desired.

D) All of the above.

Summary

- An ADC is a circuit that converts an analog voltage into a digital representation.
- An ADC uses a sample-and-hold stage to charge a capacitor to the same voltage as the input signal. The capacitor is able to hold this voltage while a conversion is performed.
- An ADC uses a conversion stage to convert the sampled voltage into a digital value.
- The resolution of an ADC is the number of bits (*n*) that are in the final digital output.
- An ADC will digitize across a programmable range of input voltages from V_{R+} to V_{R-}.
- The precision of an ADC is the amount of voltage that the LSB can represent. This depends on the input voltage range and number of bits in the ADC.
- The accuracy of an ADC is always ± ½ LSB of the final answer.
- The sample rate of an ADC represents how fast it samples the incoming signal. If the ADC is significantly faster than the input signal, the digital values can be used to reconstruct the original input signal.
- The MSP430FR2355 has one ADC core that can be programmed to have a resolution of 8-bits, 10-bits, or 12-bits. It can also digitize 1 of 16 possible input channels. It also has a selectable clock source.
- The ADC on the MSP430FR2355 has a variety of interrupts that can be used to track the status of the conversion.

Exercise Problems

Section 15.1: Analog-to-Digital Converters

15.1.1 What is the resolution of a 10-bit ADC?

15.1.2 What is the precision of a 10-bit ADC with an input voltage range of +5 V?

15.1.3 What is the precision of an 8-bit ADC with an input voltage range of +3.4 V?

15.1.4 What is the analog input voltage for a 10-bit ADC with an input voltage range of 5 V if $N_{ADC} = 0x00FF$?

15.1.5 If the digital output is of a 10-bit ADC with an input range of +5 V is +1.5 V, what is the range of voltages that the output could take on (i.e., its accuracy)?

Section 15.2: ADC Operation on the MSP430FR2355

15.2.1 How many input channels does the ADC on the MSP430FR2355 have?

15.2.2 How many of the input channels of the ADC on the MSP430FR2355 can come from ports?

15.2.3 How many resolution settings does the ADC on the MSP430FR2355 have?

15.2.4 What happens to ADCIFG0 when ADCMEM0 is read?

15.2.5 What is a more efficient way to wait for the ADC conversion to complete than polling the ADCIFG0 flag?

Chapter 16: The Clock System

This chapter looks at the *clock system* (CS) of the MSP430FR2355 [1–3, 14]. The CS generates the internal clocks that are used by the CPU, memory, and peripherals. The CS includes on-chip oscillators and support for external clock sources. The CS generates multiple clock sources that can be used to drive the peripherals at different frequencies than the CPU to optimize performance and reduce power. This chapter gives an overview of the clock system on the MSP430FR2355 and how to design a program in C to change settings from its default configuration at power up.

Learning Outcomes—After completing this chapter you will be able to:

16.1 Describe the basic concept of MSP430FR2355's clock system.
16.2 Design programs that configures the MSP430FR2355's clock system in C.

16.1 Overview of the MSP430FR2355 Clock System

The CS generates the various clocks that are used by the CPU, memory, and peripherals. The MSP430 has two CS modes, *basic* and *enhanced*. The enhanced mode provides more options for clock sources and frequency values than basic mode. The MSP430FR2355 implements the enhanced CS mode so the material is this chapter is relative to that particular mode. If using a different MSP430 MCU, consult the device-specific data sheet to determine the features of the device's clock system.

The MSP430FR2355 contains four on-chip oscillators and supports one external clock source that can be configured in either high frequency (HF) and low frequency (LF) mode. From these sources the CS creates a *master clock* (MCLK), a *subsystem master clock* (SMCLK), and an *auxiliary clock* (ACLK) that are used by the CPU, memory, and peripherals. The CS has the ability create an abundant number of clock rates for these clock sources through frequency dividers and multipliers.

16.1.1 Internal Very Low-Power Low-Frequency Oscillator (VLO)

VLO is a ~10 kHz, on-chip oscillator suited for low-power operation. In the MSP430 documentation, the oscillator itself is given the name VLO while the signal that comes from the oscillator is called *VLOCLK*; however, throughout the data sheets, the two names are often used interchangeably. VLO can be used for MCLK, SMCLK, and ACLK. VLO is appropriate for extremely low-power operation that does not require an accurate time base.

16.1.2 Internal Trimmed Low-Frequency Reference Oscillator (REFO)

REFO is a 32.768 kHz, on-chip oscillator. This oscillator is *trimmed*, meaning that the manufacturer calibrates the oscillator during final test to provide a very accurate frequency (±3.5%). In the MSP430 documentation, the oscillator itself is given the name REFO while the signal that comes from the oscillator is called *REFOCLK*. REFO can be used for MCLK, SMCLK, and ACLK. REFO is appropriate when a slower, yet accurate, time base is needed.

16.1.3 External XT1 Oscillator (XT1)

The MSP430FR2355 can accept an external clock signal through its XIN and XOUT pins. On the MSP430FR2355, XIN and XOUT are tertiary functions for port 2.7 and port 2.6, respectively. As such, in order to use them as inputs the P2SEL1:P2SEL0 bits must be configured. By default, the pins are configured as digital I/O so XT1 is disabled.

B. J. LaMeres, *Embedded Systems Design using the MSP430FR2355 LaunchPad™*,
https://doi.org/10.1007/978-3-031-20888-1_16

When in LF mode, XT1 must be driven with an off-chip, 32.768 kHz watch crystal. The XIN and XOUT pins are connected to either side of the crystal. When in HF mode, the XIN pin can be driven with an off-chip ceramic oscillator at frequencies higher than 32.768 kHz. Note that XT1 is *either* in LF or HF mode at any time, not both. Within the MCU there are a variety of settings that can be configured on the incoming signal before it is used as a source for other systems within the CS. XT1 can be used to drive MCLK, SMCLK, and ACLK. When using XT1 for ACLK in HF mode, there is an additional divider block that can be used to divide down the incoming signal to ~32 kHz.

16.1.4 Internal Digitally Controlled Oscillator (DCO)

The DCO is an on-chip, trimmed, and programmable oscillator with a frequency that can be configured with software. The DCO can produce frequencies of 1, 2, 4, 8, 12, 16, 20, and 24 MHz for use by MCLK and SMCLK. The DCO can run in two different configurations. The first is an open loop, non-stabilized configuration. In this configuration, the DCO simply attempts to create a clock that is close to the desired frequency based on the user's configuration settings; however, it has the potential for the frequency to shift over time due to temperature and voltage variations. The second configuration is using a frequency locked loop (FLL) that implements a feedback loop to actively mitigate frequency shifting by stabilizing (or locking) the DCO to a reference clock. In the MSP430 documentation, the oscillator itself is given the name DCO while the signal that comes from the oscillator is called *DCOCLK*. DCOCLK goes through a final divider stage before being able to be used as MCLK and SMCLK, so the output of the DCO that is used is a signal called DCOCLKDIV. Figure 16.1 shows the DCO block diagram.

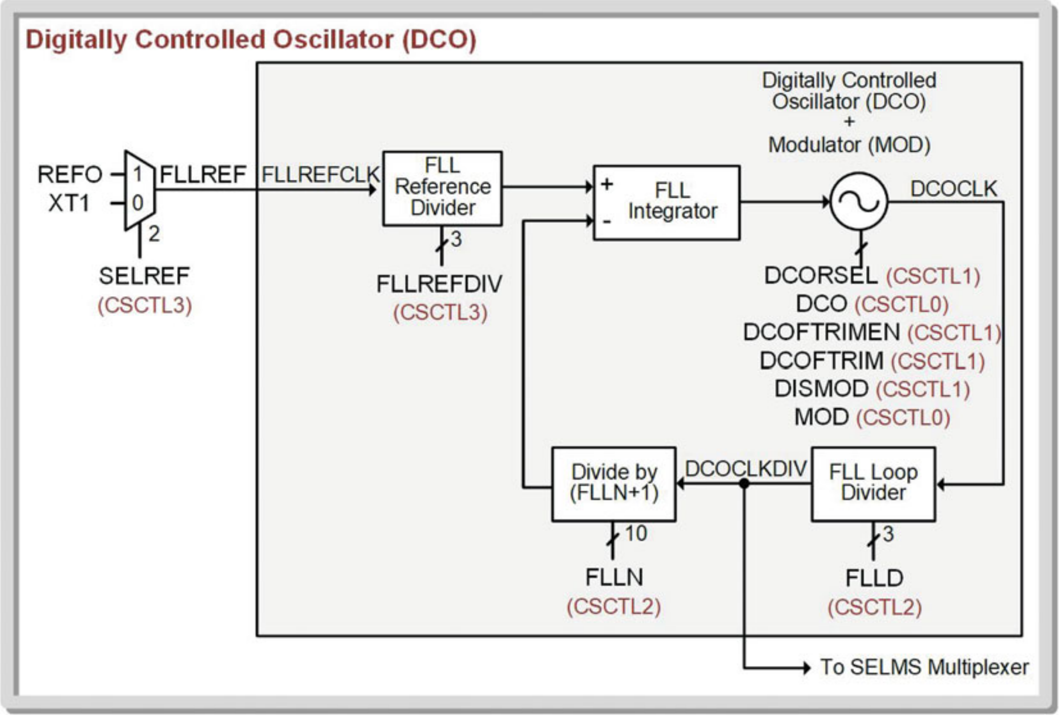


Fig. 16.1
Frequency locked loop (FLL) block diagram and configuration

By default, the DCO is in the FLL configuration. The basic concept of the FLL is that a feedback loop attempts to keep the DCOCLK at the specified frequency by continually updating the 9× control bits of the DCO (which are also called *DCO*). The output of the DCO is fed back to an FLL integrator (after going through multiple divider stages). The FLL integrator compares the feedback signal (the "−" input of the FLL Integrator in Fig. 16.1) to the reference clock (the "+" input of the FLL Integrator in Fig. 16.1) and attempts to make them the same frequency by continuously adjusting the DCO control bits. Since the feedback signal from DCO goes through divider stages to create a slower clock that is fed into the integrator, the DCOCLK signal itself can be much higher than FLLREFCLK. This is how the DCO system can create clock frequencies up to 24 MHz while using a FLLREFCLK that is as slow as 32.768 kHz.

The DCO on the MSP430FR2355 can use either REFO or XT1 as FLLREFCLK and has the ability to divide down the reference clock prior to going to the integrator. By default, the REFCLK comes from XT1. However, this depends on XT1 being enabled manually by altering the port function select bits for its external pins. If the port function select bits are left in their default digital I/O mode, the REFO clock is automatically selected as the reference.

There are a large number of control bits for the DCO, many of which are disabled upon reset. The most basic control of the DCO frequency comes from using the *DCO Range Select* (DCORSEL) bits and the FLLD and FLLN dividers within the feedback loop. The 3 bits of DCORSEL put the DCO frequency (DCOCLK) into one of 8× frequency ranges. Each range is around the 8× target frequencies of the MSP430FR2355 (i.e., 1, 2, 4, 8, 12, 16, 20, and 24 MHz). However, the DCO is not stabilized, or locked, to the final frequency until the dividers are setup to produce the correct control frequency that is as close to FLLREFCLK as possible. There are two dividers that set the frequency that is sent back to the integrator. The first is the FLL loop divider (FLLD), and the second is the FLLN divider. The FLLD and FLLD dividers should divide down DCOCLK to get as close to the reference clock as possible. From there, the integrator begins continuously altering the 9× DCO control bits to make the feedback signal match the reference clock. The 9× control bits give the ability to divide each DCORSEL range into 512 separate sub-ranges, or *taps*, that can be chosen in order to get the feedback signal as close to the reference frequency as possible. Once the two clock signals are as close as they can possible be, the FLL is said to be *locked* and it will produce a constant DCOCLK and DCOCLKDIV signal that can be used as MCLK and SMCLK thereafter. As an example of how the DCO FLL works, let's consider the default configuration of the system upon reset shown in Fig. 16.2.

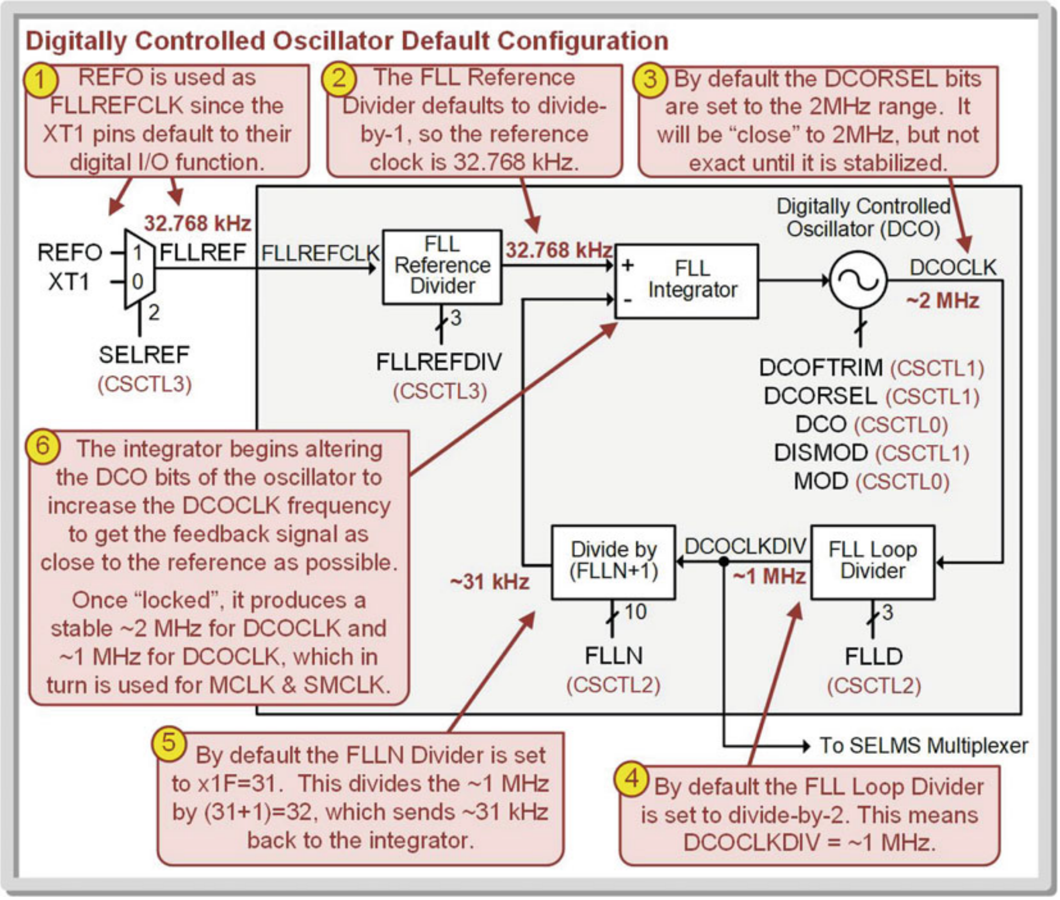

Digitally Controlled Oscillator Default Configuration
1 REFO is used as FLLREFCLK since the XT1 pins default to their digital I/O function.
2 The FLL Reference Divider defaults to divide-by-1, so the reference clock is 32.768 kHz.
3 By default the DCORSEL bits are set to the 2MHz range. It will be "close" to 2MHz, but not exact until it is stabilized.
32.768 kHz
REFO
XT1
1
0
2
FLLREF
FLLREFCLK
SELREF
(CSCTL3)
FLL Reference Divider
3
FLLREFDIV
(CSCTL3)
32.768 kHz
+
-
FLL Integrator
Digitally Controlled Oscillator (DCO)
DCOCLK
~2 MHz
DCOFTRIM (CSCTL1)
DCORSEL (CSCTL1)
DCO (CSCTL0)
DISMOD (CSCTL1)
MOD (CSCTL0)
6 The integrator begins altering the DCO bits of the oscillator to increase the DCOCLK frequency to get the feedback signal as close to the reference as possible.
Once "locked", it produces a stable ~2 MHz for DCOCLK and ~1 MHz for DCOCLK, which in turn is used for MCLK & SMCLK.
~31 kHz
Divide by (FLLN+1)
10
FLLN
(CSCTL2)
DCOCLKDIV
~1 MHz
FLL Loop Divider
3
FLLD
(CSCTL2)
To SELMS Multiplexer
5 By default the FLLN Divider is set to x1F=31. This divides the ~1 MHz by (31+1)=32, which sends ~31 kHz back to the integrator.
4 By default the FLL Loop Divider is set to divide-by-2. This means DCOCLKDIV = ~1 MHz.

