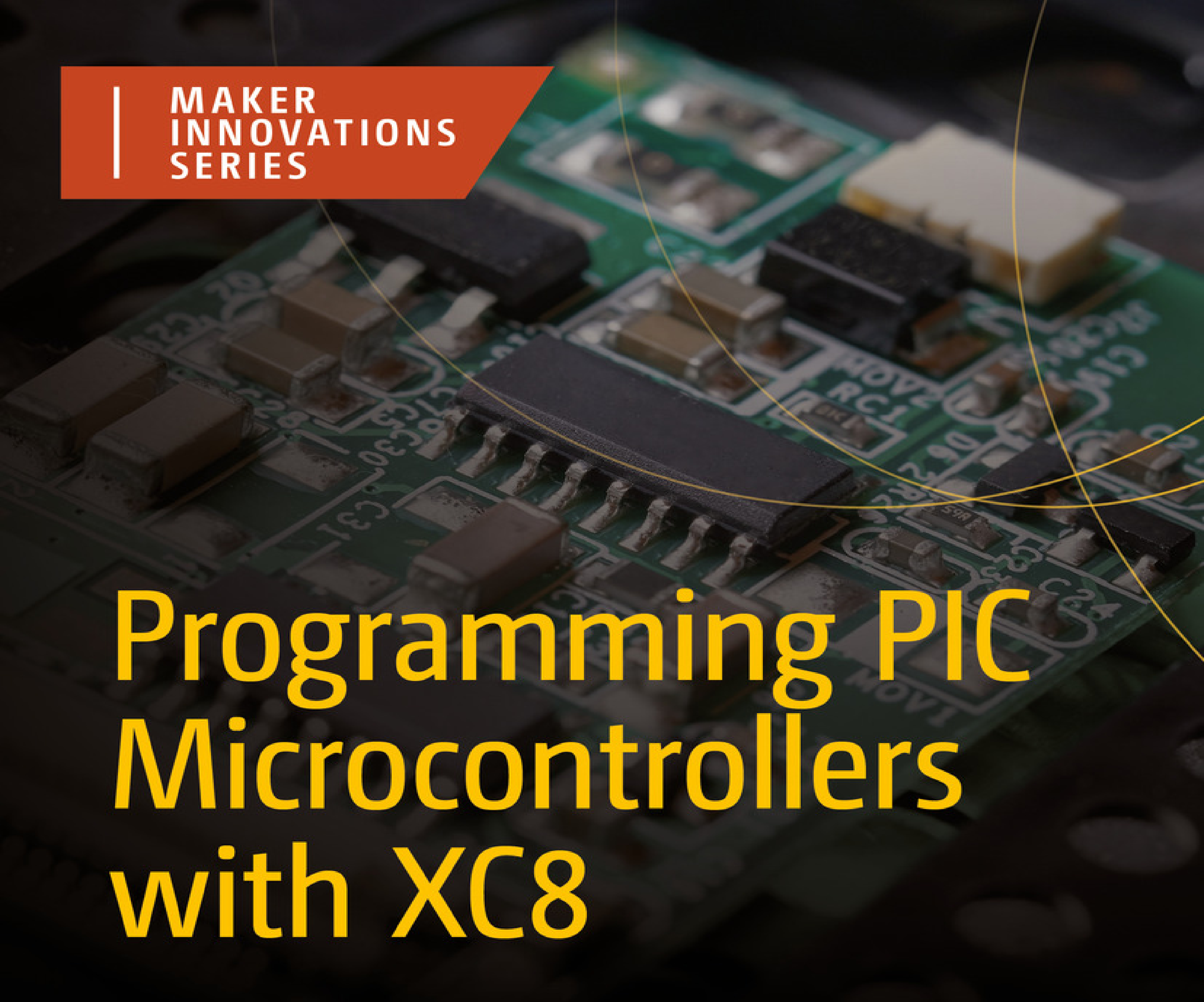


MAKER INNOVATIONS SERIES

Programming PIC Microcontrollers with XC8

Mastering Classical Embedded Design

Second Edition

Armstrong Subero

Apress®

Armstrong Subero

Programming PIC Microcontrollers with XC8

Mastering Classical Embedded Design

2nd ed.

Apress®

Armstrong Subero
Moruga, Trinidad and Tobago

ISSN 2948-2542 e-ISSN 2948-2550
Maker Innovations Series
ISBN 979-8-8688-0466-3 e-ISBN 979-8-8688-0467-0
https://doi.org/10.1007/979-8-8688-0467-0

This Apress imprint is published by the registered company APress Media, LLC, part of Springer Nature.
The registered company address is: 1 New York Plaza, New York, NY 10004, U.S.A.

Preface

The world of microcontrollers has changed. When I first started with microcontroller devices, information was scarce, and the devices the average developer in a small business or a maker had access to were limited. Tools were expensive, and C compilers were a privilege. The majority of the code was written in Assembly.

Microcontrollers are now more accessible than ever. When I first came up with the idea for the first edition of the book, the Arduino platform was still primarily based around 8-bit microcontrollers, the Rust programming language was still relatively new, and Microchip Technology had just completed its acquisition of Atmel. Since that time, a lot has changed. Long gone are the days of Assembly-based programming for microcontrollers; the majority of the members of the PIC architecture are now designed with the C programming language in mind. Core independent peripherals, which operate independently of the CPU, are now no longer a luxury but a core defining feature of PIC microcontrollers. If we look at other brands of microcontrollers, the majority are based around ARM cores, some with multiple cores on board and integrated wireless functionality. The speeds and processing power of some of these devices are approaching what would have been only available to microprocessors at the time the first edition of this book was published as there are now microcontrollers that run at 1 GHz!

Despite 32-bit microcontrollers slowly eroding the market share of 8-bit microcontrollers, in my view for the foreseeable future, there will always be a need for 8-bit devices not only for legacy and historical reasons, but the vast majority of embedded devices still require low power consumption and low cost and provide more functionality with less board space and less time to develop a product. 8-bit PIC microcontrollers excel at working in tandem with these powerful 32-bit devices. Things like intelligent sensors and co-processing are where 8-bit PIC microcontrollers shine. Another application use case is Application-Specific Integrated Circuits (ASICs) replacements and extremely low power design. Modern PIC microcontrollers aren’t your grandfather's PIC microcontroller devices with just a few timers and limited interrupts; the peripherals on these new devices cater to mixed-signal design and can help a lot in almost any application. Some 8-bit PIC members not only have board comparators and analog-to-digital converters (ADCs) but also op amps, digital-to-analog converters (DACs), and configurable logic cells so you can replace quite a few ASICs with a single device.

Of course, they aren't just limited to replacing ASICs or assisting 32-bit microcontrollers. The PIC microcontrollers are still well suited for developing stand-alone embedded systems. This is especially true in the modern scape of embedded systems where devices are not only expected to have some sort of cloud connectivity, but it is required by the vast majority of new designs. Not only must these devices be able to push data to the cloud but they must also do so with strict budget and power requirements. The Internet of Things (IoT) is still the dominant use case for embedded devices, but a paradigm shift has taken place. Being able to connect to the Internet is great, but doing so securely is where attention is given. Words like embedded security and functional safety are where the industry focus is going forward.

It is my hope with this book that you develop the skills to successfully use PIC microcontrollers in your designs. 8-bitters aren't dead. The number of new functionalities that are being added to PIC microcontrollers is staggering. Microchip Technology just keeps on innovating, adding new peripherals and features. I never thought I would see the day when Direct Memory Access (DMA) will be available on 8-bit microcontroller devices, but that day is here! I hope that this book can guide you to creating efficient embedded devices quickly and easily.

Preface to the First Edition

With the onset of the Internet of Things (IoT) revolution, embedded systems development is becoming very popular in the maker community and the professional space as well. IoT is a trillion-dollar business. PIC microcontrollers are one of the technologies that can be used to develop IoT devices. This is due to the low cost, wide availability, and low power consumption of these devices. Additionally, due to the wide range of PIC microcontrollers available, there are PIC microcontrollers that can match your designs, from 8 pins to over 144 pins. They cover 8-, 16-, and 32-bit architectures.

People argue that 8-bit architecture is irrelevant in the complex embedded systems of today. However, 8-bit microcontrollers are here to stay, even if it is for the simple purpose of learning about microcontroller architecture. The relatively simple and beautifully engineered architecture of 8-bit PIC microcontrollers makes them invaluable for learning the inner workings of microcontrollers. It is a lot easier to learn all the registers of these simple 8-bit devices and follow the path of program execution than with more complex ones. After learning about PIC microcontrollers, I found it easy to move on to the more popular 16-bit and then 32-bit devices. In this book, I hope to share the tips and tricks I learned along the way.

Why Did I Write This Book?

The first edition of my book *Programming PIC Microcontrollers with XC8* was fairly successful. I have received (and still receive) emails from my readers asking for assistance, giving feedback, and making suggestions as to what they would have liked to learn about and which areas needed expanding. I took notes and took the critical feedback I received and used it to improve this book. I wrote this book with one goal in mind: to help you master embedded design with PIC microcontrollers.

When I first started programming PIC microcontrollers, I imagined that a lot of information would be available; now there is information available, but a lot of it is dated and uses devices from 20 years ago! So I thought we would change that and provide a solid guide to getting you started with the XC8 compiler from Microchip Technology. Even the underlying architecture of the XC8 compiler has changed as the faithful and decades-old MPASM has been replaced by the PIC-AS assembler. These changes have had a large impact on development time and quality. The XC8 compiler when used with MPLAB X can make your development a breeze.

Additionally, Microchip Technology provides the MPLAB Code Configurator (MCC) that can generate code to use the onboard peripherals of a lot of PIC microcontrollers. This provides a nice Hardware Abstraction Layer (HAL). However, as with all code generation tools, it is not perfect. Sometimes it is necessary to tweak the generated code. I can recall when the dsPIC33CH dual-core devices first came out, I hurriedly bought some of the low pin count SOIC devices from the family as soon as they were in stock on Mouser. When they arrived, eager to learn the new architecture, I used the code generator, and my device didn’t run. On examining the generated code and reading the datasheet, I realized a few included and configuration bits were not set correctly. All it took was one look at the device datasheet, and within the hour, I was able to get the device running.

Without that skill set, you will have to wait for the next available software patch (if one ever becomes available) or are forced to use another device. Some developers like myself prefer to write their bare metal libraries; when you write your libraries from the ground up, you learn the device architecture and quirks and can work around errata that sometimes are not accounted for by code generators. Additionally, you can make your code compact and efficient, and having the power to browse a datasheet and configure a peripheral is a skill set that will make you a cut above the rest. Not only will you be able to write efficient and reliable software, but you will be able to understand almost any code base you come across with ease.

In a professional setting, you will almost always have to write some bare metal software at some point, so it was my hope when writing this book to give you the skills to be able to build your own embedded devices with PIC microcontrollers. The skills learned here can be scaled to 16-bit and 32-bit devices.

Whom Is This Book For?