Fig. 16.2
DCO default configuration

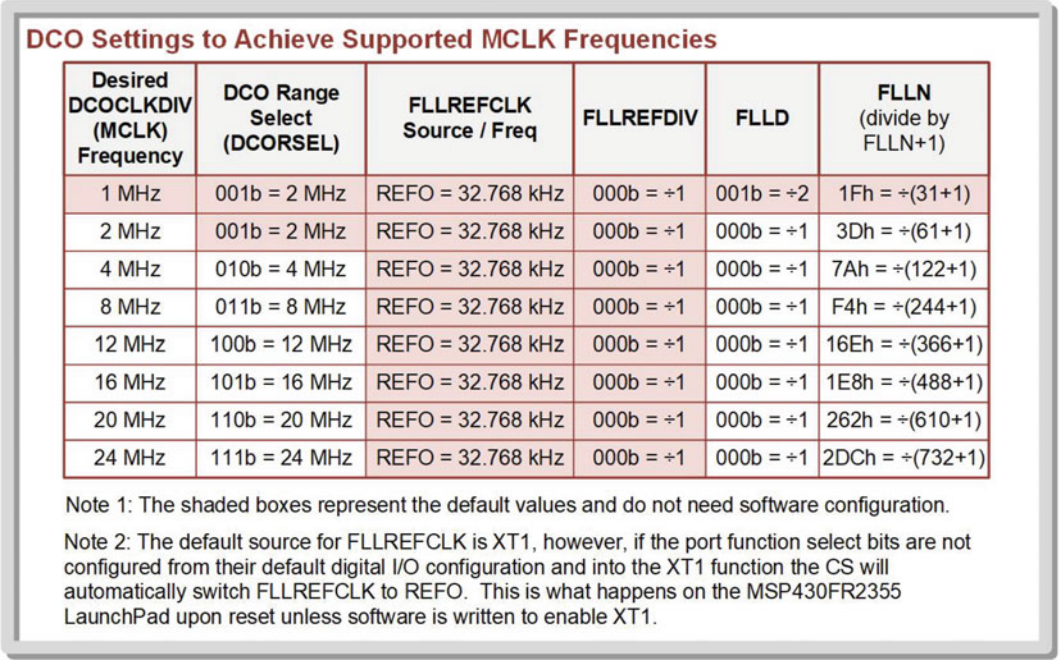
The frequency of DCOCLK can be found by multiplying the REFCLK by the 3× divider ratios in the FLL. In the default configuration shown in Fig. 16.2, DCOCLK = (f_{REFCLK} × FLLREFDIV × (FLLN + 1) × FLLD) = (32.768 k × 1 × 32 × 2) = 2 MHz. Note that in this calculation we are using the value of the divider, not the actual settings of the control bits (i.e., setting FLLD to a value of 100b means divide-by-16 so we use 16 in this calculation).

The DCO contains signals that alert the user if frequency lock is lost. The FLLUNLOCK bits are continuously updated to indicate if the FLL is locked or if DCOCLK is currently too high or too low.

The DCO also contains a *modulator* (disabled by default) that can be used to achieve an even more accurate frequency than the default FLL configuration. In the case where the final frequency desired for DCOCLK falls between two of the DCO tap values, the modulator can be used to continuously toggle back and forth between the two frequency taps to give an average frequency that is directly between the taps. The modulator is enabled by the DISMOD bit. When enabled, the FLL integrator automatically controls both the DCO and MOD bits.

If it is desired to run the DCO in an open-loop configuration (not recommended), there are additional control bits that can be used to adjust the DCOCLK frequency. First, the FLL can be disabled by setting either the SCG0 or SCG1 bits in the SR. When the FLL is disabled, the DCO operates based solely on its configuration settings instead of using active control. In the open loop configuration, the DCO bits can be configured manually by software. An additional setting called *DCO frequency trim (DCOFTRIM)* can be enabled using DCOFTRIMEN and provides a course adjustment of the DCOCLK frequency in 8× steps within the DCORSEL. The DCOFTRIM can be thought of as a course turning while the DCO bits provide a fine tuning.

While there are an abundant number of settings for the DCO, the most minimal configuration of DCOCLK and DCOCLKDIV comes from simply adjusting DCORSEL, FLLD, and FLLN and using the default source for FLLREFCLK (REFO). Table 16.1 gives some example configuration values to achieve each of the clock frequencies supported in the MSP430FR2355 using as many default settings as possible.



DCO Settings to Achieve Supported MCLK Frequencies

Desired DCOCLKDIV (MCLK) Frequency	DCO Range Select (DCORSEL)	FLLREFCLK Source / Freq	FLLREFDIV	FLLD	FLLN (divide by FLLN+1)
1 MHz	001b = 2 MHz	REFO = 32.768 kHz	000b = ÷1	001b = ÷2	1Fh = ÷(31+1)
2 MHz	001b = 2 MHz	REFO = 32.768 kHz	000b = ÷1	000b = ÷1	3Dh = ÷(61+1)
4 MHz	010b = 4 MHz	REFO = 32.768 kHz	000b = ÷1	000b = ÷1	7Ah = ÷(122+1)
8 MHz	011b = 8 MHz	REFO = 32.768 kHz	000b = ÷1	000b = ÷1	F4h = ÷(244+1)
12 MHz	100b = 12 MHz	REFO = 32.768 kHz	000b = ÷1	000b = ÷1	16Eh = ÷(366+1)
16 MHz	101b = 16 MHz	REFO = 32.768 kHz	000b = ÷1	000b = ÷1	1E8h = ÷(488+1)
20 MHz	110b = 20 MHz	REFO = 32.768 kHz	000b = ÷1	000b = ÷1	262h = ÷(610+1)
24 MHz	111b = 24 MHz	REFO = 32.768 kHz	000b = ÷1	000b = ÷1	2DCh = ÷(732+1)

Note 1: The shaded boxes represent the default values and do not need software configuration.

Note 2: The default source for FLLREFCLK is XT1, however, if the port function select bits are not configured from their default digital I/O configuration and into the XT1 function the CS will automatically switch FLLREFCLK to REFO. This is what happens on the MSP430FR2355 LaunchPad upon reset unless software is written to enable XT1.

Table 16.1
DCO settings to achieve supported MCLK frequencies

16.1.5 Internal High-Frequency Oscillator (MODCLK)

MODCLK is an on-chip, 5 MHz oscillator. MODCLK is only an option for the ADC and I2C peripherals. In the MSP430 documentation, the oscillator itself is given the names MODO or MODOSC while the signal that comes from the oscillator is called *MODCLK*.

16.1.6 Master Clock (MCLK)

MCLK is the clock source for the CPU, memory, digital I/O, interrupt system, CRC, and hardware multiplier. The source of MCLK is selectable from either REFOCLK, DCOCLKDIV, XT1CLK, or VLOCLK. MCLK can be pre-divided by an MCLK divider stage. MCLK is generally the fastest clock in the MCU.

16.1.7 Subsystem Master Clock (SMCLK)

SMCLK is a clock option for peripherals that can operate independently from the CPU such as timers, eUSCIs, and the ADC. SMCLK is always derived from MCLK and can be pre-divided by an SMCLK divider stage. Figure 16.3 gives the block diagram for the MCLK and SMCLK generation within the CS.

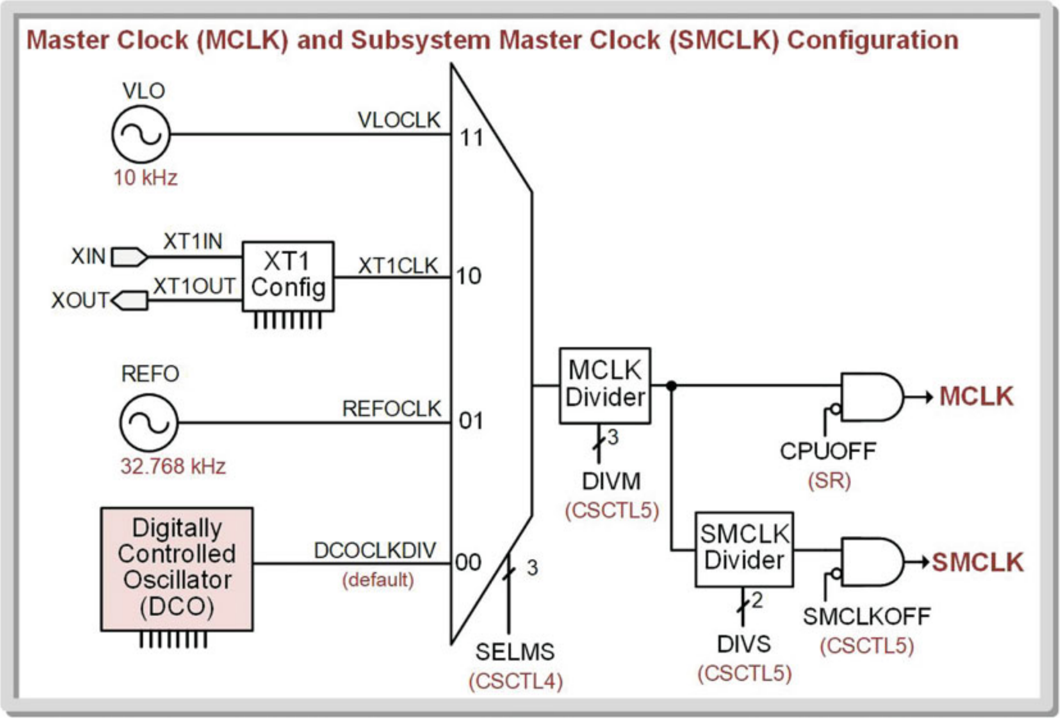
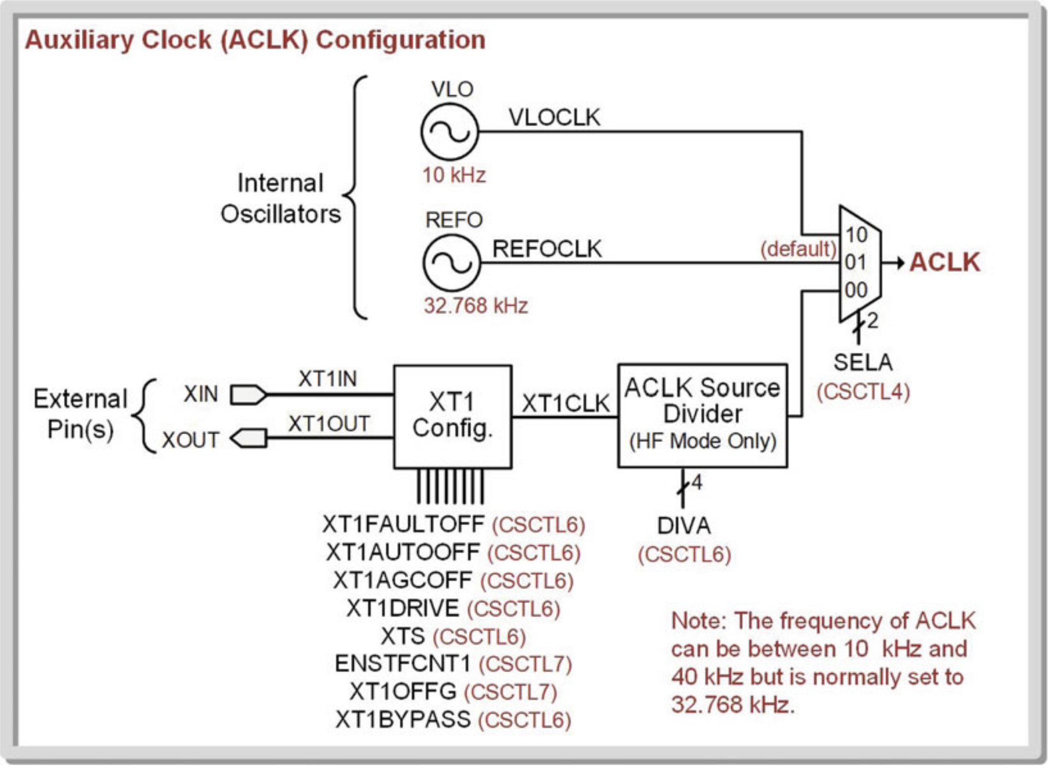


Fig. 16.3
Master clock (MCLK) overview

16.1.8 Auxiliary Clock (ACLK)

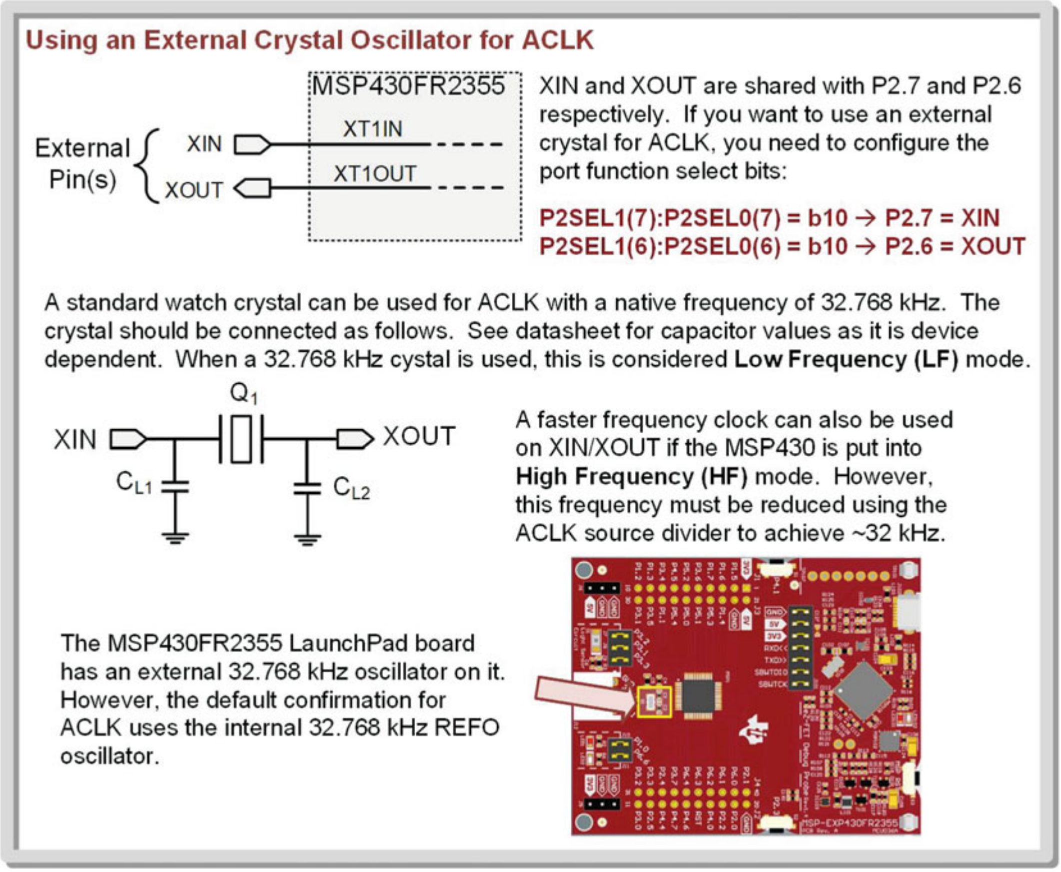
ACLK can be used for peripherals that operate independent from the CPU such as timers, eUSCIs, and the ADC. ACLK is appropriate when a peripheral requires a low-frequency, but accurate clock. The source of ACLK is software selectable from either VLO, REFO, or XT1. The frequency of ACLK is generally 32.768 kHz, but it is possible to operate it as low as 10 kHz when selecting VLO as the source or as high as 40 kHz when using XT1 as the source. Since XT1 in HF mode can accept a clock source that has a frequency higher than 40 kHz for use as MCLK, a divider stage is included in order to reduce the version of XT1 that is used for ACLK. The ACLK source divider can be configured to reduce XT1 in HF mode to be within the 10–40 kHz range. The device specific data sheets give guidance on this situation including the recommended external ceramic clock oscillators to use with the corresponding divider settings. When XT1 is connected direct to a 32.768 kHz watch crystal and in LF mode, the ACLK source divider is set to 1 and not configurable. Figure 16.4 gives the block diagram for the ACLK generation within the CS.