This book is for anyone interested in embedded design. For this book, you will need some basic electronic devices and some electronic equipment and knowledge of how to use them. At a minimum, you should have access to a multimeter. An oscilloscope is strongly recommended, and a function generator is a nice plus. I expect that the reader knows the C programming language. Knowledge of variables, conditionals, loops, and basic data structures will suffice. I also assume you know basic analog and digital electronics. I also make the presumption that you have used another simpler platform, such as Arduino, or even an embedded Linux computer like the Raspberry Pi since the focus of this book is on the specifics of the PIC microcontroller. A complete newcomer can follow along, but this book is a blend of theory, code, and schematics and assumes an underlying familiarity with microcontrollers.

What You Will Need for This Book?

You will need a few components, software, and hardware tools to get all the examples up and running. All of these are covered in Chapter 1. I know of individuals who build microcontroller circuits in simulation. I strongly recommend against this. I have received numerous emails about code that won't work only to discover after troubleshooting that the reader was using simulation software. Sometimes code will run in a software simulation but would not work on a real device.

I highly recommend building the actual circuits to gain hands-on experience that will help you in the industry. Some features aren't emulated properly in a simulated device, and to get a true development experience, you need to build physical projects. Unlike other programming disciplines, embedded systems development allows you to build things that can be used in our physical world, not just push pixels around the screen. I have also found it more enjoyable to prototype circuits, as you also learn valuable skills in circuit design and troubleshooting that you will have for a lifetime. Although for many people using a development board is simpler, for those wanting a true "hands-on" approach to learning, prototyping on breadboards is a valuable skill.

What Will You Learn in This Book?

This book consists of 14 chapters that will help you get on your way to programming PIC microcontrollers in XC8. Each chapter is compact and filled with information.

- Chapter 1 looks at setting up shop, including the hardware and software necessary to get the most out of this book.
- Chapter 2 covers the basics of the C programming language and intermediate features.
- Chapter 3 reviews the basics of electronics for embedded systems.
- Chapter 4 presents the basics of PIC microcontrollers and zooms in on the PIC16F1719.
- Chapter 5 covers I/O and interfacing LEDs, seven-segment displays, and pushbuttons.
- Chapter 6 presents interrupts, timers, counters, and PWM.
- Chapter 7 covers interfacing actuators.
- Chapter 8 examines serial communications, interfacing displays, and GSM and GPS module interfacing.
- Chapter 9 looks at more display interfacing including OLEDs, touch LCDs, and touch interfaces.
- Chapter 10 consists of understanding the ADC and DAC.
- Chapter 11 is an in-depth look at core independent peripherals including CLC, NCO, comparator, and Fixed Voltage Reference.
- Chapter 12 covers wireless connectivity with Wi-Fi and Bluetooth.
- Chapter 13 demonstrates the use of the low power features of the microcontrollers including minimizing power consumption and the watchdog timer.
- Chapter 14 takes us deep into embedded design as we look at the design of two projects. One is a classical temperature-controlled fan, and the other is a touch screen clock.

Along the way, we'll learn about embedded system architectures, PID controllers, various sensors, general embedded systems, event scheduling, and so much more. Upon finishing this book, I hope that you will have the foundation you need to take on the world of embedded systems design and build useful gadgets, instruments, IoT devices, and beyond. This is the book I wish I had when I was getting started with PIC microcontrollers.

Table of Contents

About the Author

Armstrong Subero

has been tinkering with electronics for as long as he can remember. The thrill of creating something from the ground up and watching it work is something that he never tires of. His entire life changed when he discovered microcontrollers. They were so powerful and simple and complex all at the same time. When he finished school, he taught himself programming and, for a while, worked part-time from a home office. He landed his first job as a systems technologist completely self-taught, and a lot of it was due to his in-depth knowledge and passion for microcontroller technology. Armstrong has used many microcontroller families during his work, but he has an affinity for PIC microcontrollers. He has several published books including *Programming PIC Microcontrollers with XC8*, First Edition; *Codeless Data Structures and Algorithms*; and *Programming Microcontrollers with Python*. Armstrong currently works for the Ministry of National Security in his country.

A. Subero, *Programming PIC Microcontrollers with XC8*, Maker Innovations Series
https://doi.org/10.1007/979-8-8688-0467-0_1

1. Preparing for Development

Armstrong Subero[1]

(1) Moruga, Trinidad and Tobago

It would be nice to be able to jump right into building projects and using our microcontroller. However, before we do so, we need to properly set up our environment for working. This chapter is catered to people who have used microcontroller platforms such as Arduino, PICAXE, or BASIC Stamp-based platforms and want to build bare-bones microcontroller systems. Beginners should have no trouble following along though. If you have experience breadboarding circuits using ICSP tools or have previously used PIC microcontrollers, you may skip this chapter. However, I *strongly* recommend that you read this chapter, as it provides a lot of insight as to what you need as well as getting everything prepared.

Gathering Your Hardware

This is the first chapter on your journey to embedded systems design with PIC microcontrollers and XC8. The first thing we will do is gather the necessary components you will need to follow along with this book. Long gone are the days when a couple of thousands of dollars would be needed to begin microcontroller development. For relatively little money, you can experiment with microcontroller development. This is especially true of PIC microcontrollers, where for a few pennies, you can purchase one of these ubiquitous beasts.

People familiar with programming place emphasis on writing programs, while people with a background in electronics place emphasis on building the circuits for the controllers. I have found that both are equally important, and as you follow along with this book, remember that not everything can be solved using software. If you correctly learn how the hardware operates, you could potentially write very little code that combines hardware in unique ways to get the desired result.

Let's jump into it and look at the things you will need.

Microcontroller

Although the book generally assumes that you have some experience with microcontrollers, this section reviews the basic microcontroller technology. Read this section thoroughly if you're a first-time user of microcontrollers. The information you learn in this section will not only apply to PIC microcontrollers but also other microcontrollers you may use.

General-purpose computers such as smartphones, tablets, laptops, and desktops are designed to perform a variety of tasks. A laptop or tablet can be used to read books, watch movies, and even write programs and web applications. This is because they were designed for that purpose, thanks to the integration of the microprocessors into these units that allow them to perform these many different tasks.

The microprocessor, however, is not an island. It is dependent on supporting circuitry to work properly. These include RAM chips, SSD, and other peripherals. While it is revolutionary, the strength of the microprocessor is also its shortcoming. Although it can perform general tasks, it may not be the best solution for performing a single task.

Let's take the example of an electric toothbrush. If we want to design a basic electric toothbrush, then some basic considerations must go into its function. The toothbrush must turn on a motor when the user pushes a button and alert the user if they have been brushing their teeth too long. In such an instance, a minimum of processing power is needed to adequately perform this task. Yes, it is possible to program a board that contains a 4 GHz 64-bit processor with 16GB of RAM running the latest OS to do this task, but that would be akin to using a lawn mower to shave your legs. It would be better for many reasons to use a microcontroller.

So what exactly is a microcontroller? A *microcontroller* is a self-contained unit that has a microprocessor with RAM, ROM, I/O, and a host of other peripherals onboard. Thus, a microcontroller contains all the processing power necessary to perform the specific task at hand and that task alone. Back to the toothbrush example, it would be more feasible to use a 4-bit microcontroller with a few bytes of RAM and ROM to check the switch, turn on the motor, keep track of how long the user has been brushing, and sound an alarm if that time exceeds some preset value.

Microcontrollers are used for applications that have specific requirements such as low cost, low power consumption, and systems that require real-time performance. It is thanks to these features that a world where computers are becoming increasingly ubiquitous is now possible.

At the time of writing, there are 4-, 8-, 16-, and 32-bit microcontrollers. Anyone looking to start a new design should realistically choose an 8-bit or 32-bit microcontroller. Large-volume, low-cost, and lowest power consumption 8-bit devices generally tend to have an edge, whereas for higher-performance applications, 32-bit devices are the obvious choice. You mustn't get attached to one particular microcontroller. Some people insist that they can do anything with 8 bits, whereas others only use 32-bit parts. You must realize that microcontrollers are simply tools applied to a particular task, so it stands to reason that some tasks are better suited to 8-bit microcontrollers and others to 32-bit ones.

The microcontroller we use in this book is the 8-bit PIC16F1719 (see Figure 1-1). The PIC microcontroller was chosen because it has a relatively simple architecture. Once you understand 8-bit PIC microcontrollers, it's easy to understand more complex micros. I chose this particular PIC microcontroller because it is a modern device and has a lot of onboard peripherals. It also has a relatively large amount of RAM and program memory and, most importantly, a lot of onboard peripherals. There are members of its family with the same features that have a smaller pin count.

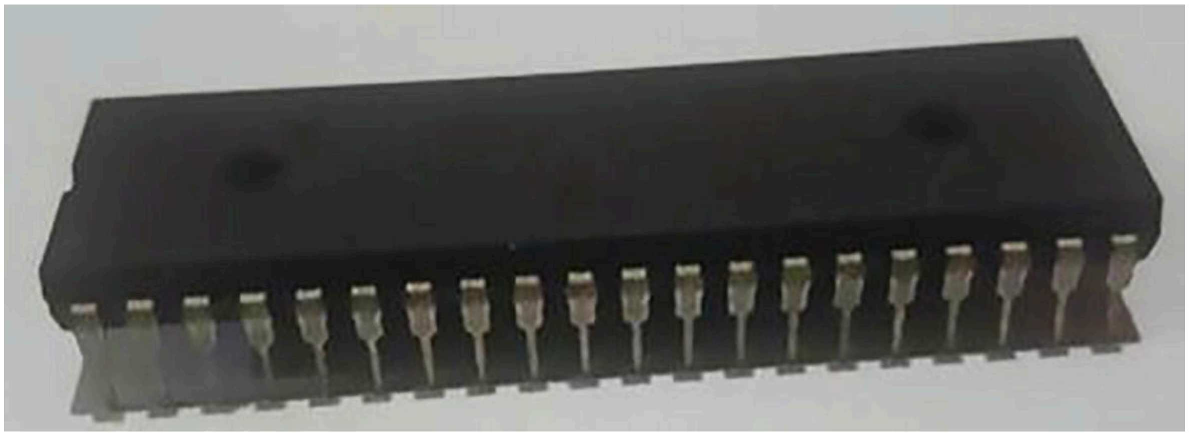


Figure 1-1 PIC16F1719 in a DIP package

A benefit of this particular microcontroller is that, in addition to being modern, it is produced in a DIP package, which makes it very easy to prototype on a breadboard. Therefore, you use it to test your design and use an SMD version in the final version of your product.

Programmer

A microcontroller is a blank slate without a program. Microcontrollers and other stored program devices rely on a programmer to load the program to the chip. I have found that using a microchip makes it easiest to understand how to program devices. Many device vendors have extremely expensive tools that are hard to find, even on their websites! To program PIC microcontrollers, you need a PICkit 3 or an MPLAB ICD 3.