Note: The frequency of ACLK can be between 10 kHz and 40 kHz but is normally set to 32.768 kHz.

Fig. 16.4
Auxiliary clock (MCLK) configuration

When using an external watch crystal for ACLK, the XIN and XOUT pins are connected to either side of the external crystal with load capacitors (C_{L1} and C_{L2}) connected on each pin to ground. The value of the load capacitors to use is found in the device-specific data sheet. The MSP430FR2355 LaunchPad™ board contains a 32.768 kHz watch crystal for as ACLK or MCLK/SMCLK through XIN and XOUT. Figure 16.5 shows the crystal details for the LaunchPad™ board.



Using an External Crystal Oscillator for ACLK

XIN and XOUT are shared with P2.7 and P2.6 respectively. If you want to use an external crystal for ACLK, you need to configure the port function select bits:

P2SEL1(7):P2SEL0(7) = b10 → P2.7 = XIN
P2SEL1(6):P2SEL0(6) = b10 → P2.6 = XOUT

A standard watch crystal can be used for ACLK with a native frequency of 32.768 kHz. The crystal should be connected as follows. See datasheet for capacitor values as it is device dependent. When a 32.768 kHz cystal is used, this is considered **Low Frequency (LF)** mode.

A faster frequency clock can also be used on XIN/XOUT if the MSP430 is put into **High Frequency (HF)** mode. However, this frequency must be reduced using the ACLK source divider to achieve ~32 kHz.

The MSP430FR2355 LaunchPad board has an external 32.768 kHz oscillator on it. However, the default confirmation for ACLK uses the internal 32.768 kHz REFO oscillator.

Fig. 16.5
Using an external crystal oscillator for ACLK

16.1.9 Default Settings on Power-Up

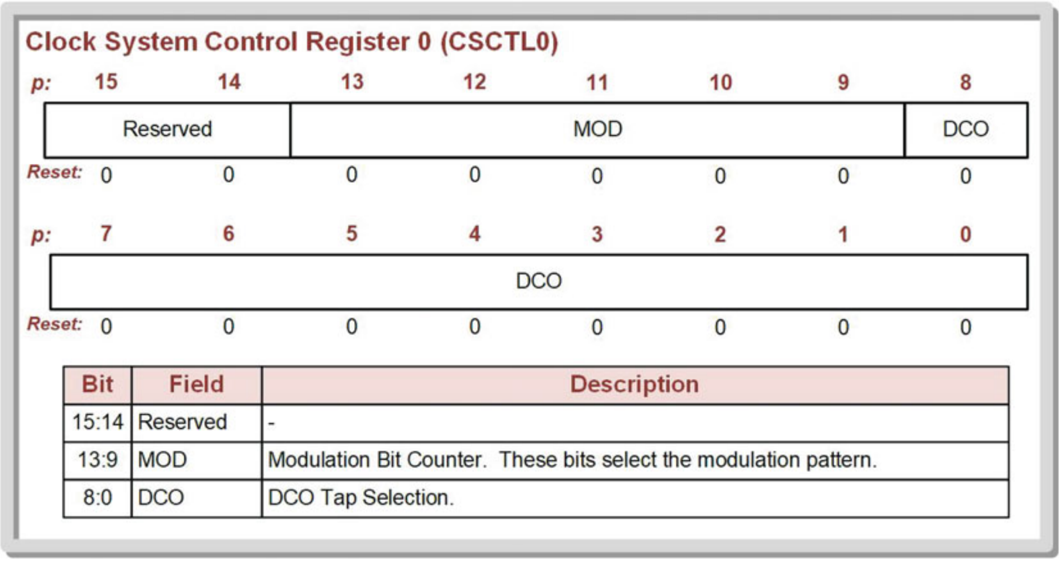
Upon startup, the default frequency for **MCLK is 1 MHz** and is sourced from the DCO. This is a result of the default configuration for the SELMS multiplexer choosing DCOCLKDIV to drive MCLK and the DIVM divider defaulting to divide by 1. The default settings for the DCO puts the DCOCLK into the 2 MHz range with a FLLREF clock source of REFO (after XT1 is determined to be disabled because the port function select bits default to digital I/O mode), a FLLD setting of divide by 2, and a FLLN divider of (31 + 1). Upon startup, the default frequency for **SMCLK is 1 MHz** and is sourced from MCLK. This is a result of the MCLK default frequency being 1 MHz (sourced from the DCO) and the DIVS divider defaulting to divide-by-1. Upon startup, the default frequency for **ACLK is 32.768 kHz** and is sourced from REFO.

16.1.10 CS Configuration Registers

The complete list of CS registers on the MSP430FR2355 MCU is as follows:

- *Clock system control register 0 (CSCTL0).*
- *Clock system control register 1 (CSCTL1).*
- *Clock system control register 2 (CSCTL2).*
- *Clock system control register 3 (CSCTL3).*
- *Clock system control register 4 (CSCTL4).*
- *Clock system control register 5 (CSCTL5).*
- *Clock system control register 6 (CSCTL6).*
- *Clock system control register 7 (CSCTL7).*
- *Clock system control register 8 (CSCTL8).*

All configuration settings for the CS are given in Figs. 16.6, 16.7, 16.8, 16.9, 16.10, 16.11, 16.12, 16.13, and 16.14.



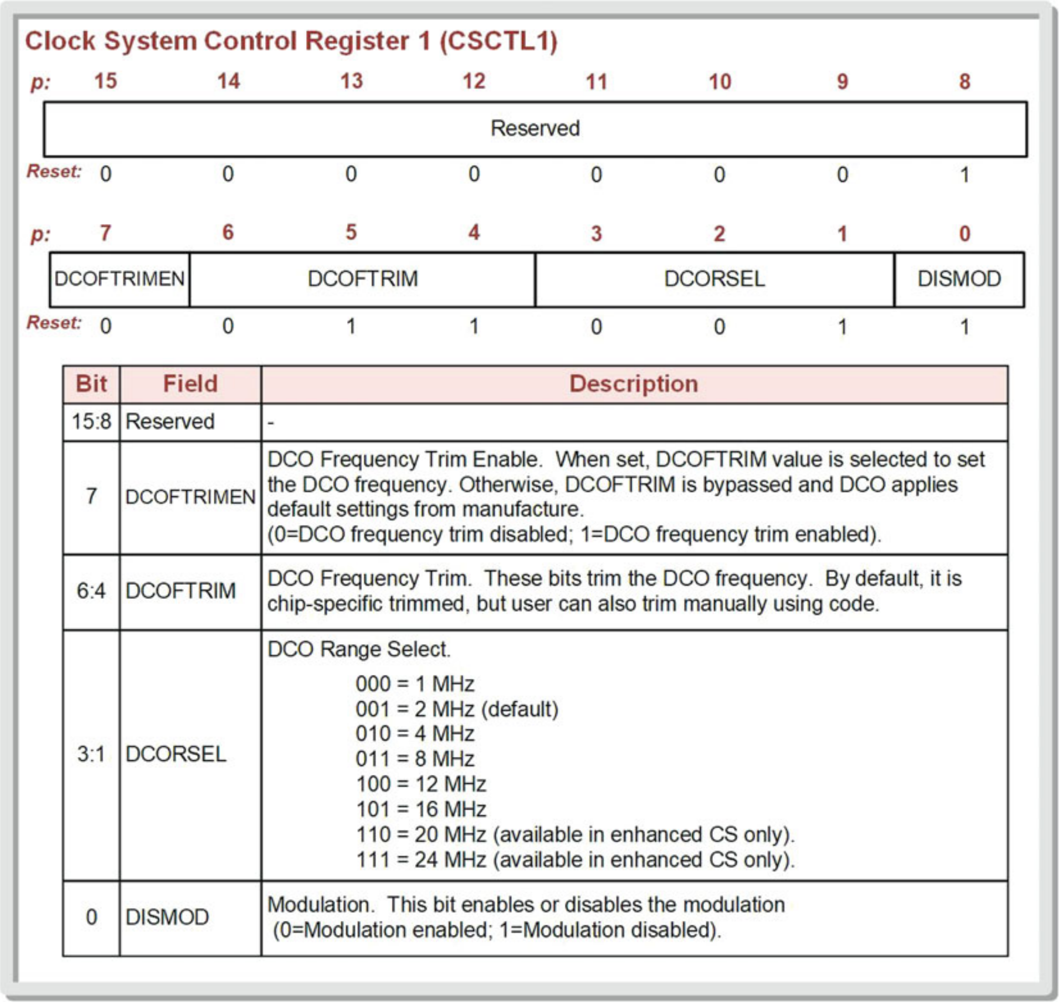
Clock System Control Register 0 (CSCTL0)

p:	15	14	13	12	11	10	9	8
	Reserved		MOD					DCO
Reset:	0	0	0	0	0	0	0	0

p:	7	6	5	4	3	2	1	0
	DCO							
Reset:	0	0	0	0	0	0	0	0

Bit	Field	Description
15:14	Reserved	-
13:9	MOD	Modulation Bit Counter. These bits select the modulation pattern.
8:0	DCO	DCO Tap Selection.

Fig. 16.6
Clock system control register 0 (CSCTL0)



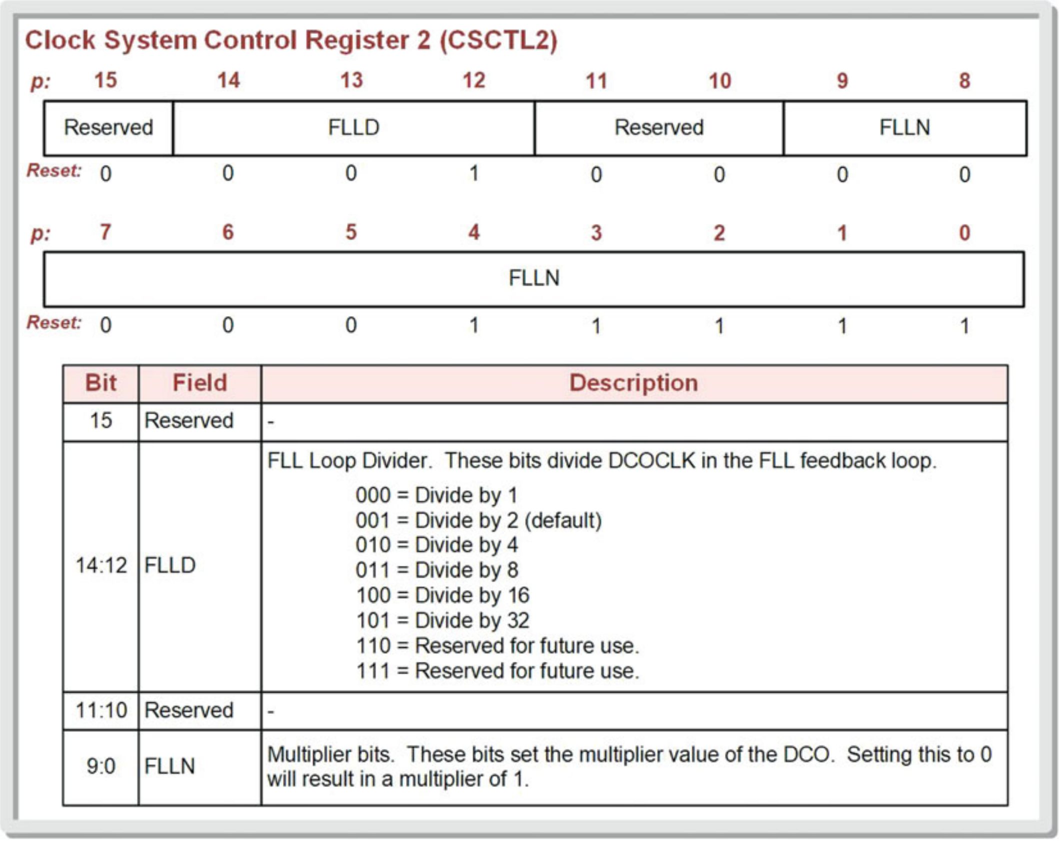
Clock System Control Register 1 (CSCTL1)

p:	15	14	13	12	11	10	9	8
	Reserved							
Reset:	0	0	0	0	0	0	0	1

p:	7	6	5	4	3	2	1	0
	DCOFTRIMEN	DCOFTRIM			DCORSEL			DISMOD
Reset:	0	0	1	1	0	0	1	1

Bit	Field	Description
15:8	Reserved	-
7	DCOFTRIMEN	DCO Frequency Trim Enable. When set, DCOFTRIM value is selected to set the DCO frequency. Otherwise, DCOFTRIM is bypassed and DCO applies default settings from manufacture. (0=DCO frequency trim disabled; 1=DCO frequency trim enabled).
6:4	DCOFTRIM	DCO Frequency Trim. These bits trim the DCO frequency. By default, it is chip-specific trimmed, but user can also trim manually using code.
3:1	DCORSEL	DCO Range Select. 000 = 1 MHz 001 = 2 MHz (default) 010 = 4 MHz 011 = 8 MHz 100 = 12 MHz 101 = 16 MHz 110 = 20 MHz (available in enhanced CS only). 111 = 24 MHz (available in enhanced CS only).
0	DISMOD	Modulation. This bit enables or disables the modulation (0=Modulation enabled; 1=Modulation disabled).