I have used both and highly recommend that you buy an ICD 3. The reason is that the ICD 3 is much faster and saves you a lot of time in programming and debugging, especially if you plan on moving up to larger devices. However, you should only buy the ICD 3 if you are certain that you will be working with PIC microcontrollers for a long time, as at the time of writing, it costs over $200. The PICkit 3 may be used if you are evaluating the PIC microcontroller series, as it is available for $50.00. Generally, get the PICkit 3 if you are evaluating PIC microcontrollers and the ICD 3 if you intend to work with these devices for a while.

Figure 1-2 shows the PICkit 3, and Figure 1-3 shows the ICD 3.

Figure 1-2 PICkit 3

Figure 1-3 ICD 3

The ICD 3, ICD 4, and ICD 5 use an RJ-11 type adapter. I recommend that you get this programmer as well as an adapter to allow for easy breadboarding from RJ-11 to ICSP.

You can also follow along if you have a new programmer such as a PICkit 4, PICkit 5, or MPLAB Snap programmer. You can also buy a PICkit 3 Clone from retailers, or you can build your own.

Gathering the Software

The hardware is necessary for building the circuits. However, we are not fiddling with 555 timers here! We need software to make everything work. All the software needed to program PIC microcontrollers can be found on the Microchip Technology website.

MPLAB X IDE

I have heard people complain about the old IDE microchip thousands of times. Let me assure you that MPLAB X is nothing like MPLAB IDE (see Figure 1-4). It is a lot better. Microchip Technology has come a long way. I have used a lot of vendor tools, and Microchip offers the most effective plug-and-play functionality I have come across. Some rather pricey compilers don't offer much more than the ones provided for PIC microcontrollers. Microchip even offers an IDE that is cloud based! This cloud-based MPLAB Xpress IDE is best suited for new users or if you want to program the microcontroller on a machine that you need special permissions. A good example of this is would-be students or a corporate environment where going through the IT department would be a lengthy process.

If you purchased an Xpress evaluation board and are still not sure if you want to use the PIC microcontroller, then you may use the cloud-based IDE to get up and running quickly. However, if you decided on using PIC microcontrollers, then the on-premises software for microcontroller development is a lot better. The primary reason is that if something goes wrong, you can be assured that it is not a connection problem. The other reason is that as your code grows and your skills develop, you will need all the features of MPLAB X, which has the power of NetBeans behind it. Stick with the on-premises software.

I know there are going to be those among you who prefer to use a command-line interface and text editor. I also enjoy that method of doing things, when there is no IDE available. I like the KISS principle—let's not make things more complicated than they need to be. This book takes a pragmatic approach. IDEs are simple to use. Thus, we use them.

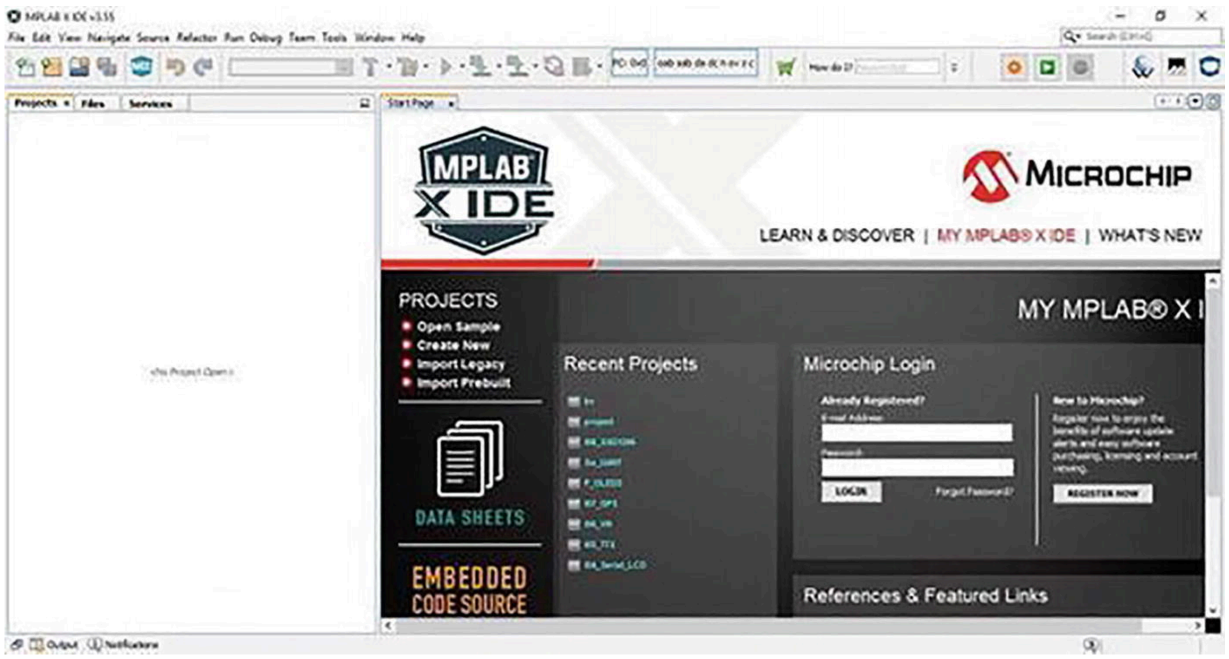


Figure 1-4 MPLAB X IDE

XC Compilers

A lot of people don't value compilers. Many vendors boast about how easy it is to get started with their chips and pack mouthwatering goodies into every bite of silicon. However, they make the compilers so expensive that they aren't worth it in the end. Microchip offers the XC compilers to get started with PIC microcontrollers. The best part is it's free of charge. In this book, I focus on XC8. However, be rest assured that once you get over the learning curve of how this compiler operates, you will be thankful that you chose to use PIC microcontrollers. This is because it is easy to transition from 8- to 16- and 32-bit microcontrollers without having to learn a different environment. The XC8 compiler is available for download on the Microchip Technology website.

Setting Up Shop

In this book, I interface the microcontroller to a lot of modules and design a lot of circuits. However, if you want to do likewise, you must acquire at least a minimum of equipment to be able to get the most out of this book. Recommended equipment is covered in the following sections.

Multimeter

The multimeter is a staple of electronics. Therefore, I highly recommend you invest in at least *two* multimeters. The reason you need at least two is because you need to measure voltage and current at the same time. For this book, any multimeter that can measure DC voltage, current, and resistance should suffice.

Oscilloscope

No electronics workbench, lab, or shop is complete without an oscilloscope. This device is undoubtedly one of the most important test instruments you'll have, particularly when you're working with microcontroller-based circuits. Even if you do not want a full scope, I recommend you get the Velleman pocket oscilloscope. It is reasonably priced and works rather well for basic work.

Power Supply

Make sure to get a good bench power supply. The 1.2−15 v range and at least a 5 amp rating will suffice.

Shopping for Supplies

When starting with microcontrollers and electronics in general, people often wonder where they can buy supplies and items. In general, you can buy most of these items from Amazon, eBay, Digi-Key, Mouser Electronics, or AliExpress. I recommend you buy passives from sites like AliExpress and eBay, as you are likely to get better deals on these in the Chinese market. However, microcontrollers, active devices in general, and programmers should always be bought from reputable suppliers, as they may not be genuine or may not function as required. There are instances where companies bought chips (namely, ATmega328P) from the Chinese market and it turned out that these chips were total imitations and did not work.

To sum it all up: be vigilant when purchasing electronic components and equipment. If it's too good to be true, *then stay away*. *Do not buy it.*

In general, you need to set up a basic electronic shop. You need various resistors, capacitors, and a few semiconductors, and of course your basic side cutters, pliers, and screwdrivers.

Table 1-1 lists the components you need to purchase to get the most out of this book.

Table 1-1 Recommended Hardware for This Book

Item	Quantity	Vendors	Product Numbers
PICkit 5/PICkit 3	1	Digi-Key Electronics	PG164150 (PICkit 5) AliExpress/Amazon
		Mouser Electronics	579-PG164130 (PICkit 3) 579-DV164035 (ICD 3)
PIC 16F1719	1	Digi-Key Electronics	PIC16F1719-I/P-ND
		Mouser Electronics	579-PIC16F1719-I/P
ESP8266 Wi-Fi Module	1	Digi-Key Electronics	1188-1154-ND
		Mouser Electronics	909-MOD-WIFI-ESP8266
Logic Level Converter Module	2	Digi-Key Electronics	1568-1209-ND
		Mouser Electronics	474-BOB-12009
2n2222 or Similar (2N3904)	2	Digi-Key Electronics	2N3904FS-ND
		Mouser Electronics	610-2N3904
LM34 Temperature Sensor	1	Digi-Key Electronics	LM34DZ/NOPB-ND
		Mouser Electronics	926-LM34DZ/NOPB
Nextion NX3224T024_11 Touch LCD	1	ITEAD Studio	IM150416002

Item	Quantity	Vendors	Product Numbers
		Amazon ASIN	B015DMP45K
SSD1306 OLED (I2C)	1	Amazon ASIN	B01G6SAWNY
		AliExpress (Various Sellers)	
24LC16B EEPROM	1	Digi-Key Electronics	24LC16B-I/P-ND
		Arrow Electronics	24LC16B-E/P
HD44780 Character LCD	1	Digi-Key Electronics	1528-1502-ND
		Adafruit Industries	181
MCP4131 Digital Potentiometer	1	Digi-Key Electronics	MCP4131-104E/P-ND
		Arrow Electronics	MCP4131-103E/P
SIM800L GSM Module	1	Amazon ASIN	B01A8DQ53E
		AliExpress (Various Sellers)	
UBLOX Neo-6M GPS Module	1	Amazon ASIN	B071GGZDDR
		AliExpress (Various Sellers)	
EMIC 2 TTS Module	1	Parallax Inc.	30016
		SparkFun Electronics	DEV-11711
Serial LCD Module	1	Parallax Inc.	27977
		Digi-Key Electronics	27977-ND
RGB LED	1	Digi-Key Electronics	754-1492-ND
		Mouser Electronics	604-WP154A4SUREQBFZW

Item	Quantity	Vendors	Product Numbers
SN754410NE	1	Digi-Key Electronics	296-9911-5-ND
		Mouser Electronics	595-SN754410NE
ULN2003	1	Digi-Key Electronics	497-2344-5-ND
		Mouser Electronics	511-ULN2003A
Servo Motor	1	Jameco Electronics	1528-1075-ND
		Mouser Electronics	485-154
5v Stepper Motor	1	Jameco Electronics	237825
Brushed DC Motor	1	Digi-Key Electronics	1528-1150-ND
		Mouser Electronics	485-711
Seven Segment Displays	2	Digi-Key Electronics	754-1467-5-ND
		Mouser Electronics	630-HDSP-513E
Pushbuttons	5	Digi-Key Electronics	P8011S-ND
		Mouser Electronics	667-EVQ-PAC07K
LEDs	10	Digi-Key Electronics	C503B-RCN-CW0Z0AA1-ND
		Mouser Electronics	941-C503BAANCY0B025
1N4001 Diode	2	Digi-Key Electronics	641-1310-3-ND

Item	Quantity	Vendors	Product Numbers
		Mouser Electronics	821-1N4001
10 uF Capacitors	2	Digi-Key Electronics	493-4771-1-ND
		Mouser Electronics	647-UCA2G100MPD1TD
1k Resistors	10	Digi-Key Electronics	CF14JT1K00CT-ND
		Mouser Electronics	71-PTF561K0000BZEK
10k Resistors	10	Digi-Key Electronics	CF14JT10K0CT-ND
		Mouser Electronics	279-YR1B10KCC

In addition, you need an HC-05 Bluetooth module, which can be found on various sellers on AliExpress and Amazon. Make sure to have a bench to dedicate solely to electronics work, and you will also need to buy some storage containers for all your components. Unlike using platforms, where everything is on a board that you can simply pack away in a kit, setting up your chip on a breadboard requires time. Therefore, having a dedicated workbench will save you a lot of time.

Conclusion

That brings us to the end of the first chapter. In this chapter, we covered gathering the required hardware and software to get started with PIC microcontroller development. This chapter laid the groundwork required to continue your fascinating journey. While you are waiting for your items to arrive, you may take a look at the next chapter, which focuses on getting you acquainted with the C programming language.

A. Subero, *Programming PIC Microcontrollers with XC8*, Maker Innovations Series
https://doi.org/10.1007/979-8-8688-0467-0_2

2. The C Programming Language

Armstrong Subero[1]

(1) Moruga, Trinidad and Tobago

In this chapter, we begin our foray into embedded systems development. We will begin by looking at the language of choice for embedded systems development. While there are many languages that you may use to develop your embedded applications, no language is as popular in modern development as C, and that trend won't change anytime soon. In this chapter, I of course cannot cover every nuance of the C programming language. However, I will cover the most important subset of the language that will be used for embedded systems development, particularly with resource-constrained PIC microcontroller devices. While more experienced persons may comfortably skip this chapter, persons who are not familiar with C or come from an environment such as Arduino will benefit a lot from this chapter as it gives a "survey" of the embedded C programming language.

C

When I first started embedded systems development, the language of choice was Assembly. Assembly language programming is tedious and error-prone, and many times it's hard to effectively express higher-level algorithms without a lot of hard work. For older microcontrollers, Assembly was the only reasonable option due to the limitations of the device architecture. Don't get me wrong, if you are writing highly optimized code, a senior engineer may from time to time need to integrate inline Assembly to optimize certain compute-intensive algorithms, particularly those for digital signal processing and motor control algorithms.

Throughout the years, we have seen many languages come and go. Languages like C++, Java, Python, and, most recently, Rust have been used to develop embedded systems. While each does find its use, hardly any of them can replace C. C was designed for memory-constrained devices, and it provides an unparalleled level of control over the device hardware. C++, while a good alternative, adds too much overhead for most projects utilizing microcontrollers. Java is used in some high-end systems, and Python has seen some use in the hobby and education markets. However, if you ever want to get into professional embedded development or only have time to learn one language, that language should be C.

Will Rust Dethrone C in Embedded Development?

Before we get into C programming, I thought I would address the topic of using Rust in modern embedded systems development. Recently there has been a lot of talk surrounding Rust replacing C as an alternative for embedded systems development. Indeed, the emergence of Rust has sparked enthusiasm in the embedded community. Rust has a focus on things that C generally lacks. Things like safety, concurrency, and zero-cost abstractions make Rust a good contender for embedded development.

Note In case you are familiar with the term, a zero-cost abstraction is the name we give to the ability to utilize language constructs such as functions and data structures without adding any runtime overhead. This feature is desirable in embedded devices that have limited resources.

However, for low-memory devices and particularly devices with 8-bit architecture, C has unmatched proximity to the hardware and minimal runtime. Rust was designed for 32-bit and 64-bit architectures, and it shows it is not suitable for 8-bit devices. Something that is often overlooked is that general-purpose software development, while having a lot in common with embedded development, is not the same. While general-purpose software development benefits from using the latest framework and the hottest new library, in the space of embedded development, tried and true is preferred over new and hip.

There are decades worth of trusted and battle-tested C libraries that we can develop our systems with; a lot of developers are familiar with C, and they know what works. If a company has invested in tools, both hardware and software, that are optimized for C programming, there is little incentive to switch to Rust. So many legacy embedded systems are written in C that have certifications and licenses in place that at least in our lifetime, Rust will not replace C as the dominant language in embedded programming.

Where I do see a place for Rust is in the generation of safety-critical and safe, in general static, libraries that can be linked to existing C projects. The compiled object code from Rust can be incorporated into larger projects during the linking phase, which is a process you as an embedded developer can look at if you have an interest in Rust development. However, for new and legacy 8-bit designs, Rust won't replace C for the vast majority of projects.

C Programming

The C programming language was designed when computing power was a fraction of what it is today. Forget gigabytes of memory; kilobytes were the order of the day, and there were no gigahertz multicore processors; a single-core processor running at a few megahertz was commonplace. While general-purpose computing has far moved beyond that point, embedded devices still maintain those specifications. Modern 8-bit PIC microcontrollers generally run in sub 50 MHz range and have a few kilobytes of RAM and ROM available for us to work with, making it ideal for us to use with C.

The beauty of C lies in its adaptability to run across a variety of platforms. C can manipulate bits and bytes at a fundamental level. In the realm of embedded programming, every byte of memory and every cycle of processing power can have significance depending on the application. C can directly access and manipulate hardware.

For all its complexity and power, C is a rather small language. As anyone who has had the pleasure of working with C++ will tell you, sometimes having fewer language features can be a blessing in disguise. These are the main topics we need to cover in C programming, which will allow you to write your applications with the language:

- Program Structure and Syntax
- Control Structures
- Loops
- Arrays
- Functions
- Pointers
- Structures and Unions
- Dynamic Memory Allocation
- Preprocessor Directives

This subset of the language forms the core components of C programming and of structural programming in general. Once you master these core principles, you will be well on your way to mastering intermediate and advanced concepts of the language.

C Program Structure

C programs are founded on the principles of structured programming. Structured programming is a programming paradigm that emphasizes the use of well-organized, hierarchical structures that we use to write our programs. When we write our C programs, we break down our program into small and manageable units that allow us to write programs that are clear and easy to maintain. When we do this, we can create code that is logical and readable. Not only that but our code will follow a linear sequence that is in the order which will be executed by our processor. If we look at Listing 2-1, we see the structure of a typical C program.

```
/* This is a comment that usually contains
information such as the name of the author, date,
and program name and details */
```

```
// This is where files are included and
preprocessor directives  take place

// This is an example of a header file
#include <stdio.h>

// This is the main function
int main(void)
{ // opening parentheses is very important

 printf("Hello World!"); // some code is executed

   return 0; // no code is executed after a return
statement
} // closing parentheses
```

Listing 2-1 The Structure of a C Program

While at first this program structure may seem to be very complex, as we break it down in the following sections, you will be able to understand how the C program works.

Comments

The first thing we see in our program is the comments. Comments are used in your program to describe lines of code, what they are, and what their intended function is. Comments are useful because they are not only used to let other persons know what your code does, but it also allows you to remember what your code is doing when you look at it sometime in the future when you revisit your code. Comments you make in your code are ignored by the compiler. Comments can be used wherever you need them. We can write our comments using both single-line and multiline structures.

Multiline comments allow us to add comments to our program that span several lines. These multiline comments are done using forward slashes and asterisks. These comments are called C-style comments, and while they are usually used to comment multiple lines, they can also be used to do single-line comments as well. We see how multiline comments are done in Listing 2-2.

```
/* This is a C-style comment  that can be used
to span multiple lines */
```

Listing 2-2 Multiline Comment

There are also single-line comments that span a single line using two forward slashes. These comments are called C++-style comments and are prevalent in "modern" C standards. These were first used in C++ and then were introduced into C. These comments allow for a short and simple style. We see C++-style single-line comments in Listing 2-3.

```
// This is a C++ Style Comment Occupying a single
line
```

Listing 2-3 Single-Line Comment

I prefer this C++-style comment for the majority of my comments, and I have worked on projects that use solely C-style comments. The style of commenting you choose will depend on the organization you are working at, and for your projects, it is a personal preference.

Note The use of C++-style comments in C programs is something you will encounter a lot as you learn about C programming. As the language evolves, you will see features from each being added to each other to make them compatible with each other. For this reason, sometimes you will see C/C++ compilers or C/C++ development. C is a subset of C++, but some features from C++ were integrated into C. In this book, we focus on C, not C/C++-style development.

Variables and Constants

As we continue looking at the structure and syntax of C, we will encounter variables. A variable in C is a name we see to refer to some location in memory and allows the programmer to assign a value to the location. Variables can be changed during the execution of a program unless they are explicitly declared as constants. In C, we must declare variables before we use them in a program for the first time. This is because we store variables in main (data) memory and we need to reserve space for them. Essentially you are requesting memory from the computer to store a specific type of data. Variables are nothing more than registers in our data memory space. In Listing 2-4, we see how a variable is declared.

```
// This is a variable being declared
int A;
```

Listing 2-4 Variable Declaration

After we declare a variable, we can assign a value to it. Assignment is the process of storing a new value in an already-declared variable. For example, our variable "A" can have a value assigned to it. When we perform an assignment, we place a specific data value into the memory location allocated for the variable. Listing 2-5 shows how we can do a variable assignment.

```
// This is a variable assignment
A = 10;
```

Listing 2-5 Variable Assignment

We can also perform something known as initialization. Initialization is the name we give to the process of assigning a value to a variable at the time of its declaration. For example, if we look at Listing 2-6, we can assign our variable A the value 10 at the time of declaration.

```
// This is a variable being declared and assigned
at the same time.
int A = 10;
```

Listing 2-6 Initializing a Variable

Assigning values to variables does not make them permanent. When we assign a value to a variable, usually during the program, the variable changes, hence the name "variable." After assigning a value to a variable, the value can be changed or updated.

Different variable types may be used in the C language. For example, there are variable types that we use to store letters and some types we use to store numbers. Table 2-1 shows us the common variable types in the C language.

Besides variables, we can also have constants. A constant is the name we give to a type of variable that is fixed and does not change during the execution of the program. Constants are not stored in data memory but are stored in program memory within the microcontroller. To make a variable a constant, we add the keyword const as a prefix to the variable name. In Listing 2-7, we see an example of a constant.

```
// initialize the PI constant
const PI = 3.142;
```

Listing 2-7 Initialization of a Constant

Table 2-1 Common Primitive Variable Types

Variable	Definition	Example
char	This variable type is used to store single characters	char A = ‘A’
int	This variable type stores integers, which are the names we give to whole numbers without any fractional or decimal parts	int A = 1
float	A float is used to store decimal numbers and usually has up to 23 bits allocated for the mantissa, which is about six to seven decimal digits	float A = 6.1234567890…23
double	A double is used to store decimal numbers that usually have up to 52 bits for the mantissa, which is about 15–16 decimal digits	double A = 6.1234567890…52

These data types collectively form what is known as the primitive data types in the C programming language and are the basic building blocks for all other different kinds of data. Each of these types has predefined sizes and characteristics, which gives them the foundation for variable declarations and memory allocation.