Fig. 16.7
Clock system control register 1 (CSCTL1)



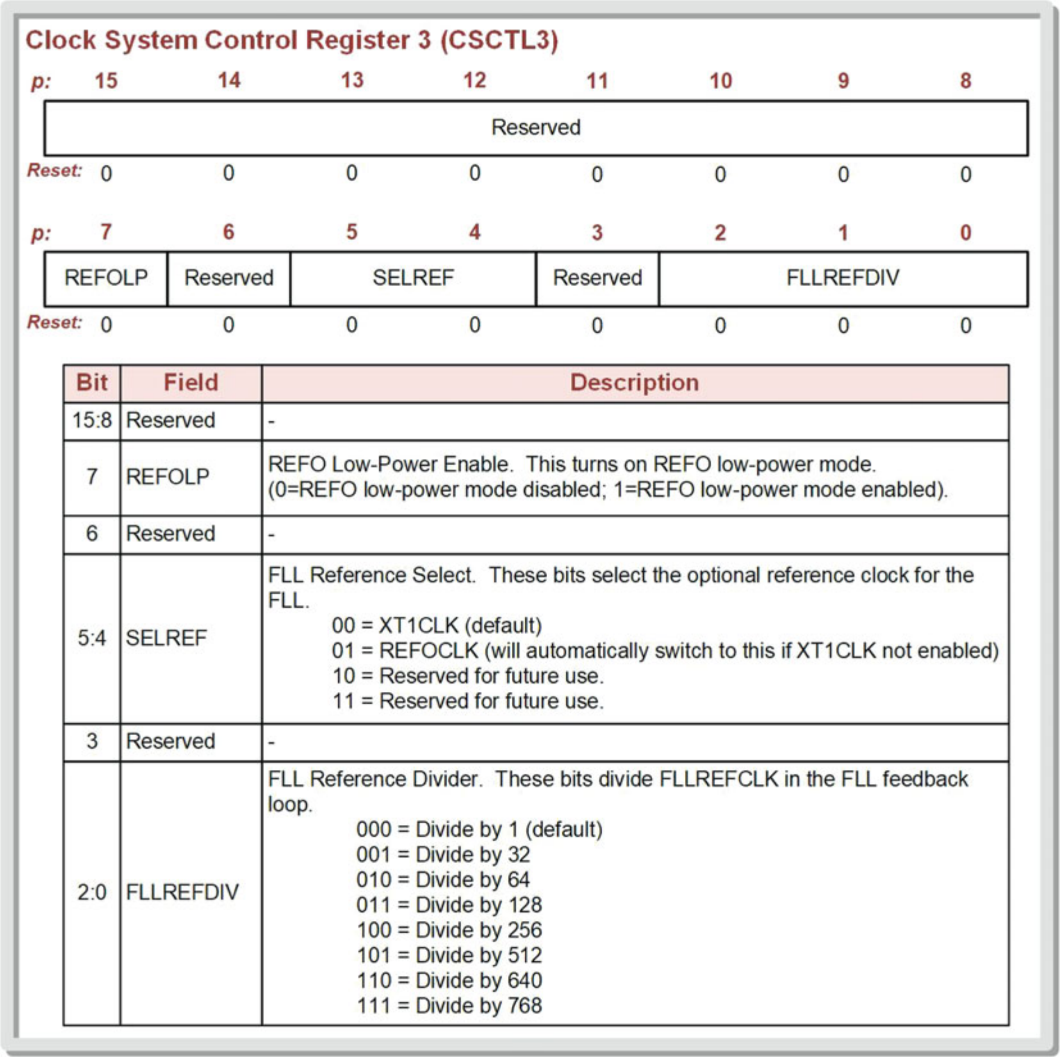
Clock System Control Register 2 (CSCTL2)

p:	15	14	13	12	11	10	9	8
	Reserved	FLLD			Reserved		FLLN	
Reset:	0	0	0	1	0	0	0	0

p:	7	6	5	4	3	2	1	0
	FLLN							
Reset:	0	0	0	1	1	1	1	1

Bit	Field	Description
15	Reserved	-
14:12	FLLD	FLL Loop Divider. These bits divide DCOCLK in the FLL feedback loop. 000 = Divide by 1 001 = Divide by 2 (default) 010 = Divide by 4 011 = Divide by 8 100 = Divide by 16 101 = Divide by 32 110 = Reserved for future use. 111 = Reserved for future use.
11:10	Reserved	-
9:0	FLLN	Multiplier bits. These bits set the multiplier value of the DCO. Setting this to 0 will result in a multiplier of 1.

Fig. 16.8
Clock system control register 2 (CSCTL2)



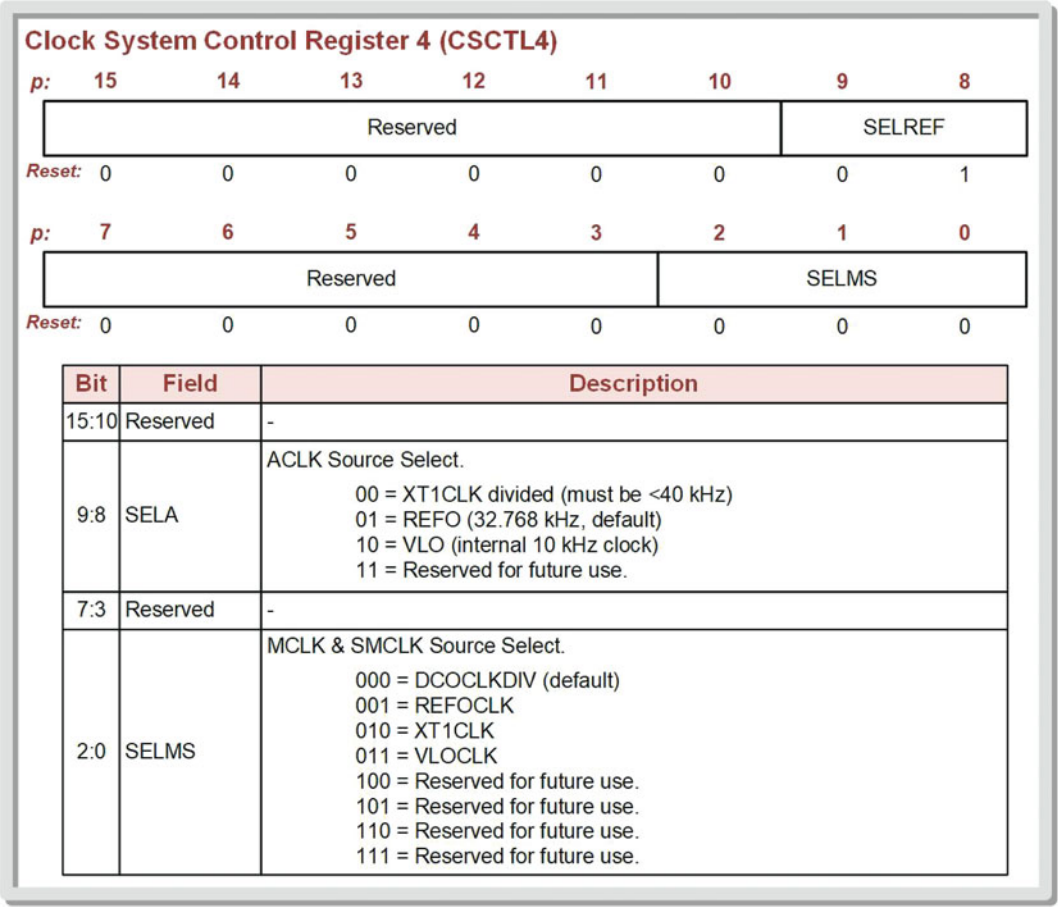
Clock System Control Register 3 (CSCTL3)

p:	15	14	13	12	11	10	9	8
	Reserved							
Reset:	0	0	0	0	0	0	0	0

p:	7	6	5	4	3	2	1	0
	REFOLP	Reserved	SELREF		Reserved	FLLREFDIV		
Reset:	0	0	0	0	0	0	0	0

Bit	Field	Description
15:8	Reserved	-
7	REFOLP	REFO Low-Power Enable. This turns on REFO low-power mode. (0=REFO low-power mode disabled; 1=REFO low-power mode enabled).
6	Reserved	-
5:4	SELREF	FLL Reference Select. These bits select the optional reference clock for the FLL. 00 = XT1CLK (default) 01 = REFOCLK (will automatically switch to this if XT1CLK not enabled) 10 = Reserved for future use. 11 = Reserved for future use.
3	Reserved	-
2:0	FLLREFDIV	FLL Reference Divider. These bits divide FLLREFCLK in the FLL feedback loop. 000 = Divide by 1 (default) 001 = Divide by 32 010 = Divide by 64 011 = Divide by 128 100 = Divide by 256 101 = Divide by 512 110 = Divide by 640 111 = Divide by 768

Fig. 16.9
Clock system control register 3 (CSCTL3)



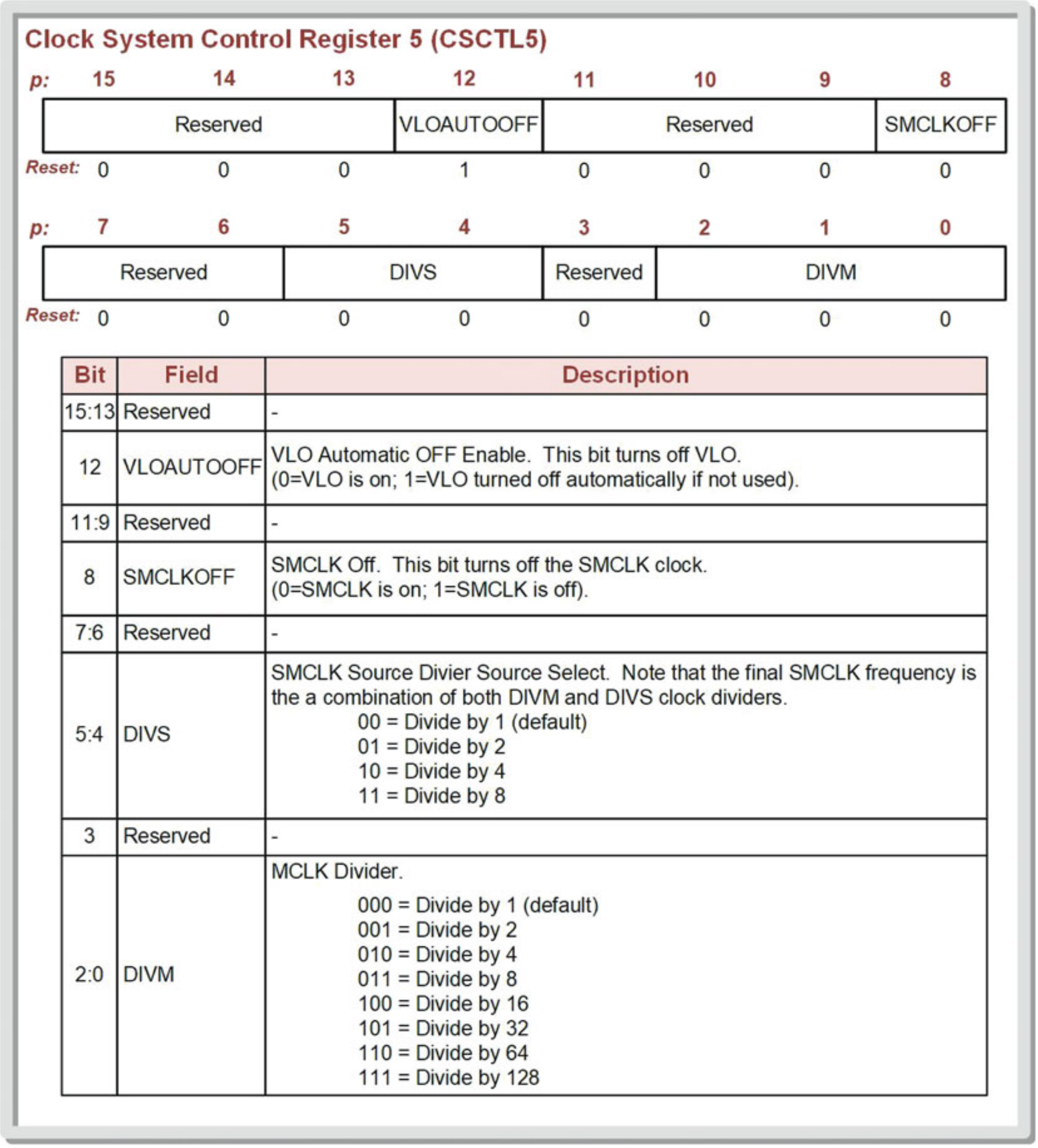
Clock System Control Register 4 (CSCTL4)

p:	15	14	13	12	11	10	9	8
	Reserved						SELREF	
Reset:	0	0	0	0	0	0	0	1

p:	7	6	5	4	3	2	1	0
	Reserved					SELMS		
Reset:	0	0	0	0	0	0	0	0

Bit	Field	Description
15:10	Reserved	-
9:8	SELA	ACLK Source Select. 00 = XT1CLK divided (must be <40 kHz) 01 = REFO (32.768 kHz, default) 10 = VLO (internal 10 kHz clock) 11 = Reserved for future use.
7:3	Reserved	-
2:0	SELMS	MCLK & SMCLK Source Select. 000 = DCOCLKDIV (default) 001 = REFOCLK 010 = XT1CLK 011 = VLOCLK 100 = Reserved for future use. 101 = Reserved for future use. 110 = Reserved for future use. 111 = Reserved for future use.