A Closer Look at Integer Types

When designing microcontroller programs, generally you want to avoid floating-point variables unless necessary. While some 32-bit microcontrollers may have a floating-point unit (FPU) that will allow them to perform floating-point operations with ease, generally you avoid using floating-point variables unless you have to use them. For this reason, we will look at the integer types in a bit more detail.

Integer types in addition to the standard “int” may also have types such as “char”, which ranges from -128 to 127; “short” types, which range from -32,768 to 32,767; a “long” type, which ranges from -2,147,483,648 to 2,147,483,647; and a “long long” type, which is a 64-bit integer with very large size.

Something to note about these different integer types is that each one can be either signed or unsigned. We use the unsigned modifier to represent integers that have non-negative values and the signed modifier for signed values. Since they both occupy the same space in memory, if we use signed representation, the total value we can represent as a signed representation is less than if we were to have an unsigned representation for that value. For example, a signed int can represent -2,147,483,648 to 2,147,483,647, but if we were to use an unsigned representation, the value that would be represented would be from 0 to 4,294,967,295.

When representing integer types, we may run into an issue. The issue is that the range that can be represented in each integer type is dependent on the compiler. So to avoid this problem, it is my recommendation when using the integer data types to use the “stdint.h” header to write more portable and robust code. This is because this library provides functions that allow for control over the size and signedness of the integers. These include the following:

- Signed Integers: int8_t, int16_t, int32_t, and int64_t, which are signed integers with precise bit widths of 8, 16, 32, and 64 bits, respectively
- Unsigned Integers: uint8_t, uint16_t, uint32_t, and uint64_t, which are unsigned integers of 8, 16, 32, and 64 bits, respectively

Some other integer types are exact width types “intmax_t” and “uintmax_t”, which are signed and unsigned maximum widths supported by the implementation of your compiler. In general, try to use the standard types when you are writing new programs, but you should be aware of the legacy types in case there are programs you need to maintain or update.

Operators

Mathematics and logic are what a CPU thrives on. In C, many symbols allow the microcontroller to perform logical and mathematical functions. Operators are fundamental symbols that we use to manipulate, compare, or combine data in expressions. Operators perform specific actions on operands, which are variables, constants, and expressions allowing us to later change their value and generate new results. C has different classes of operators that all serve different purposes. The operators we will look at here are arithmetic, relational, and logical operations as they are the operators you are most likely to encounter in your program development.

The first type of operator we will look at is the arithmetic operator. Arithmetic operators are used for performing mathematical computations within our programs. Look at Listing 2-8 to see a list of the arithmetic operators in C.

```
// Addition operation adds operands
X + Y;

// Subtraction operation subtracts operands
X - Y;

// Multiplication multiplies operands
X * Y;

// Division divides operands
X / Y;

// Modulus finds the remainder after division
X % Y;

// Increment increases value by one
X++ or ++x;

// Decrement decreases the value by one
Y-- or --Y;
```

Listing 2-8 Examples of Arithmetic Operators

The operators perform the basic arithmetic operations you are accustomed to addition, subtraction, multiplication, and division. You also have the modulus operator that finds the remainder after division and the increment and decrement operators.

What you should know about the increment and decrement operators is that we can use the increment "++" or decrement "--" operator either as prefix, where the operator precedes the variable where the value of the variable is changed before the expression is evaluated, or postfix notation, where the operator follows the variable.

The difference is as follows. When the prefix notation is used, the value of the variable will be changed before the expression is evaluated. Let's say we have the value 5 assigned to variable x in our code; then if we write "++x" in our code, x will increment to 6 and return the updated value 6 in the expression. If we do it postfix, then x will return the current value which is 5 in the expression and then increment it to 6. For the majority of programs, whether you use prefix or postfix notation rarely will it affect the operation of your program. In this book, I use the postfix notation.

Note In the following sections, you will see reference being made to Boolean operators and Boolean values. These are values in programming that represent true or false states. They are often represented as integer 0 for false and integer 1 for true and are the backbone of decision-making in C programs.

The next type of operators we will look at is relational operators. Relational operators are used to compare values and determine the relationship between them. The key defining facet of these operators is that they return a Boolean value "1" for true or "0" for false based on whether the comparison is true or not. We must understand relational operators since they allow us to build logical conditions within our code. Using relational operators, we not only compare numerical values but we can also evaluate expressions and variables of different data types. Listing 2-9 gives us the relational operators.

```
// Checks for equality
X == Y;
```

```
// Checks that values are not equal
X != Y;

// Determines if the first operand is greater than
the second X > Y;

// Determines if the first operand is less than
the second
X < Y;

// Checks if the left operand is greater than or
equal to the // right one
X >= Y;

// Checks of the left operand are less than or
equal to the
// right one
X <= Y;
```

Listing 2-9 Examples of Relational Operators

We also have logical operators within our program. Logical operators are used to perform logical operations on Boolean values and give us a Boolean outcome. We use logical operators in conjunction with relational operators to make rather complex decision-making. Listing 2-10 gives us example of logical operators.

```
// Logical AND operator
 X && Y;

// Logical OR operator
X || Y;

// Logical NOT operator
!(X);
```

Listing 2-10 Examples of Logical Operators

Boolean operators follow something we know as short-circuit evaluation. This means that if the result of the decision can be determined by the first operand, then the second operand might not be evaluated.

Bitwise Operations

In C programming, what is simple at heart but in reality forms part of an advanced skillset is known as bitwise operations. These are part of a larger part of low-level programming I like to call “bit-twiddling.” Bitwise operations manipulate individual bits within binary operations of data. We use them when we want to access and manipulate individual bits in registers, and it’s something we will see as we move forward. The bitwise operators are as follows:

- AND (&): Compares corresponding bits of two operands. If both bits are 1, the result is 1; otherwise, it’s 0.
- OR (|): Compares corresponding bits of two operands. If either bit is 1, the result is 1.
- XOR (^): Compares corresponding bits of two operands. If the bits are different, the result is 1; if they’re the same, the result is 0.
- NOT (~): Flips each bit of a single operand (0 becomes 1, and 1 becomes 0).
- Left Shift (<<): Shifts the bits of the left operand to the left by several positions specified by the right operand.
- Right Shift (>>): Shifts the bits of the left operand to the right by several positions specified by the right operand.

You sometimes need to use these bitwise operations when you are doing time-critical operations and need to maximize efficiency. A little fun application of these on an MCU is generating complex patterns in LEDs. It’s not all fun and games though; the ability to manipulate bits with bitwise operations can significantly improve efficiency in handling binary data, which we will see later on.

We now have the foundation we need to move on to the other topics in C programming.

Controlling Program Flow

Now that we have a survey of C programs and the basics of the structure and syntax, we can take a look at what is known as control structures in programming. Control structures are mechanisms that allow us to modify the flow of execution within our program. When we introduce control structures in our program, we can allow the execution of different code paths in our program under various conditions. The three main types of control structures are as follows:

- Decision-making and selection structures
- Looping and iterative structures
- Control transfer structures

By altering the sequence of execution of code within our program, we can achieve a lot of desirable effects.

if Statement

The first control structure we will look at is the if statement, which is a type of selection structure. We use the if statement to make decisions within a program. The if statement is the most fundamental of all control structures. Using the if statement, we can make decisions and execute different sections of code depending on whether conditions are true or false. The general structure of the if statement is given in Listing 2-11.

```
if (condition)
{
  // code block to execute if it is true
}
```

Listing 2-11 The if Statement

The condition is the expression within the structure that evaluates to be either true or false. If the condition is true, then the code block within the curly braces is executed. If the condition is false, then the code block is skipped, and the program continues with the next statement after the “if” block.

else Statement

The complementary to the “if” statement is the “else” keyword. As we mentioned in the previous section, according to the condition we wish to check sometimes the result is that it evaluates to false. In such a scenario, rather than going to the next statement after the code block, we take a different course of action when the condition in the "if" statement is not met. We see the structure of the "if-else" structure in Listing 2-12.

```
if(condition)
{
   // code to be executed if the condition is true
}
else
{
  // code to be executed if the condition is false
}
```

Listing 2-12 The if-else Statement

When the condition in the if statement evaluates to true, the code block in the if condition is executed, and if the condition is false, then the code block within the "else" statement's curly braces is executed instead.

else-if Statement

We can extend the else statement even more with the else-if statement. By adding the "else-if" statement to an existing if-else block, we can deal with multiple conditions sequentially and provide more branches of logic. Using this statement, we can check for more than two conditions in our program. In Listing 2-13, we have an "if else-if else" structure.

```
if(condtion1)
{
  // code to be executed if conditon1 is true
}
else if(condition2)
{
  // code to be executed if condtion2 is true
}
else
{
  // code to be executed if all conditions are
false
}
```

Listing 2-13 The “if else-if else” Structure

When we look at the code block, we can trace the logic execution as follows. The initial if statement checks the first condition. If the condition is true, then the code block executes, and the program skips the else if and else blocks. If the condition is false, however, then the program evaluates condition2, and if it is true, its associated code block executes and the program skips the remaining “else if” and “else” blocks. If both conditions are false, then the code block within the final statement will be executed.

switch Statement

In C programming, the primary selection structure is the switch statement. According to the statement you are evaluating, sometimes you end up with a rat nest of else-if statements. When we have many options to consider based on a single variable, to improve code readability and provide a cleaner structure, we opt for the switch code construct.

With the switch statement, we can perform multiway branching based on the value of the expression we pass to the statement. We see the structure of the switch statement in Listing 2-14.

```
int choice = 2;

switch (choice)
{
      case 1:
          // code for choice 1
          break;
      case 2:
          // code for choice 2
          break;
      default:
          // code for default case
          break;
}
```

Listing 2-14 The Switch Statement

In this structure, we switch based on our choice variable. Since the variable is 2, the code for choice 2 will execute. If you look at the construct, you will observe that there are break statements within every case block. You must remember to include these break statements within the cases; otherwise, the flow would fall to subsequent cases until the break statement is reached. If none of the other cases are met, the program flow will fall to the default case.

Generally, if you are evaluating a single variable against multiple constant values, you would use a switch statement; however, if you have complex conditions and nonconstant expressions or need to be flexible in how you evaluate multiple conditions sequentially, you would use the if-else construct.

for Loop

Many times, when programming, you need to execute a block of code repeatedly for a specified number of times, and to do so, we iterate over a range or until a certain condition is met. In such a scenario, we would turn to looping structures to control our program flow. The first such looping structure we will look at is the for loop. The syntax of the for loop is given in Listing 2-15.

```
for (initialization; condition; update)
{
   // code to be executed in each iteration
}
```

Listing 2-15 The for Loop Structure

In this code, the initialization step initializes a control variable and is executed only once at the beginning of the loop. Our condition is a Boolean expression that determines whether to continue the loop. If the condition is true, the loop continues; otherwise, it terminates. The update statement modifies the control variable after each iteration and is typically to increment or decrement the variable’s value.

while Loop

An alternative to the for loop is the while loop. The while loop is used when we want a block of code to execute repeatedly for some time as long as the condition specified is true. Unlike the "for" loop, the while loop is simpler but, in my opinion, more dangerous to use as you are relying only on the condition for loop continuation. If the while loop is handled improperly and the loop condition remains true, then you can enter into what is known as an infinite loop. When an infinite loop takes place, a program will continue to execute indefinitely without termination under the normal program execution. We see what the while loop looks like in Listing 2-16.

```
while (condition)
{
  // code to be executed
}
```

Listing 2-16 The while Loop

If we look at Listing 2-17, we will see an example of another while loop, which is also an infinite while loop.

```
while (1)
{
  // code that runs forever
}
```

Listing 2-17 Infinite while Loop

Within the realm of embedded systems programming, you will usually see this code construct. It is called a super loop. The super loop is used in embedded systems where there is no operating system or there are only bare metal programs.

However, for most general purposes, we integrate a stop or termination condition into the loop as shown in Listing 2-18.

```
int i = 0;

while (i < 5)
```

```
{
   printf("Value of i: %d\n", i);
   i++;
}
```

Listing 2-18 while Loop with Termination Condition

In this loop, we initialize the variable “i” to 0 before entering the loop. The “i < 5” is the condition that checks if “i” is less than 5. As long as this condition is true, the loop continues. The variable i is then incremented by one each loop iteration “i++”. This ensures that the loop will terminate when a certain condition is met. In almost all situations where you would use a while loop, a for loop can be substituted. There are situations, however, where you may not know how many iterations are needed beforehand; for example, you may opt to use a while loop. If you do use a while loop, to avoid infinite loops, always remember to ensure that there is a condition where the loop condition eventually becomes false to avoid these issues.

do-while Loop

The next looping structure you need to look at is the do-while, also called the “do” loop. The do-while loop executes a block of code repeatedly until a specified condition becomes false. It works just like the while loop; however, the check takes place at the end of each iteration, which ensures that the code block executes at least once, even if we have an initially false condition. The structure of the do-while loop is given in Listing 2-19.

```
do
{

} while (condition);
```

Listing 2-19 The do-while Loop

I have rarely encountered situations where a do-while loop was needed; one place where they are used though is in console-driven menu-driven programs where users select options from a menu.

Control Transfer Statements

In the next two sections, we will take a look at what are known as control transfer statements. These allow the transfer of control from one part of a program to another. These allow programmers to alter the flow of execution, change the sequence of code execution, and jump to specific points within a program. The four control transfer functions we will look at are the break statements, continue statements, and goto. There is also the return statement in this category, which we will look at when we discuss functions.

break Statements

When we looked at the switch construct, we encountered the break statement. The break statement is used within loops, and the switch statement is by immediately exiting the nearest enclosing loop or switching to the next case in a "switch" statement. We typically use it for terminating our loops prematurely before we complete all the iterations. We see an example in Listing 2-20.

```
for(int i = 0; i < 5; i++)
{
  if (i == 3)
  {
    break;
  }
   printf("Value of i: %dt\n", i);
}
```

Listing 2-20 The break Statement

In this case, by including the break statement once the loop reaches three iterations, the loop will be terminated.

continue Statement

We use the continue statement when we want to skip the current iteration of a loop and move to the next iteration. When there is a specific condition that we want to skip the remaining code for and move to the next iteration of the loop without terminating the loop entirely, this is the statement we use. This gives us very fine-grained execution. Let's understand how this statement works by looking at the example in Listing 2-21.

```
for (int i = 0; i < 5; i++)
 {
    if (i == 2)
   {
        continue; // Skips printing when i equals
2
   }
    printf("Value of i: %d\n", i);
}
```

Listing 2-21 A continue Statement Example

In this example, what we do is that we have a for loop. When the i variable in the loop is equal to 2, the print statement is skipped due to the "continue". The loop then moves to the next iteration printing values from 0 to 4, except 2.

goto Statement

The goto statement is the forbidden statement within the C language. No other keyword causes as much shame and frowning as the goto statement. This statement allows us to perform an unconditional jump to a labeled statement anywhere, altering the program's execution. The goto statement is powerful but can lead to spaghetti code, which is why it is looked down upon so much.

Note Spaghetti code is the name we give to the style of programming where there is a convoluted and tangled control flow of a program. The logic of such a program is hard to follow, and the flow is difficult or many times near impossible to decipher for larger programs, akin to tangled strands of spaghetti, hence its name

In Listing 2-22, we see an example of the goto statement.

```
int x = 0;
```

```
start:
x++;
if (x < 10)
{
    goto start; // Jumps back to the 'start' label
}
```

Listing 2-22 The goto Statement

Generally, in modern C programming, the goto statement in my experience is seen as an “advanced” method of interrupting a program flow. It is an advanced technique because of the power it wields, and generally only senior-level engineers can get away with putting it into a code base. It has use in escaping from deeply nested for loops or if statements. I am including it here because if you come across a legacy code base, you may encounter this statement from time to time.

Arrays

A data structure you are sure to meet when writing C programs is the array. An array allows multiple values of the same data type to share a single variable name. These values are known as elements of the array. This way of having elements of the same data type makes an array known as something called a homogeneous collection. An array holds all these elements consecutively in memory. We can declare an array with or without a specified size, for example:

```
int temperatures[5] = {29, 25, 26, 25, 28};
```

or without a size being specified, the sequence can be written as

```
int temperatures[] = {29, 25, 26, 25, 28};
```

While we can declare the array with or without a specified size, at the time of declaration, the size of the array is fixed, which is to say the size remains constant throughout the program’s execution. If you need dynamic storage, then another data structure known as a linked list may be something that you can look into, but for the majority of programs you will write, an array will be enough.

You can access the elements in an array using their indexes, also called the indexes of the array. Within an array, the first element in the array has an index of 0. So to access the first element within the array, we can do something as follows:

```
int Monday = temperature [0]; // Monday will have
value 29
```

You can also assign a value to a single element of an array like so:

```
temperatures [2] = 27; // Element 2 now has the
value 27
```

Elements in an array are stored contiguous in memory, allowing for fast access based on indexes.

Functions

A function is a self-contained block of code for performing a specific task. Within the realm of structured programming, they allow us to create modular and good code structures. The three tenets of functions are as follows:

- Encapsulation: Functions allow us to encapsulate specific functionality within a reusable block of code, which allows other parts of the program to use their services without the need to understand their internal implementation.
- Modularity: Functions allow for the breaking down of complex tasks into smaller manageable units, which allows for modularity and code reuse.
- Return Types and Parameters: Functions also have return types and parameters that take inputs to perform operations based on the values.

The structure of a function is seen in Listing 2-23.

```
return_type function_name(parameter1_type
parameter1, parameter2_type parameter2, ...)
{
    // Function body - code that performs the task

    // Return statement (if the function has a
return type)
    return value;
}
```

Listing 2-23 Structure of a Function

If we look at the function, we will see that the function has a return type that specifies the type of data that the function will return after performing its designated tasks. There is also a return value that gives specific results that output back to the part of the program that initiated the function call. As we will see, developing programs forms the important basis for developing embedded applications. In Listing 2-24, we see an example function with a return value that can be done.

```
 // Function to calculate the square of a number
and return the result
int square(int num) {
    return num * num; // Returns the square of
'num'
}

// Using the returned value in an expression
int result = square(5); // 'result' will hold the
value 25 (square of 5)
```

Listing 2-24 Function with a Return Value

In this example, the function calculates the square of a value and returns the result. The result variable we created will hold the return value of the function.

When dealing with functions, something we can learn about is a function prototype. Function prototypes serve as declarations of information about a function to the compiler before the actual function definition. They specify the function's name, return type, and parameter types without providing the implementation details. The primary reason we create function prototypes is in the linking process during compilation. When we use function prototypes, they enable the linker to associate function calls in one part of the code with the actual definitions in another part that can be in the same file or across several files. We can look at an example of a function prototype:

```
// Function prototype
int addNumbers(int a, int b);

// Function definition
int addNumbers(int a, int b) {
    return a + b;
}

// Function call
int result = addNumbers(5, 7);
```

As we write programs through this book, you will see several uses of function prototypes.

Pointers

Pointers in C are variables that store memory addresses and allow the direct manipulation and referencing of memory locations. Despite the simple concept, many people have a problem grasping it. For this reason, I sparingly use pointers in this book, but pointers are a powerful feature of C that sets it apart from other languages. Once you get used to using pointers, you will have no problem finding uses for them to make very efficient and powerful programs.

Pointers are quite simple to understand. Despite the fact that most people are confused as to the purpose of the pointer, the idea behind them is really simple. Pointers are variables, just like an int, char, or float. An int stores numbers, a char stores characters, and a float stores a decimal number. A pointer is a variable like any other that stores the memory address of another variable.

Don't let the ampersand and asterisk scare you; all a pointer does is "point" to the location of other variables or data in the computer's memory.

We declare a pointer using an asterisk ("*") with the variable type they will point to. We initialize them with the address of the variable before use, for example:

```
        int *ptr;
        int x = 10;
        ptr = &x;
```

The Address-of operator ("&") retrieves the memory address of a variable, which can be stored in a pointer. The Dereferencing operator ("*") is what we use to access the value at the memory address held by the pointer.

Using pointers, we can perform things like array manipulation and direct memory allocation. One extremely useful application of pointers is that pointers are used to pass variables by reference, which allows functions to modify original variables. This is a powerful feature that we make use of in some of the projects in this book.

Structures and Unions

When writing C programs, you can do a lot only using arrays to write your programs. However, you will find that when you start to write complex programs, you need a more complex data type to manage your program. At this point, you will probably make use of structures. Structures are user-defined data types that allow you to group different variables of different data types under a single name, which can be contrasted with arrays we looked at earlier, which store only a single data type.

Using structures, we can create complex data structures by bundling together related data elements. This allows us to mimic and model real-world objects. Think of a person. A person will have a name, an age, and a height. Using a structure, we can create a structure to represent this arrangement. We see how the structure is done in Listing 2-25.

```
// A structure names a person with three members'
names, ages, and height
struct Person
{
    char name[50];
    int age;
    float height;
};
```

Listing 2-25 The Structure Code Construct

The composite data type we create with a structure can be defined using the "struct" keyword. The power of a struct lies in its ability to create variables of the type of the struct. We can create variables of type "struct Person" like this:

```
Struct Person person1, person2;
```

Now that we have our variables, we can access members of the structure we created using the dot (".") operator. For example, we can access and modify our person struct as follows:

```
strcpy(person1.name, "John");
person1.age = 25;
person1.height = 175.5;
```

Like other software constructs we have looked at thus far, we can also initialize structures at declaration, as in

```
struct Person person3 = {"Alice", 30, 165.2};
```

Using structs, we can create some very powerful ways to organize data and create new data types to suit our application. Anyone familiar with object-oriented programming (OOP) may recognize a struct as a type of rudimentary form of object encapsulation. Though we cannot perform advanced OOP functionality, there is still a lot we can do with the facilities provided to us by structs in C.