Fig. 16.10
Clock system control register 4 (CSCTL4)



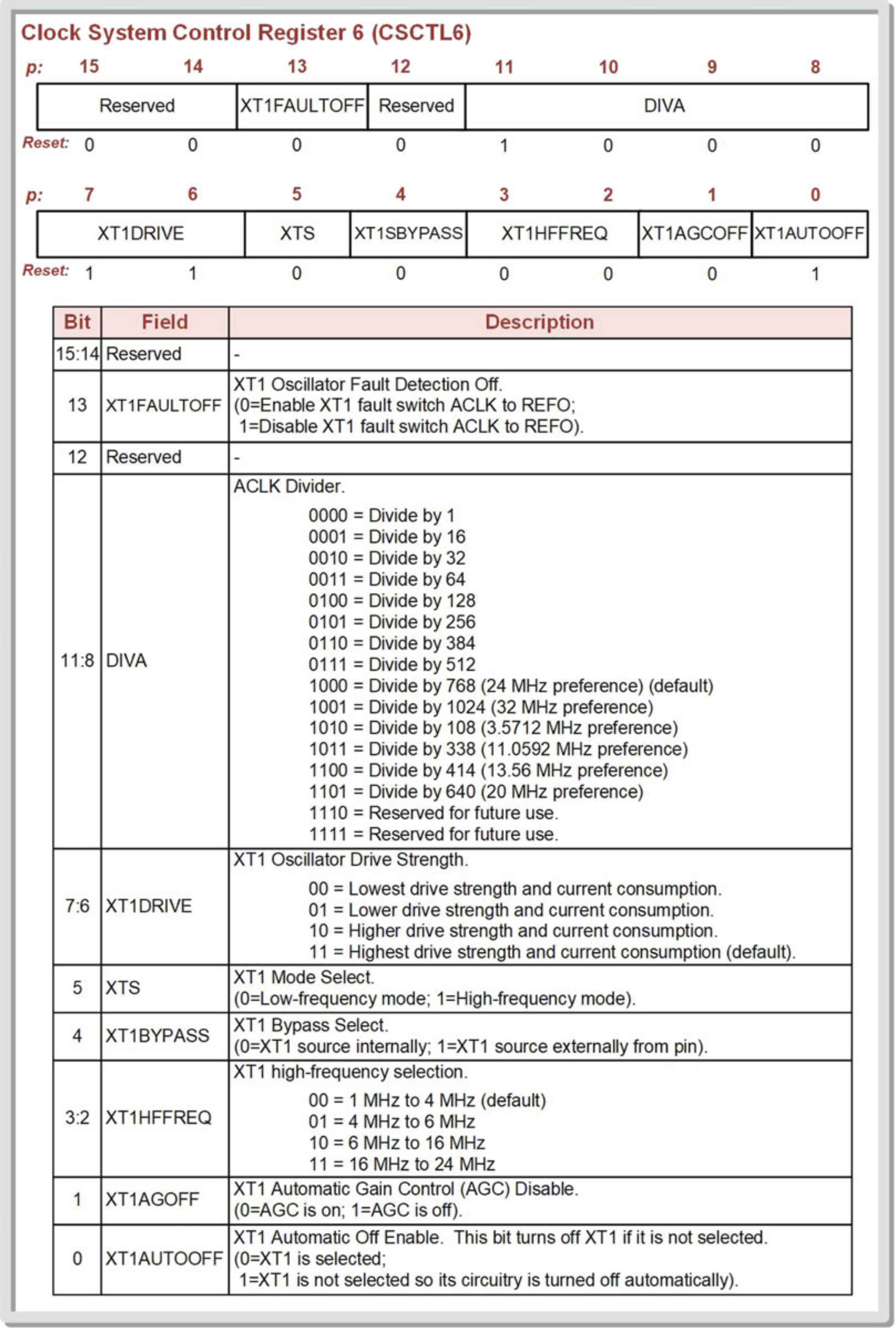
Clock System Control Register 5 (CSCTL5)

p:	15	14	13	12	11	10	9	8
	Reserved			VLOAUTOOFF	Reserved			SMCLKOFF
Reset:	0	0	0	1	0	0	0	0

p:	7	6	5	4	3	2	1	0
	Reserved		DIVS		Reserved	DIVM		
Reset:	0	0	0	0	0	0	0	0

Bit	Field	Description
15:13	Reserved	-
12	VLOAUTOOFF	VLO Automatic OFF Enable. This bit turns off VLO. (0=VLO is on; 1=VLO turned off automatically if not used).
11:9	Reserved	-
8	SMCLKOFF	SMCLK Off. This bit turns off the SMCLK clock. (0=SMCLK is on; 1=SMCLK is off).
7:6	Reserved	-
5:4	DIVS	SMCLK Source Divier Source Select. Note that the final SMCLK frequency is the a combination of both DIVM and DIVS clock dividers. 00 = Divide by 1 (default) 01 = Divide by 2 10 = Divide by 4 11 = Divide by 8
3	Reserved	-
2:0	DIVM	MCLK Divider. 000 = Divide by 1 (default) 001 = Divide by 2 010 = Divide by 4 011 = Divide by 8 100 = Divide by 16 101 = Divide by 32 110 = Divide by 64 111 = Divide by 128

Fig. 16.11
Clock system control register 5 (CSCTL5)



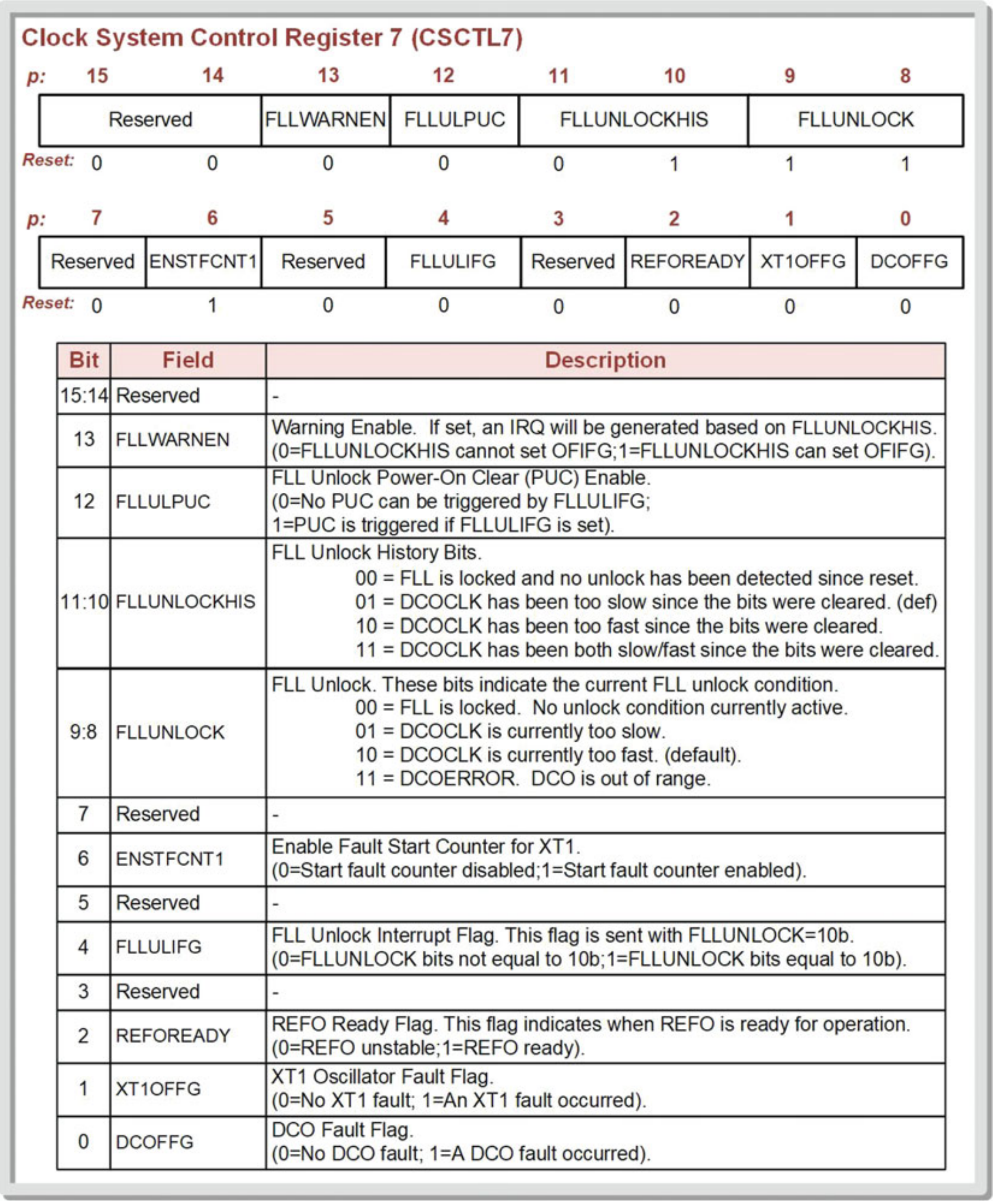
Clock System Control Register 6 (CSCTL6)

p:	15	14	13	12	11	10	9	8
	Reserved		XT1FAULTOFF	Reserved	DIVA			
Reset:	0	0	0	0	1	0	0	0

p:	7	6	5	4	3	2	1	0
	XT1DRIVE		XTS	XT1SBYPASS	XT1HFFREQ		XT1AGCOFF	XT1AUTOOFF
Reset:	1	1	0	0	0	0	0	1

Bit	Field	Description
15:14	Reserved	-
13	XT1FAULTOFF	XT1 Oscillator Fault Detection Off. (0=Enable XT1 fault switch ACLK to REFO; 1=Disable XT1 fault switch ACLK to REFO).
12	Reserved	-
11:8	DIVA	ACLK Divider. 0000 = Divide by 1 0001 = Divide by 16 0010 = Divide by 32 0011 = Divide by 64 0100 = Divide by 128 0101 = Divide by 256 0110 = Divide by 384 0111 = Divide by 512 1000 = Divide by 768 (24 MHz preference) (default) 1001 = Divide by 1024 (32 MHz preference) 1010 = Divide by 108 (3.5712 MHz preference) 1011 = Divide by 338 (11.0592 MHz preference) 1100 = Divide by 414 (13.56 MHz preference) 1101 = Divide by 640 (20 MHz preference) 1110 = Reserved for future use. 1111 = Reserved for future use.
7:6	XT1DRIVE	XT1 Oscillator Drive Strength. 00 = Lowest drive strength and current consumption. 01 = Lower drive strength and current consumption. 10 = Higher drive strength and current consumption. 11 = Highest drive strength and current consumption (default).
5	XTS	XT1 Mode Select. (0=Low-frequency mode; 1=High-frequency mode).
4	XT1BYPASS	XT1 Bypass Select. (0=XT1 source internally; 1=XT1 source externally from pin).
3:2	XT1HFFREQ	XT1 high-frequency selection. 00 = 1 MHz to 4 MHz (default) 01 = 4 MHz to 6 MHz 10 = 6 MHz to 16 MHz 11 = 16 MHz to 24 MHz
1	XT1AGOFF	XT1 Automatic Gain Control (AGC) Disable. (0=AGC is on; 1=AGC is off).
0	XT1AUTOOFF	XT1 Automatic Off Enable. This bit turns off XT1 if it is not selected. (0=XT1 is selected; 1=XT1 is not selected so its circuitry is turned off automatically).

Fig. 16.12
Clock system control register 6 (CSCTL6)



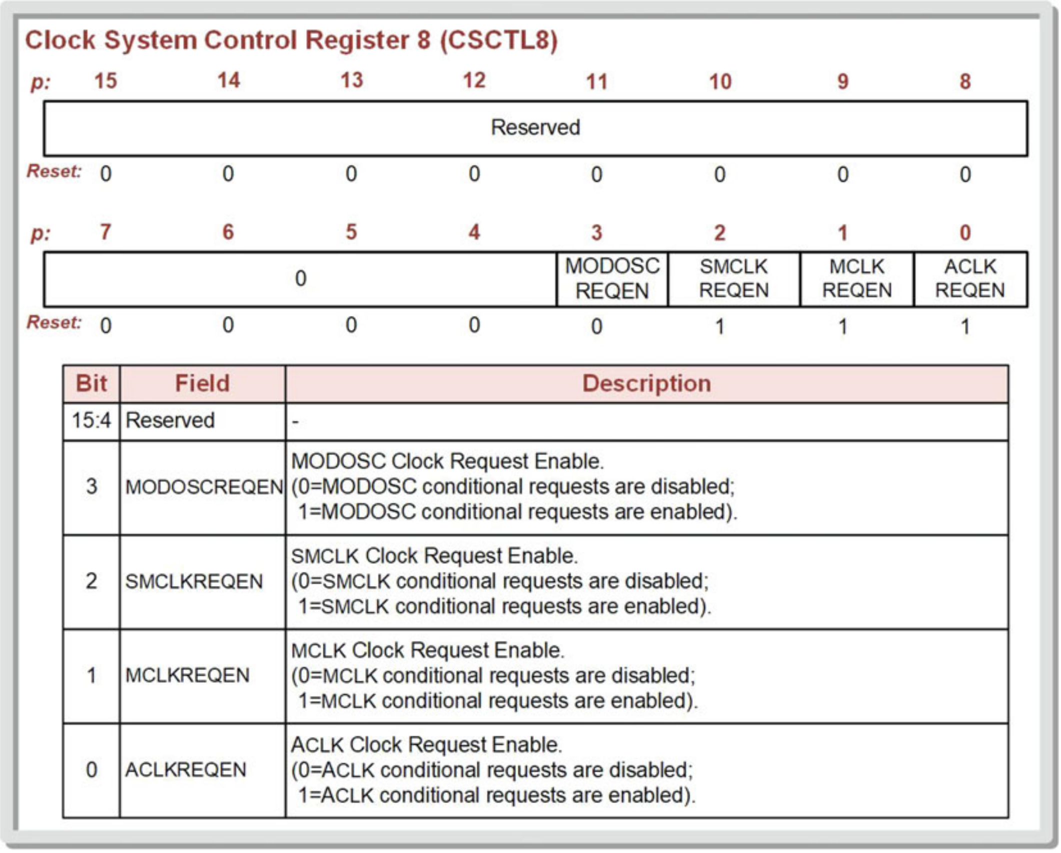
Clock System Control Register 7 (CSCTL7)

p:	15	14	13	12	11	10	9	8
	Reserved		FLLWARNEN	FLLULPUC	FLLUNLOCKHIS		FLLUNLOCK	
Reset:	0	0	0	0	0	1	1	1

p:	7	6	5	4	3	2	1	0
	Reserved	ENSTFCNT1	Reserved	FLLULIFG	Reserved	REFOREADY	XT1OFFG	DCOFFG
Reset:	0	1	0	0	0	0	0	0

Bit	Field	Description
15:14	Reserved	-
13	FLLWARNEN	Warning Enable. If set, an IRQ will be generated based on FLLUNLOCKHIS. (0=FLLUNLOCKHIS cannot set OFIFG;1=FLLUNLOCKHIS can set OFIFG).
12	FLLULPUC	FLL Unlock Power-On Clear (PUC) Enable. (0=No PUC can be triggered by FLLULIFG; 1=PUC is triggered if FLLULIFG is set).
11:10	FLLUNLOCKHIS	FLL Unlock History Bits. 00 = FLL is locked and no unlock has been detected since reset. 01 = DCOCLK has been too slow since the bits were cleared. (def) 10 = DCOCLK has been too fast since the bits were cleared. 11 = DCOCLK has been both slow/fast since the bits were cleared.
9:8	FLLUNLOCK	FLL Unlock. These bits indicate the current FLL unlock condition. 00 = FLL is locked. No unlock condition currently active. 01 = DCOCLK is currently too slow. 10 = DCOCLK is currently too fast. (default). 11 = DCOERROR. DCO is out of range.
7	Reserved	-
6	ENSTFCNT1	Enable Fault Start Counter for XT1. (0=Start fault counter disabled;1=Start fault counter enabled).
5	Reserved	-
4	FLLULIFG	FLL Unlock Interrupt Flag. This flag is sent with FLLUNLOCK=10b. (0=FLLUNLOCK bits not equal to 10b;1=FLLUNLOCK bits equal to 10b).
3	Reserved	-
2	REFOREADY	REFO Ready Flag. This flag indicates when REFO is ready for operation. (0=REFO unstable;1=REFO ready).
1	XT1OFFG	XT1 Oscillator Fault Flag. (0=No XT1 fault; 1=An XT1 fault occurred).
0	DCOFFG	DCO Fault Flag. (0=No DCO fault; 1=A DCO fault occurred).

Fig. 16.13
Clock system control register 7 (CSCTL7)



Clock System Control Register 8 (CSCTL8)

p:	15	14	13	12	11	10	9	8
	Reserved							
Reset:	0	0	0	0	0	0	0	0

p:	7	6	5	4	3	2	1	0
	0				MODOSC REQEN	SMCLK REQEN	MCLK REQEN	ACLK REQEN
Reset:	0	0	0	0	0	1	1	1

Bit	Field	Description
15:4	Reserved	-
3	MODOSCREQEN	MODOSC Clock Request Enable. (0=MODOSC conditional requests are disabled; 1=MODOSC conditional requests are enabled).
2	SMCLKREQEN	SMCLK Clock Request Enable. (0=SMCLK conditional requests are disabled; 1=SMCLK conditional requests are enabled).
1	MCLKREQEN	MCLK Clock Request Enable. (0=MCLK conditional requests are disabled; 1=MCLK conditional requests are enabled).
0	ACLKREQEN	ACLK Clock Request Enable. (0=ACLK conditional requests are disabled; 1=ACLK conditional requests are enabled).

Fig. 16.14
Clock system control register 8 (CSCTL8)

Concept Check

CC16.1 Why not just run the entire MCU off a single clock?

A) Peripherals, especially timers, often need to run at much slower clock rates than the CPU in order to generate/track timing events that are measured in seconds, minutes, and hours.

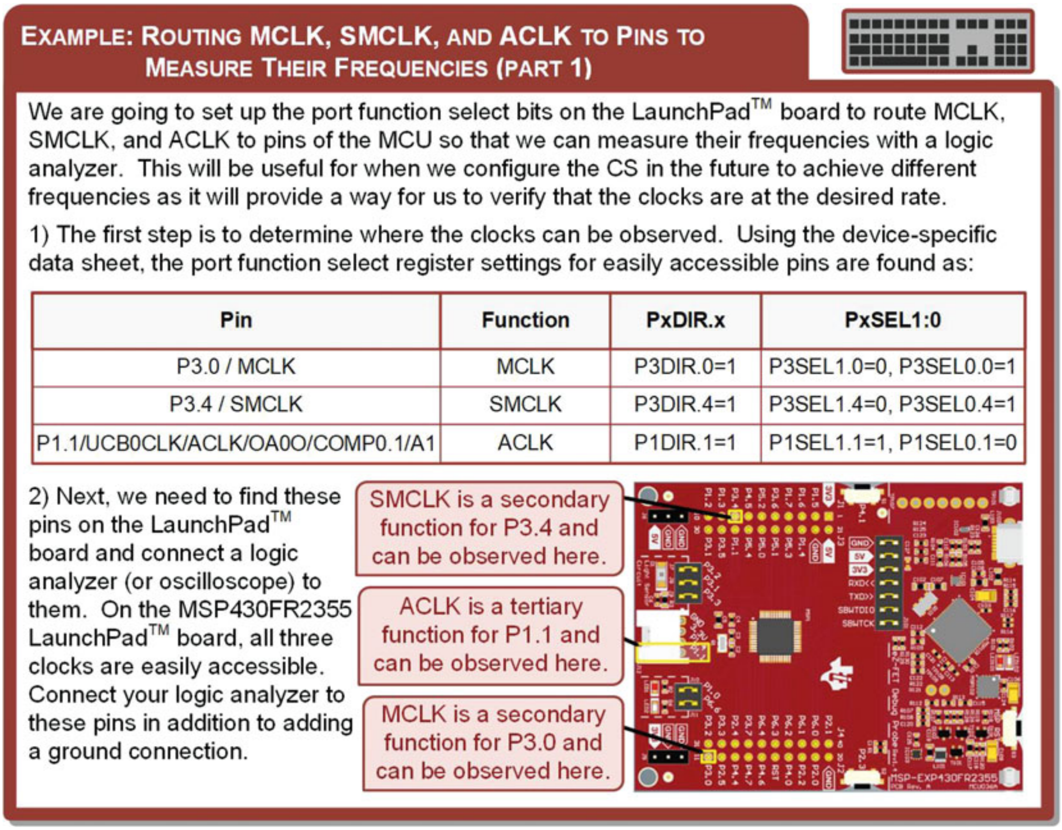
B) Using different clocks allows portions of the MCU to be turned off to save power.

C) An MCU can be used in many different applications including some that require high performance and some that require low power. Having clocks that can run at both high and low rates allows the same MCU to meet the demand of many applications.

D) Add of the above.

16.2 Configuring the CS on the MSP430FR2355

This section gives a few examples of how to configure the CS system on the MSP430FR2355 LaunchPad™. Let' s begin by observing the 3× main clocks to measure their frequencies. Examples 16.1 and 16.2 show how to route MCLK, SMCLK, and ACLK to port pins of the MCU for observation with a logic analyzer to measure their frequencies.



EXAMPLE: ROUTING MCLK, SMCLK, AND ACLK TO PINS TO MEASURE THEIR FREQUENCIES (PART 1)

We are going to set up the port function select bits on the LaunchPad™ board to route MCLK, SMCLK, and ACLK to pins of the MCU so that we can measure their frequencies with a logic analyzer. This will be useful for when we configure the CS in the future to achieve different frequencies as it will provide a way for us to verify that the clocks are at the desired rate.

1) The first step is to determine where the clocks can be observed. Using the device-specific data sheet, the port function select register settings for easily accessible pins are found as:

Pin	Function	PxDIR.x	PxSEL1:0
P3.0 / MCLK	MCLK	P3DIR.0=1	P3SEL1.0=0, P3SEL0.0=1
P3.4 / SMCLK	SMCLK	P3DIR.4=1	P3SEL1.4=0, P3SEL0.4=1
P1.1/UCB0CLK/ACLK/OA0O/COMP0.1/A1	ACLK	P1DIR.1=1	P1SEL1.1=1, P1SEL0.1=0

2) Next, we need to find these pins on the LaunchPad™ board and connect a logic analyzer (or oscilloscope) to them. On the MSP430FR2355 LaunchPad™ board, all three clocks are easily accessible. Connect your logic analyzer to these pins in addition to adding a ground connection.

SMCLK is a secondary function for P3.4 and can be observed here.

ACLK is a tertiary function for P1.1 and can be observed here.

MCLK is a secondary function for P3.0 and can be observed here.

Example 16.1
Routing MCLK, SMCLK, and ACLK to pins to measure their frequencies (part 1)

Example: Routing MCLK, SMCLK, and ACLK to Pins to Measure Their Frequencies (part 2)

3) In CCS, create a new C/C++ Empty Project (with main.c) titled: **C_CS_Clk_Observation**.

4) Type in the following code in main.c after the statement to stop the watchdog timer.

```
#include <msp430.h>

  //-- Setup General I/O
    WDTCTL = WDTPW | WDTHOLD;    // stop watchdog timer
    PM5CTL0 &= ~LOCKLPM5;        // Disable the LOW POWER MODE

  //-- Route MCLK to pin P3.0
    P3DIR  |= BIT0;              // Make P3.0 an OUTPUT
    P3SEL1 &= ~BIT0;             // Switch P3.0 to MCLK (P3SEL1:0.0 = 01b)
    P3SEL0 |= BIT0;

  //-- Route SMCLK to pin P3.4
    P3DIR  |= BIT4;              // Make P3.4 an OUTPUT
    P3SEL1 &= ~BIT4;             // Switch P3.4 to SMCLK (P3SEL1:0.4 = 01b)
    P3SEL0 |= BIT4;

  //-- Route ACLK to pin P1.1
    P1DIR  |= BIT1;              // Make P1.1 an OUTPUT
    P1SEL1 |= BIT1;              // Switch P1.1 to ACLK (P1SEL1:0.1 = 10b)
    P1SEL0 &= ~BIT1;

    while(1){}                   // loop forever while clocks run

    return 0;
}
```

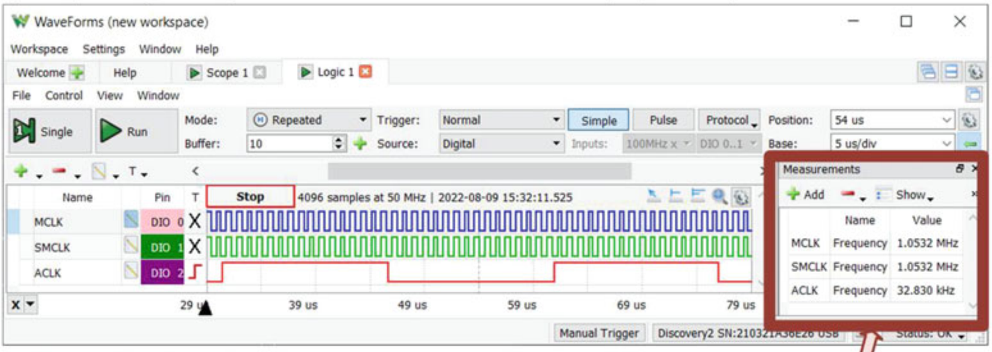
- Route MCLK to P3.0.

- Route SMCLK to P3.4.

- Route ACLK to P1.1.

5) Save, debug, and run your program.

6) Run your logic analyzer and insert measurements to display the frequencies of each clock.


WaveForms (new workspace)
Workspace Settings Window Help
Welcome Help Scope 1 Logic 1
File Control View Window
Single
Run
Stop
Manual Trigger
Status: OK

Mode: Repeated
Trigger: Normal
Simple Pulse Protocol
Position: 54 us
Buffer: 10
Source: Digital
Inputs: 100MHz x DIO 0..1
Base: 5 us/div

4096 samples at 50 MHz | 2022-08-09 15:32:11.525

Name	Pin
MCLK	DIO 0
SMCLK	DIO 1
ACLK	DIO 2

Measurements

	Name	Value
MCLK	Frequency	1.0532 MHz
SMCLK	Frequency	1.0532 MHz
ACLK	Frequency	32.830 kHz

Discovery2 SN:210321A56E26 USB

Did it work? Are you able to see MCLK and SMCLK at their default frequency of ~1 MHz and ACLK at its default frequency of ~32.768 kHz?

Example 16.2
Routing MCLK, SMCLK, and ACLK to pins to measure their frequencies (part 2)

Next, let's see how we can change the clock sources for MCLK and ACLK in addition to changing the divider setting for SMCLK. Example 16.3 shows how to change the source for MCLK to REFO and the source for ACLK to VLO and divide SMCLK by 2.

Example: Changing Sources for MCLK and ACLK and Dividing SMCLK by 2

Let's now change the source for MCLK from DCOCLKDIV to REFO (32.768 kHz) and the source for ACLK from REFO to VLO (10 kHz). Let's also change the divider for SMCLK to divide-by-2 in order to make SMCLK ½ of MCLK (16.384 kHz). We will use the same measurement setup as in the last example where MCLK, SMCLK, and ACLK are routed to pins on the MCU.

1) In CCS, create a new C/C++ Empty Project (with main.c) titled: **C_CS_Changing_Clk_Sources**.

2) Type in the following code in main.c after the statement to stop the watchdog timer.

```
#include <msp430.h>
  //-- Setup General I/O
    WDTCTL = WDTPW | WDTHOLD;     // stop watchdog timer
    PM5CTL0 &= ~LOCKLPM5;         // Disable the LOW POWER MODE
  //-- Route MCLK to pin P3.0
    P3DIR  |= BIT0;               // Make P3.0 an OUTPUT
    P3SEL1 &= ~BIT0;              // Switch P3.0 to MCLK (P3SEL1:0.0 = 01b)
    P3SEL0 |= BIT0;
  //-- Route SMCLK to pin P3.4
    P3DIR  |= BIT4;               // Make P3.4 an OUTPUT
    P3SEL1 &= ~BIT4;              // Switch P3.4 to SMCLK (P3SEL1:0.4 = 01b)
    P3SEL0 |= BIT4;
  //-- Route ACLK to pin P1.1
    P1DIR  |= BIT1;               // Make P1.1 an OUTPUT
    P1SEL1 |= BIT1;               // Switch P1.1 to ACLK (P1SEL1:0.1 = 10b)
    P1SEL0 &= ~BIT1;
  //-- Switch MCLK Source to REFO (32.768 kHz)
    CSCTL4 |= SELMS__REFOCLK;     // SELMS=001b=REFOCLK
  //-- Change SMCLK Divider to divide-by-2 (16.384 kHz)
    CSCTL5 |= DIVS__2;            // DIVS=01b=Divide-by-2
  //-- Switch ACLK Source to VLO (10 kHz)
    CSCTL4 &= ~SELA__REFOCLK;     // Default is SELA=01, so 1st clear LSB
    CSCTL4 |= SELA__VLOCLK;       // SELA=10=VLO

    while(1){}                    // loop forever

    return 0;
}
```

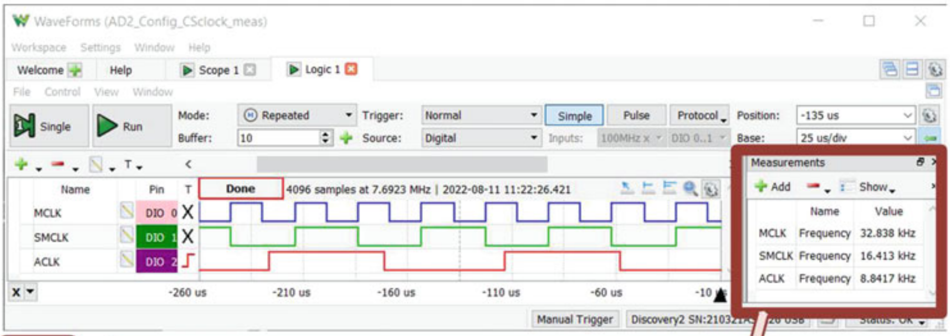
- Change MCLK source to REFO using SELMS.

- Divide SMCLK by 2 using DIVS.

- Change ACLK source to VLO using SELA. Notice that we need to first clear the LSB since SELA's default value is 01b.

3) Save, debug, and run your program.

4) Run your logic analyzer and insert measurements to display the frequencies of each clock.



Mode: Repeated Trigger: Normal Position: -135 us

Buffer: 10 Source: Digital Base: 25 us/div

4096 samples at 7.6923 MHz | 2022-08-11 11:22:26.421

	Name	Value
MCLK	Frequency	32.838 kHz
SMCLK	Frequency	16.413 kHz
ACLK	Frequency	8.8417 kHz

Did it work? Are you able to see MCLK = ~32.768 kHz, SMCLK = ~16.384 kHz, and ACLK = ~10kHz?

Example 16.3
Changing the sources for MCLK and ACLK and dividing SMCLK by 2

Now let's look at how we can change the DCO settings to get a different MCLK frequency. Example 16.4 shows how to set MCLK to 8 MHz using the DCO settings values from Table 16.1. In this example the DCO is put into its 8 MHz range using DCORSEL, FLLD is set to divide-by-1, and FLLN is set to 244. These settings divide the 8 MHz clock down to ~32 kHz for the feedback signal that is fed into the FLL integrator to be stabilized to the FLLREFCLK of 32.768 kHz (REFO).

Example: Changing MCLK to 8 MHz Using the DCO and FLL

Let's change the frequency of MCLK to 8 MHz. We will use the default source for MCLK (DCOCLKDIV) with the DCO in FLL mode using a reference clock of REFO (32.768 kHz). To configure the FLL, we'll put the DCO in its 8 MHz range (DCORSEL), set FLLD=divide-by-1, and set FLLN=244. We'll route MCLK to a pin to measure its frequency with a logic analyzer.

1) In CCS, create a new C/C++ Empty Project (with main.c) titled: **C_CS_Changing_DCO_Freq**.