When you start getting into structs, you also will soon become aware of another construct in C called the union. Unions are like structs, but they store different variables in the same memory location, allocating memory to hold the largest member while allowing different types to share the same memory space.

We can declare a union using the "union" keyword, which makes its layout similar to structures as shown in Listing 2-26.

```
// Declaration of a union named 'MyUnion' with two
members: 'intValue' and 'floatValue'
union MyUnion
```

```
{
    int intValue;
    float floatValue;
};
```

Listing 2-26 The Structure

We can also access members of the union using the dot (".") operator as shown in Listing 2-27.

```
union MyUnion u;

// Accessing and modifying the intValue member
u.intValue = 42;

// Accessing the floatValue member (but
interpreting the bits of the intValue)
printf("%f\n", u.floatValue);
```

Listing 2-27 Using the Union

Using unions, we can do type conversion, as the same memory location can be interpreted based on context. It's also very efficient for memory optimization. Using the union, only one member of the union is used at a time, but multiple types of data can be stored. Unions can be a great tool, but you need to pay attention to how you change from one member of the union to another, as changing one member of a union and accessing another without proper synchronization can lead to undefined behavior.

Dynamic Memory Allocation

At this point in your C journey, you may have realized that programming in C is tightly intertwined with knowledge of memory including organization and manipulation. In C, we can perform what is known as dynamic memory allocation, which is the process of allocating and deallocating memory during the execution of the program. This process allows us to create variables whose size can be determined at runtime. To manipulate memory and allow their allocation and deallocation, we use a function from the C standard library. We use “malloc”, “calloc”, “realloc”, and “free” from the standard C library.

The “malloc” function allocates a block of memory of a size we specify in bytes and returns a pointer to the beginning of the block.

```
int *ptr = (int *)malloc(5 * sizeof(int)); //
Allocates memory for 5 integers
```

“calloc” allocates memory for an array of elements. When we use “calloc”, we initialize the memory to zero and return a pointer to the allocated memory.

```
int *ptr = (int *)calloc(5, sizeof(int)); //
Allocates memory for 5 integers, initializes to
zero
```

Sometimes we need to resize a previously allocated memory block to a new size, and we return a pointer to the resized block. To do this, we use the “realloc” function.

```
ptr = (int *)realloc(ptr, 10 * sizeof(int)); //
Resizes the memory block to hold 10 integers
```

When we use “malloc”, “calloc”, or “realloc”, we allocate memory; however, once we are done with the memory, we need to remember to release it so that it can be used again. To release the memory, we use the “free” function.

```
free(ptr); // Releases the memory allocated to ptr
```

We can see a standard example of how these memory allocations and deallocations can take place by looking at the example in Listing 2-28.

```
// Allocates memory for 5 integers
int *ptr = (int *)malloc(5 * sizeof(int));

if (ptr != NULL)
```

```
{
    // Use ptr to access allocated memory
    ptr[0] = 10;
    printf("%d\n", ptr[0]);

    free(ptr); // Deallocate memory when done
using it
}
else
{
    // Handle allocation failure
    printf("Memory allocation failed.\n");
}
```

Listing 2-28 Memory Allocation and Deallocation Example

Freeing the allocated memory is very important in our program. If we do not free previously allocated memory, then a memory leak can occur, and that leads to gradual consumption of memory resources over time. When we don't return memory that we no longer need, then we can introduce a host of problems in our application.

In microcontroller-based systems, this is especially dangerous as these devices have very limited memory resources available to them. 8-bit devices in particular can suffer from fragmentation where memory is continuously allocated and deallocated, where the memory is broken up into smaller unusable blocks that reduce the usable memory space.

Dynamic memory allocation is generally discouraged in embedded systems programming, particularly for microcontrollers like the PIC16F1719, due to the risk of memory fragmentation. Memory fragmentation occurs when continuous use of allocation and deallocation operations scatters the memory space, leaving sufficient total memory but not in contiguous blocks, which can prevent the allocation of larger blocks of memory. This fragmentation can lead to inefficient memory use and potential system instability, which are critical concerns in systems with limited memory resources. Instead, static memory allocation is preferred for its predictability and reliability in managing memory.

Preprocessor Directives

Before we discuss preprocessor directives, let's take some time to think a little about IDEs and compilers. The IDE (Integrated Development Environment) is a program just like your text editor, browser, or video game. The difference is that the IDE program has a special purpose. It contains everything you need to develop the program that will run on your microcontroller. That means it consists of various parts, such as a code editor where you type your code, a debugger that helps you look for errors in your code, and a lot of other things that simplify the whole process of development.

One such part of the IDE is the *compiler*. A compiler converts your code (in this case, written in C) into instructions that the microcontroller will understand. When this code is compiled, it is converted into something called an *object* file. After this step, a component called the *linker* takes these object files and converts them into the final file that will be executed by your microcontroller. There may be other steps in this process of generating the final *hex file* (program to be written to the microcontroller), but this is all you need to know.

Now we can understand what a preprocessor directive is. The preprocessor is another part of the IDE that uses directives, which cause the C program to be edited before compilation.

These preprocessor directives begin with a hashtag symbol. In XC8, you will encounter preprocessor directives a lot, especially with libraries that are designed to target more than one chip.

#define

The #define directive is the first we will look at. The #define directive is used to define constants, macros, or symbolic constants before compilation. It helps us create shorthand notations for values and expressions in our program. This statement is used a lot in embedded programming and is very useful. Instead of having to keep typing some constant, it is easier to use the #define directive. This is also useful in instances where constants may need to be changed.

For example, if we are writing a driver for an LCD that comes in two compatible variants—128x64 and 128x32—then instead of having to constantly write those numbers, since the dimensions of the LCD would remain the same, it is easier to write it as shown:

```
#define LCD_HEIGHT 128
#define LCD_WIDTH 64
```

A little warning though: Remember to omit the semicolon after the macro as it will generate compiler errors. Another important use of the #define directive is in the creation of function-like macros. These are macros that can be used to create a small "function" and are useful for the creation of small functions that may appear many times in your code, for example:

```
#define MAX(x, y) ((X) > (Y) ? (X) : (Y))
```

The most important use of such functions I have found in practice is that they do not require a specific type and can use any generic type. In the example in Listing 2-24, it doesn't matter if the parameters are int, float, or double; the maximum would still be returned. Learning to use the #define directive is very important. Sometimes you may see this referred to as a *lambda function*.

#if, #ifdef, #ifndef, #elif, and #else

These preprocessor directives are used for conditional compilation in the program. These directives are important. These directives are commonly used for debugging and to develop libraries that target multiple chips.

They are straightforward; we see an example of their use in Listing 2-29.

```
#ifdef PIC16F1717
#define SPEED 200
#elif defined ( PIC24F )
#define SPEED 300
#else
#define SPEED 100
#endif
```

Listing 2-29 Using Preprocessors for Conditional Compilation

Note that the conditional directives must end with an `#endif` statement.

#pragma

This is a C directive that in general-purpose programming is used for machine- or operating system−specific directives. This directive is compiler-specific, and it gives specific instructions or directives that we need to use.

Note The #pragma directive is not part of the C or C++ standard but is widely supported by many compilers. They are usually not portable between different compilers.

This directive is most commonly encountered to set the configuration bits of the PIC microcontroller. For example, here is an excerpt of the configuration of a PIC16F1719 microcontroller using pragma to set different configuration words of the device given in Listing 2-30.

```
#pragma config FOSC = INTOSC    // Oscillator
Selection Bits (INTOSC oscillator: I/O function on
CLKIN pin)

#pragma config WDTE = OFF       // Watchdog Timer
Enable (WDT enabled)

#pragma config PWRTE = OFF      // Power-up Timer
Enable (PWRT disabled)

#pragma config MCLRE = OFF      // MCLR Pin
Function Select (MCLR/VPP pin function is MCLR)

#pragma config CP = OFF         // Flash Program
Memory Code Protection (Program memory code
protection is disabled)

#pragma config BOREN = OFF      // Brown-out Reset
Enable (Brown-out Reset disabled)
```

Listing 2-30 Excerpt of Using #pragma to Set Configuration Bits

Assembly vs. C

While we have some forward-thinking persons who believe that Rust will replace C in embedded systems, some persons cling to the belief that Assembly is the only right way to develop 8-bit embedded applications. This may have been true many years ago, but all modern 8-bit PIC (and AVR) devices have a C-optimized architecture. Almost every 8-bit device manufactured by Microchip Technology from around the year 2015 has a C-optimized architecture as one of its highlighted features. Assembly language programming (while a desirable skill) just doesn't have the same allure as it did before.

Embedded applications today, even 8-bit ones, include complex protocols and algorithms that would be difficult to create with Assembly language programs. Most IoT applications, even far edge nodes consisting of 8-bit devices, would benefit from being able to implement algorithms in C compared to Assembly. Of course, if you wish to learn your chip on a deeper level, or you need tight optimization of certain time-critical or speed-critical parts of your code, you may need to use Assembly for that. For 95% of your applications though, you will not need to use Assembly language. I would still, however, recommend you learn the basics of Assembly because when debugging your applications, stepping through Assembly or looking at the disassembly of your program can tell you a lot. However, in this book, I omit the use of Assembly.

Conclusion

This chapter contained a basic overview of the C programming language. It would of course be impossible for me to cover all the constructs of the language in a single chapter. Covering the C programming language can take an entire book!

With just the concepts presented here, you can do a lot, as we covered the most important syntax of the language for our purposes. However, simply knowing the keywords of a language does not help you master it. It takes practice. If you are not proficient in the C language, I encourage you to find books and Internet resources to help you in your journey with the C programming language. If you are completely new to programming in general, I recommend you learn the basics from a good book. The book I recommend is *Beginning C: From Beginner to Pro,* Sixth Edition, by Ivor Horton and German Gonzales-Morris, available from Apress. That book will explore the basics in more depth as well as many advanced concepts. Many free resources on the Web teach complete beginner programming concepts.

A. Subero, *Programming PIC Microcontrollers with XC8*, Maker Innovations Series
https://doi.org/10.1007/979-8-8688-0467-0_3

3. Basic Electronics for Embedded Systems

Armstrong Subero[1]

(1) Moruga, Trinidad and Tobago

It would be impossible for me to cover all the electronics that are required for you to comfortably design embedded systems. Realistically speaking, no book, even one on electronics, can teach you all that you have to know. Truthfully you pick up the bits and pieces you need gradually over time and integrate them into your designs. This chapter will give you an overview of the foundations you need to be able to do your embedded designs.

Electronics

The difference between embedded systems designers and software engineers or IT technicians is the in-depth knowledge of the low-level hardware that embedded designers possess. Embedded designers must know electronics to effectively design embedded systems. We must remember, above everything else, that computers are simply complex electronic devices and microcontrollers are simply miniature computers. Resistors, capacitors, diodes, and transistors are some of the building blocks of computer hardware. To understand these more complex devices, it is important to understand the basic electronic components from which these devices are built.