2) Type in the following code in main.c after the statement to stop the watchdog timer.

```
#include <msp430.h>

  //-- Setup General I/O
    WDTCTL = WDTPW | WDTHOLD;     // stop watchdog timer
    PM5CTL0 &= ~LOCKLPM5;         // Disable the LOW POWER MODE

  //-- Route MCLK to pin P3.0
    P3DIR  |= BIT0;               // Make P3.0 an OUTPUT
    P3SEL1 &= ~BIT0;              // Switch P3.0 to MCLK (P3SEL1:0.0 = 01b)
    P3SEL0 |= BIT0;

  //-- Change MCLK to 8MHz
    CSCTL1 |= DCORSEL_3;          // Change DCO Range to 8 MHz

    CSCTL2 &= ~FLLD_1;            // Change FLLD to divide-by-1

    CSCTL2 &= ~FLLN;              // Set FLLN equal to 244.
    CSCTL2 |= 244;

    while(1){}                    // loop forever

    return 0;
}
```

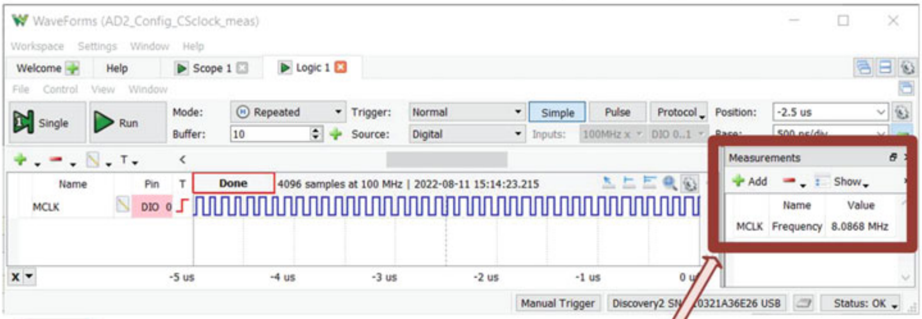
- Change DCORSEL from 001b to 011b to put into its 8 MHz range.

- FLLD defaults to 001b. We want 000b for divide-by-1 so we clear its LSB.

- FLLN defaults to 1Fh. We will first clear the 10x bits of FLLN using a bitwise AND and then set it to 244 using a bitwise OR.

3) Save, debug, and run your program.

4) Run your logic analyzer and insert measurements to display the frequency of MCLK.



Mode: Repeated
Trigger: Normal
Position: -2.5 us
Buffer: 10
Source: Digital
Inputs: 100MHz x DIO 0..1
Base: 500 ns/div

Done — 4096 samples at 100 MHz | 2022-08-11 15:14:23.215

Name	Pin
MCLK	DIO 0

Measurements

	Name	Value
MCLK	Frequency	8.0868 MHz

Did it work? Are you able to see MCLK = ~8 MHz?

Example 16.4
Changing MCLK to 8 MHz using the DCO and FLL

Concept Check

CC16.2 Why is it a good idea to observe the clocks with a logic analyzer or oscilloscope when determining the configuration settings of the CS?

A) It allows you to verify that the settings you are using are giving you the desired clock rates.

Summary

- ❖ The clock system generates the clocks used by the MCU from a variety of on-chip and off-chip oscillators. Many different clock frequencies can be achieved by using different oscillator sources and different configuration settings.
- ❖ The MSP430 has two CS modes: basic and enhanced. The enhanced mode provides more clock source and frequency options. The MSP430FR2355 implements the CS enhanced mode.
- ❖ The CS contains 4× on-chip oscillators: REFO (32.768 kHz), VLO (10 kHz), DCO (1, 2, 4, 8, 12, 16, 20, and 24 MHz), and MODO (5 MHz).
- ❖ The DCO contains a frequency locked loop (FLL) configuration that allows it to stabilize its output to a reference clock in order to produce a stable clock signal for MCLK.
- ❖ The CS supports an off-chip oscillator as the source for XT1. XT1 can be put into LF mode where it is driven with a 32.768 kHz watch crystal or into HF mode where it is driven with a higher frequency oscillator.
- ❖ The CS produces three primary clocks for use by the MCU: master clock (MCLK), subsystem master clock (SMCLK), and auxiliary clock (ACLK).
- ❖ MCLK's primary function is to drive the CPU and memory.
- ❖ SMCLK and ACLK's primary functions are to drive peripherals.
- ❖ MCLK can be sourced from VLO, REFO, DCO, and XT1.
- ❖ ACLK can be sourced from REFO, VLO, and XT1.
- ❖ SMCLK is always sourced from MCLK, but it can be divided down further using its own divider stage.
- ❖ ACLK's nominal frequency is 32.768 kHz, but it can go as low as 10 kHz and as high as 40 kHz.
- ❖ If using XT1 in HF mode for ACLK, the input frequency must be divided down to get it within the 10–40 kHz range allowed for ACLK.

Exercise Problems

Section 16.1: Overview of the MSP430FR2355 Clock System

16.1.1 Does the MSP430FR2355 implement the CS basic mode or enhanced mode?

16.1.2 What is the nominal frequency of REFO?

16.1.3 What is the nominal frequency of VLO?

16.1.4 What is the nominal frequency of MODO?

16.1.5 What are the nominal frequencies supported for MCLK?

16.1.6 What is the nominal frequency of ACLK?

16.1.7 What configuration bits set the source for MCLK?

16.1.8 What configuration bits set the source for ACLK?

16.1.9 If XT1 is unavailable for the FLLREFCLK clock, what source does the DCO default to?

16.1.10 What can MODO be used for?

Section 16.2: Configuring the CS on the MSP430FR2355

16.2.1 Can the primary CS clock outputs be routed to I/O pins on the MCU?

16.2.2 What is an easily accessible pin on the MSP430FR2355 LaunchPad™ board to observe MCLK?

16.2.3 What is an easily accessible pin on the MSP430FR2355 LaunchPad™ board to observe SMCLK?

16.2.4 What is an easily accessible pin on the MSP430FR2355 LaunchPad™ board to observe ACLK?

16.2.5 When a configuration field holds a default value that is not zero, what step must be taken prior to setting a new value?

Chapter 17: Low Power Modes

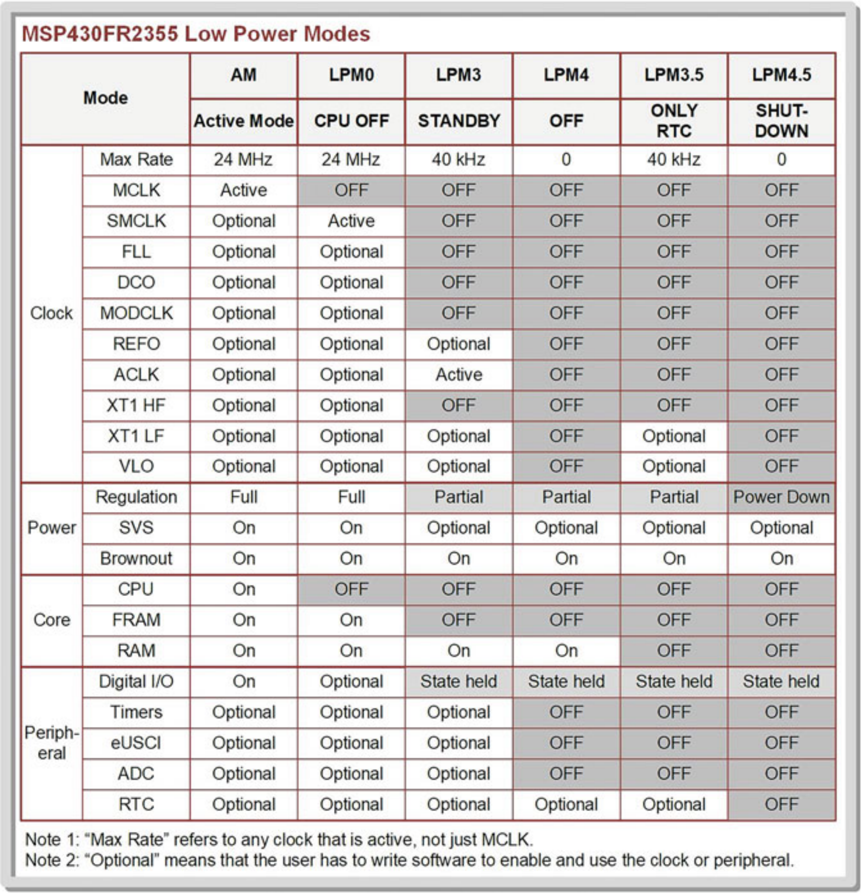
This chapter looks at the low power modes available on the MSP430FR2355 [1–3, 14]. The MSP4302355 contains a variety of low power modes that achieve decreasing levels of power consumption by turning off subsystems within the MCU. This chapter gives an overview of the available low power modes including the control signals used to enter and wake from these modes.

Learning Outcomes—After completing this chapter, you will be able to:

17.1 Describe the basic concept of MSP430FR2355's low power modes.

17.1 Overview of the MSP430FR2355's Low Power Modes

The general concept of low power mode on the MSP430 is to turn off clocks and supply voltages to subsystems within the MCU. The more systems that are turned off, the less power the MCU will use. The MSP430FR2355 contains 5× low power modes that are shown in Table 17.1.



MSP430FR2355 Low Power Modes

Mode		AM	LPM0	LPM3	LPM4	LPM3.5	LPM4.5
		Active Mode	CPU OFF	STANDBY	OFF	ONLY RTC	SHUT-DOWN
Clock	Max Rate	24 MHz	24 MHz	40 kHz	0	40 kHz	0
	MCLK	Active	OFF	OFF	OFF	OFF	OFF
	SMCLK	Optional	Active	OFF	OFF	OFF	OFF
	FLL	Optional	Optional	OFF	OFF	OFF	OFF
	DCO	Optional	Optional	OFF	OFF	OFF	OFF
	MODCLK	Optional	Optional	OFF	OFF	OFF	OFF
	REFO	Optional	Optional	Optional	OFF	OFF	OFF
	ACLK	Optional	Optional	Active	OFF	OFF	OFF
	XT1 HF	Optional	Optional	OFF	OFF	OFF	OFF
	XT1 LF	Optional	Optional	Optional	OFF	Optional	OFF
	VLO	Optional	Optional	Optional	OFF	Optional	OFF
Power	Regulation	Full	Full	Partial	Partial	Partial	Power Down
	SVS	On	On	Optional	Optional	Optional	Optional
	Brownout	On	On	On	On	On	On
Core	CPU	On	OFF	OFF	OFF	OFF	OFF
	FRAM	On	On	OFF	OFF	OFF	OFF
	RAM	On	On	On	On	OFF	OFF
Peripheral	Digital I/O	On	Optional	State held	State held	State held	State held
	Timers	Optional	Optional	Optional	OFF	OFF	OFF
	eUSCI	Optional	Optional	Optional	OFF	OFF	OFF
	ADC	Optional	Optional	Optional	OFF	OFF	OFF
	RTC	Optional	Optional	Optional	Optional	Optional	OFF

Note 1: "Max Rate" refers to any clock that is active, not just MCLK.
Note 2: "Optional" means that the user has to write software to enable and use the clock or peripheral.

Table 17.1
Low power modes on the MSP430FR2355

B. J. LaMeres, *Embedded Systems Design using the MSP430FR2355 LaunchPad™*,
https://doi.org/10.1007/978-3-031-20888-1_17

LPM0, LPM3, and LPM4 are entered using the CPUOFF, OSCOFF, SCG0, and SCG1 bits in the SR. Wake up from these three modes is possible through all enabled interrupts. Keeping these control bits in the SR is advantageous because they are stacked during an ISR. After executing the current ISR, the SR is restored from the stack and the MCU returns to the previously configured low power mode.

Entering LPM3.5 and LPM4.5 involves also disabling the power management module on the MSP430. When in these ultralow power modes, all RAM and register contents are lost; however, the digital I/O pins are locked at their current state. Wake up from LPM3.5 is possible from a power cycle, a reset, an RTC event, an LF crystal fault, or from an external pin NMI. Wake up from LPM4.5 is only possible from a power cycle, a reset, or from an external pin NMI.

17.1.1 Active Mode (AM)

Active mode (AM) on the MSP430FR2355 is when everything in the MCU is enabled and available for full speed operation (up to 24 MHz). Remember that on reset, the digital I/O must be taken out of its default low power inhibit mode. This is accomplished by clearing the LOCKLPM5 bit in the PM5CTL0 register.

17.1.2 Low Power Mode 0 (LPM0): CPU OFF

Low power mode 0 (LPM0) is called the *CPU off* mode. This mode disables MCLK, which disables the CPU but still allows SMCLK to run at full speed (up to 24 MHz) and allows full functionality of the CS. All peripherals can be used in this mode and the FRAM and RAM continue to receive power; thus, no data is lost and wake up is instant. To enter LPM0, the CPUOFF bit is asserted in the SR. Wake up is accomplished through any enabled interrupt request.

17.1.3 Low Power Mode 3 (LPM3): Standby

Low power mode 3 (LPM3) is called *standby* mode. This mode disables MCLK and SMCLK. Any peripherals that are enabled must use ACLK as their source with a maximum frequency of 40 kHz. Digital I/O is held at their last state until active mode is restored. Voltage regulation is partially shutdown for unused peripherals, but the CPU core is still powered. Power is removed from the FRAM, but the RAM is still powered so no data is lost. Some CS circuitry is disabled including the FLL, DCO, XT1HF, and MODCLK in order to save additional power. To enter LPM3, the CPUOFF, SCG0, and SCG1 bits are asserted in the SR. Wake up is accomplished through any enabled interrupt request with a wake-up time of ~10 μs.

17.1.4 Low Power Mode 4 (LPM4): Off

Low power mode 4 (LPM4) is called *off* mode. This mode disables all clocks, thus disabling the CPU core and all peripherals except for the RTC. Digital I/O is held at their last state until active mode is restored. Power is removed from the FRAM, but the RAM is still powered so no data is lost. To enter LPM4, the CPUOFF, OSCOFF, SCG0, and SCG1 bits are asserted in the SR. Wake up is accomplished through either an edge on the NMI pin (if enabled) or an RTC event (if enabled) with a wake-up time of ~10 μs.

17.1.5 Low Power Mode 3.5 (LPM3.5): RTC Only

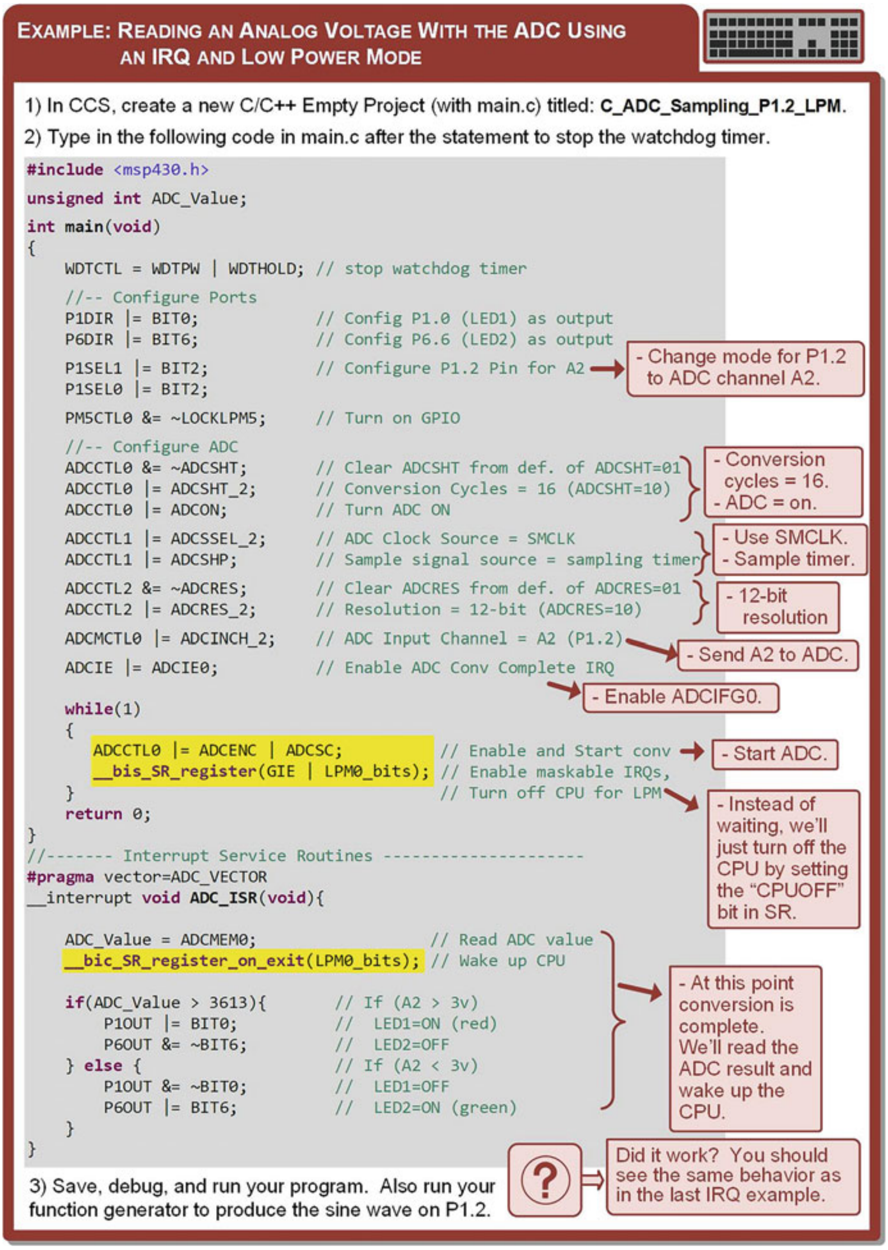
Low power mode 3.5 (LPM3.5) is called *RTC Only* mode. This mode disables all clocks and all peripherals except for the RTC peripheral running off of either VLOCLK or XT1LFCLK (max frequency 40 kHz). All other subsystems are powered down and there is no memory retention. To enter LPM3.5, the CPUOFF, OSCOFF, SCG0, and SCG1 bits are asserted in the SR in addition to disabling the power management module by setting the PMMREGOFF bit in the PMMCTL0 register. If any part of the RTC is enabled, then the MCU knows to go into LPM3.5 and not LPM4.5. Wake up from LPM3.5 causes a full system reset and takes the longest time of any low power mode to wake from (~350 μs). Wake up from LPM3.5 is accomplished through a power cycle, a reset, an LF crystal fault, an edge on an NMI pin (if enabled), or an RTC event.

17.1.6 Low Power Mode 4.5 (LPM4.5): Shutdown

Low power mode 4.5 (LPM4.5) is called *shutdown* mode. This mode is identical to LPM3.5 except no part of the RTC is enabled. All subsystems are powered down, and there is no memory retention. To enter LPM4.5, the CPUOFF, OSCOFF, SCG0, and SCG1 bits are asserted in the SR in addition to disabling the power management module by setting the PMMREGOFF bit in the PMMCTL0 register. If no part of the RTC is enabled, then the MCU knows to go into LPM4.5 and not LPM3.5. Wake up from LPM4.5 causes a full system reset and takes the longest time of any low power mode to wake from (~350 μs). Wake up from LPM4.5 is accomplished through a power cycle, a reset, an LF crystal fault, or an edge on an NMI pin (if enabled).

17.1.7 Example of Putting the MSP430FR2355 into Low Power Mode

In Example 15.3, we setup the ADC to read from an external voltage and used a polling loop in the main program to ensure that the ADC conversion start was only triggered once the prior conversion was complete. A more practical way to accomplish this logic is to instead start the conversion and then put the CPU into one of the low power modes that still allows the ADC to run. Let's look at an example of where instead of polling the ADC conversion complete flag, we start the conversion and then enter LPM0. Once the conversion completes, the ADC will trigger and IRQ that will bring the MCU out of LPM0 and execute the ISR. After the ISR, the program will return to main, start the conversion, and then re-enter LPM0 continuously. We enter LPM0 by asserting the CPUOFF bit in the SR. Example 17.1 shows how to use LPM instead of a polling loop to control when the next conversion is triggered.



EXAMPLE: READING AN ANALOG VOLTAGE WITH THE ADC USING AN IRQ AND LOW POWER MODE

1) In CCS, create a new C/C++ Empty Project (with main.c) titled: **C_ADC_Sampling_P1.2_LPM**.

2) Type in the following code in main.c after the statement to stop the watchdog timer.

```c
#include <msp430.h>
unsigned int ADC_Value;
int main(void)
{
    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer

    //-- Configure Ports
    P1DIR |= BIT0;            // Config P1.0 (LED1) as output
    P6DIR |= BIT6;            // Config P6.6 (LED2) as output

    P1SEL1 |= BIT2;           // Configure P1.2 Pin for A2
    P1SEL0 |= BIT2;

    PM5CTL0 &= ~LOCKLPM5;     // Turn on GPIO

    //-- Configure ADC
    ADCCTL0 &= ~ADCSHT;       // Clear ADCSHT from def. of ADCSHT=01
    ADCCTL0 |= ADCSHT_2;      // Conversion Cycles = 16 (ADCSHT=10)
    ADCCTL0 |= ADCON;         // Turn ADC ON

    ADCCTL1 |= ADCSSEL_2;     // ADC Clock Source = SMCLK
    ADCCTL1 |= ADCSHP;        // Sample signal source = sampling timer

    ADCCTL2 &= ~ADCRES;       // Clear ADCRES from def. of ADCRES=01
    ADCCTL2 |= ADCRES_2;      // Resolution = 12-bit (ADCRES=10)

    ADCMCTL0 |= ADCINCH_2;    // ADC Input Channel = A2 (P1.2)

    ADCIE |= ADCIE0;          // Enable ADC Conv Complete IRQ

    while(1)
    {
        ADCCTL0 |= ADCENC | ADCSC;          // Enable and Start conv
        __bis_SR_register(GIE | LPM0_bits); // Enable maskable IRQs,
    }                                       // Turn off CPU for LPM
    return 0;
}
//------- Interrupt Service Routines --------------------
#pragma vector=ADC_VECTOR
__interrupt void ADC_ISR(void){

    ADC_Value = ADCMEM0;                   // Read ADC value
    __bic_SR_register_on_exit(LPM0_bits); // Wake up CPU

    if(ADC_Value > 3613){         // If (A2 > 3v)
        P1OUT |= BIT0;            //  LED1=ON (red)
        P6OUT &= ~BIT6;           //  LED2=OFF
    } else {                      // If (A2 < 3v)
        P1OUT &= ~BIT0;           //  LED1=OFF
        P6OUT |= BIT6;            //  LED2=ON (green)
    }
}
```

- Change mode for P1.2 to ADC channel A2.

- Conversion cycles = 16.
- ADC = on.

- Use SMCLK.
- Sample timer.

- 12-bit resolution

- Send A2 to ADC.

- Enable ADCIFG0.

- Start ADC.

- Instead of waiting, we'll just turn off the CPU by setting the "CPUOFF" bit in SR.

- At this point conversion is complete. We'll read the ADC result and wake up the CPU.

3) Save, debug, and run your program. Also run your function generator to produce the sine wave on P1.2.

?

Did it work? You should see the same behavior as in the last IRQ example.

Example 17.1
Reading an analog voltage with the ADC using an IRQ and low power mode

Concept Check

CC17.1 LPM4.5 seems excessive. What kind of application would want an MCU to be completely turned off until an external event occurs?

A) A battery powered application where saving power is paramount.

B) An application where events are very infrequency (i.e., days or weeks).

C) An application where no data memory is needed to store information. The MCU just wakes up, runs a stand-alone routine, and then powers down.

D) Add of the above.

Summary

- ❖ The MSP430FR2355 supports a range of low power modes that give a decreasing levels of power consumption.
- ❖ The more aggressive the low power mode, the more items are turned off. This leads to a longer wakeup time and fewer options for how it is awoken.
- ❖ LPM0 is called *CPU off* mode and simply turns off MCLK to the CPU. Wakeup is instant.
- ❖ LPM3 is called *Standby* mode and turns off both MCLK and SMCLK. Any peripherals that are used must run off of ACLK with a max frequency of 40 kHz. Wakeup is ~10 μs.
- ❖ LPM4 is called *Off* mode and turns off everything except power to the RAM and the RTC. Wakeup is ~10 μs.
- ❖ LPM3.5 is called *RTC only* mode and turns off all clocks and powers down everything except the RTC. Wakeup is ~350 μs.
- ❖ LPM4.5 is called *Shutdown* mode and turns off all clocks and powers down everything. Wakeup is ~350 μs.

Exercise Problems

Section 17.1: Overview of the MSP430FR2355's Low Power Modes

17.1.1 How do you enter LPM0?

17.1.2 What is turned off during LPM0?

17.1.3 How do you wake from LPM0?

17.1.4 How long does it take to wake from LPM0?

17.1.5 How do you enter LPM3?

17.1.6 What is turned off during LPM3?

17.1.7 How do you wake from LPM3?

17.1.8 How long does it take to wake from LPM3?

17.1.9 How do you enter LPM4?

17.1.10 What is turned off during LPM4?

17.1.11 How do you wake from LPM4?

17.1.12 How long does it take to wake from LPM4?

17.1.13 How do you enter LPM3.5?

17.1.14 What is turned off during LPM3.5?

17.1.15 How do you wake from LPM3.5?

17.1.16 How long does it take to wake from LPM3.5?

17.1.17 How do you enter LPM4.5?

17.1.18 What is turned off during LPM4.5?

17.1.19 How do you wake from LPM4.5?

17.1.20 How long does it take to wake from LPM4.5?

Appendix A: Concept Check Solutions

CC 1.1	A
CC 2.1.1	C
CC 2.1.2	D
CC 2.1.3	D
CC 2.1.4	A
CC 2.2.1	A
CC 2.2.2	C
CC 2.2.3(a)	D
CC 2.2.3(b)	B
CC 2.2.4(a)	C
CC 2.2.4(b)	C
CC 2.2.4(c)	B
CC 2.3.1	B
CC 2.3.2	A
CC 2.4.1	D
CC 2.4.2(a)	C
CC 2.4.2(b)	B
CC 2.4.3	A
CC 3.1	A
CC 3.2	A
CC 3.3	D
CC 4.1	A
CC 4.2	D
CC 4.3	D
CC 5.1	B
CC 5.2	D
CC 5.3	C
CC 6.1	A
CC 6.2	B
CC 6.3	A
CC 6.4	B
CC 6.5	B
CC 6.6	A
CC 6.7	A
CC 7.1	B
CC 7.2	B
CC 7.3	A
CC 7.4	C
CC 7.5	A
CC 8.1	A
CC 8.2	A
CC 8.3	A
CC 8.4	D
CC 9.1	B
CC 9.2	A
CC 9.3	B
CC 10.1	A
CC 10.2	B
CC 11.1	C
CC 11.2	C
CC 12.1	D
CC 12.2	B
CC 12.3	A
CC 12.4	A
CC 12.5	D
CC 13.1	A
CC 13.2	D
CC 13.3	A
CC 14.1	A
CC 14.2	B
CC 14.3	A
CC 15.1	A
CC 15.2	D
CC 16.1	D
CC 16.2	A
CC 17.1	D

B. J. LaMeres, *Embedded Systems Design using the MSP430FR2355 LaunchPad™*,
https://doi.org/10.1007/978-3-031-20888-1

References

1. Texas Instruments Inc (2019) MSP430FR4xx and MSP430FR2xx Family User's Guide, Literature Number: SLAU445I, October 2014 – Revised March 2019. Retrieved from: https://www.ti.com/lit/ug/slau445i/slau445i.pdf
2. Texas Instruments Inc (2019) MSP430FR235x, MSP430FR215x Mixed-Signal Microcontrollers [device-specific data sheet], Literature Number: SLASEC4D, May 2018 – Revised December 2019. Retrieved from: https://www.ti.com/lit/ds/symlink/msp430fr2355.pdf
3. Texas Instruments Inc (2019) MSP430FR235x LaunchPad™ Development Kit User's Guide, Literature Number: SLAU680, May 2018. Retrieved from: https://www.ti.com/lit/ug/slau680/slau680.pdf
4. Marwedel P (2018) Embedded system design - Embedded systems foundations of cyber-physical systems, and the internet of things, 3rd edn. Springer International Publishing, Cham. https://doi.org/10.1007/978-3-319-56045-8
5. Jimenez M, Palomera R, Couvertier I (2014) Introduction to embedded systems using microcontrollers and the MSP430. Springer Science+Business Media, New York. https://doi.org/10.1007/978-1-4614-3143-5
6. Nagy C (2003) Embedded systems design using the TI MSP430 Series. Elsevier Science & Technology, Burlington. https://doi.org/10.1007/978-0-750-67623-6
7. Roh R (2018) Q4 2018 PC shipments dip amid trade tensions, CPU shortages, Windows Central, January 2019. Retrieved from: https://www.windowscentral.com/q4-pc-shipments-dip-amid-trade-tensions-cpu-shortages
8. Lineback R (2018) MCUs sales to reach record-high annual revenues through 2022, IC Insights – Research Bulletin, September 13, 2018. Retrieved from: http://www.icinsights.com/data/articles/documents/1101.pdf
9. Oscilloscope Front Panel Image is licensed under Creative Commons BY-SA. Retrieved from: https://commons.wikimedia.org/wiki/File:Oscilloscope_Front_Panel.svg
10. LaMeres B (2019) Introduction to logic circuits & logic design with VHDL. Springer Nature, Cham. https://doi.org/10.1007/978-3-030-12489-2
11. Texas Instruments Inc (2018) MSP430 Optimizing C/C++ Compiler v18.12.0.LTS User's Guide, Literature Number: SLAU132T, October 2004. Revised December 2018. Retrieved from: http://www.ti.com/lit/ug/slau132t/slau132t.pdf
12. Texas Instruments Inc (2015) Understanding the I2C Bus [Application Report], Literature Number: SLVA704, June 2015. Retrieved from: http://www.ti.com/lit/an/slva704/slva704.pdf
13. NXP Semiconductors Inc (2015) PCF8523 Real-Time Clock (RTC) and calendar [Product data sheet], Rev. 7, Literature Number: PCF8523, April 2015. Retrieved from: https://www.nxp.com/docs/en/data-sheet/PCF8523.pdf
14. Texas Instruments Inc (2020) MSP430FR2xx/FR4xx DCO+FLL Applications Guide [Application Report], Literature Number: SLAA992, December 2020. Retrieved from: https://www.ti.com/lit/an/slaa992/slaa992.pdf

B. J. LaMeres, *Embedded Systems Design using the MSP430FR2355 LaunchPad™*,
https://doi.org/10.1007/978-3-031-20888-1

Index

B.J. LaMeres, *Embedded Systems Design using the MSP430FR2355 LaunchPad*™,
https://doi.org/10.1007/978-3-031-20888-1