Electronics is the science of electron flow. In electronics, we study the design, development, and application of devices and systems based on the flow and control of electrons through different materials. Electronics is a very diverse field, but in embedded systems, the branches of electronics we are concerned with are passive devices, semiconductors, circuit theory, and digital and analog systems. Digital and analog systems, especially their conversion and interconversion, are important to us because digital electronics-based devices like microcontrollers process information in discrete binary signals (0 and 1) and analog electronics deal with continuous signals; this is something we will look at in a later chapter. It would be impossible for me to cover all of the electronics required for embedded systems, but in this chapter, I will try to cover the most important concepts.

Electronics Components

As we stated, electronic circuits are based around the flow of electrons. These electrons are used to pump electrons around the circuit and end up back at the source. We build our electronic circuits by connecting varying electronic components in the part of these flowing electrons. These electronic devices fall into one of two categories, either active devices or passive devices.

Passive components generally cause some sort of power loss in your circuit; they usually modify the power of a circuit in some way and are composed of three types: resistors, capacitors, and inductors. Active components also change the power in a circuit by usually increasing it in some way. Active devices are usually integrated circuits, but we'll learn more about those later on.

Resistors

The first component we will look at is the resistor. A resistor is used in electronics to impose resistance in circuits. Resistors are rated by the number of ohms they possess, which is essentially a measure of the amount of resistance to the flow of electrons they provide. They restrict the flow of these electrons and release heat energy. They can be made from a lot of different materials that influence their properties and are used to reduce the flow of current in a circuit and to reduce the voltage. They are also used to divide voltages and pull up I/O lines.

Resistors come in a variety of packages, including an axial package, radial package, surface mount package, and a type of Single Inline Package (SIL) called a resistor array. Resistors are passive components and are one of the simplest types of devices you'll encounter. The schematic symbols for the resistor are given in Figure 3-1.

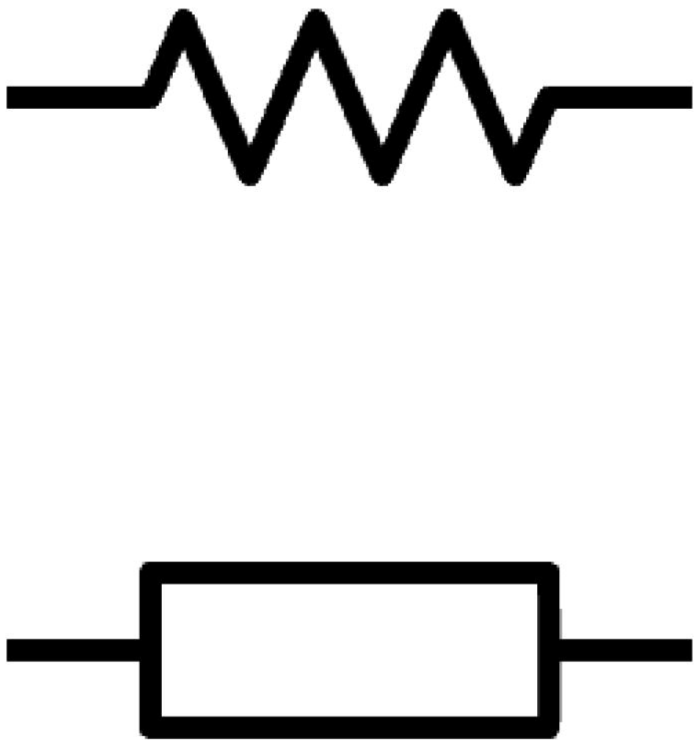


Figure 3-1 Resistor schematic symbols

Something to consider when using resistors is the wattage ratings that must not be exceeded. Typically, these values range from 1/8 watts to 2 watts. 1/4-watt resistors are the most common ones used in embedded systems design. The commonly found surface mount variety typically has a rating of 1/16 and 1/10 watts.

However, to ensure you use your resistor in its intended operations, it is best to consult the datasheet for the resistor you are planning to use. A datasheet is a document that tells you a little bit about the technical specifications of a product. For example, a resistor's datasheet includes electrical characteristics such as tolerance and operating curve, as well as the dimensions of the part. This information is very useful when you are designing a printed circuit board or PCB. Some component suppliers, such as Digi-Key and Mouser, also list the datasheets on their product pages. We see what a resistor looks in Figure 3-2.

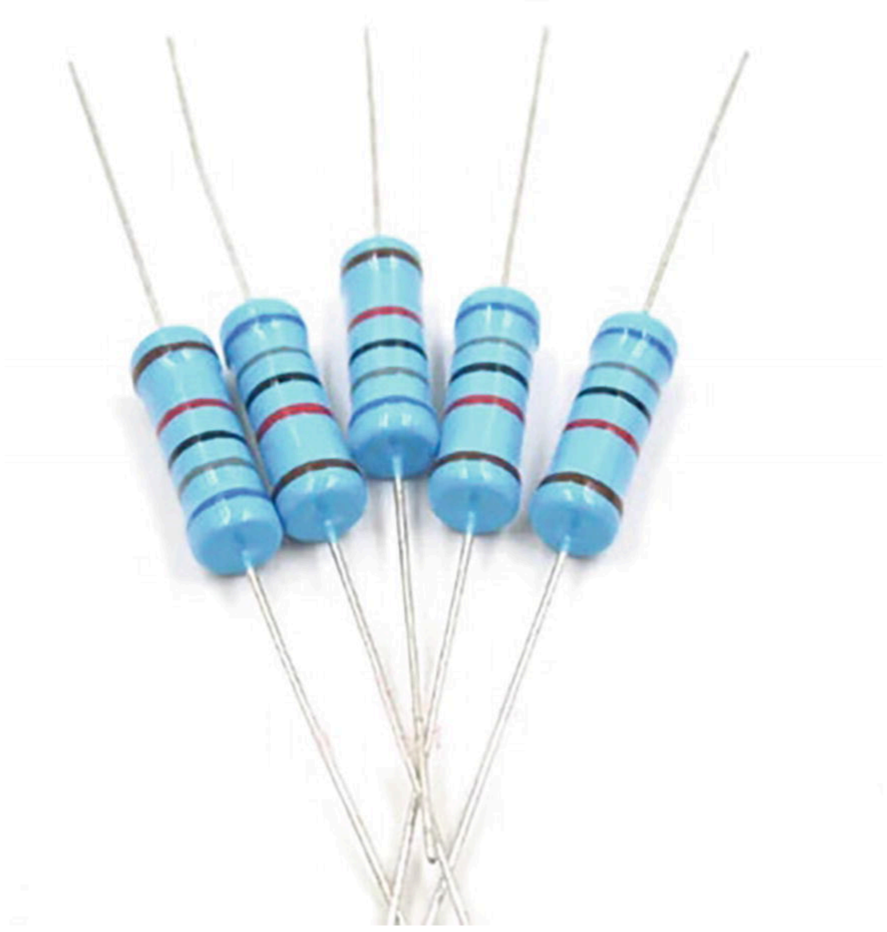


Figure 3-2 Some through-hole resistors

Most resistors have four bands. The first two bands are the most significant digits. The third band tells you of the power of 10 you have to multiply by, and the final band is the tolerance of the resistor. Usually, the tolerance can be ignored; however, for some applications, the tolerance must be within a very narrow range. Five-band resistors extend the precision of four-band counterparts by providing an additional significant digit. For five-band resistors, the first three bands represent significant digits. The fourth band is the multiplier, and the fifth band is the tolerance.

Tolerance is important depending on the application you have in mind. Usually, you can get away with using 10% or 5% tolerance.

However, with the falling prices of electronic components, it is not uncommon to see a 1% tolerance in a lot of designs. As technology improves, we may begin designing in 0.1% or higher precision values into general designs.

Surface mount resistors typically either use the E24 or E96 type markings. The E24 has three numbers. The first two numbers are the significant digits, and the third is the index of base 10 to multiply by. For example, a resistor marked 104 would be 100 kiloohms.

Something you need to consider when working with resistors is the temperature of the environment they are operating. Resistors, as we already mentioned, have wattage ratings. These ratings are connected to the dissipation capability of the resistor. The maximum dissipation of the resistor refers to the maximum capability of the resistor to convert electrical power into heat without damaging the device. As the resistor dissipates heat, it warms up the environment around it, so a colder environment will allow for greater heat dissipation. The heat of the resistor also affects the effective resistance of the device. It is best to consult the datasheet of the device, but generally increasing temperature will increase the resistance of the device. This is a relationship we can quantify with the Temperature Coefficient of Resistance (TCR).

An important construct we can look at with resistors is the voltage divider. The voltage divider is shown in Figure 3-3.

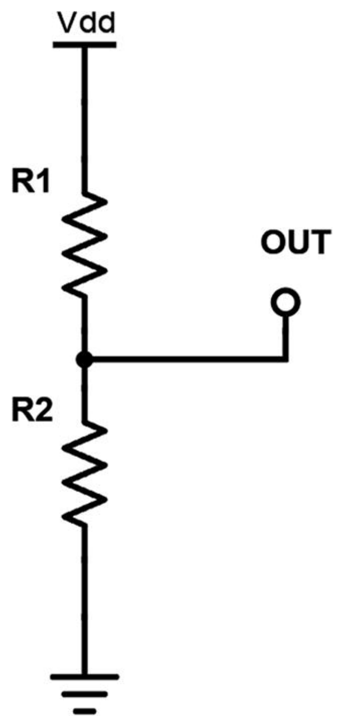


Figure 3-3 The voltage divider circuit

A voltage divider is a simple circuit arrangement that, as we see in Figure 3-3, consists of two resistors connected in series across a voltage source. What a voltage divider does is it takes an input voltage and divides it into a lower output voltage. This is done by using the ratio of two resistors. Vdd is the input voltage, and where R1 and R2 meet, that point is the output voltage. With a voltage divider, the voltage across each resistor in series is proportional to its resistance. This simple circuit forms the basis for a wide variety of applications. As we move throughout the book, you will see how important the voltage divider is in constructing all sorts of useful circuits.

Potentiometer

A potentiometer is an electronic component used to vary the amount of resistance in a circuit. The potentiometer is also known as a “pot” and contains three terminals. A pot is nothing more than a voltage divider that the user can adjust. A rheostat is another device you may encounter, and it is simply an adjustable resistor. The two-axis joystick commonly found in game controllers and volume adjust buttons are common real-world applications of potentiometers. Figure 3-4 shows the potentiometer schematic symbol, and Figure 3-5 shows an actual potentiometer.

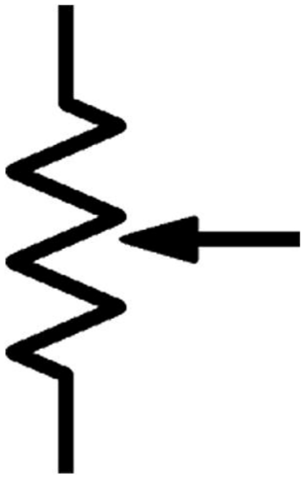


Figure 3-4 Variable resistor schematic symbol

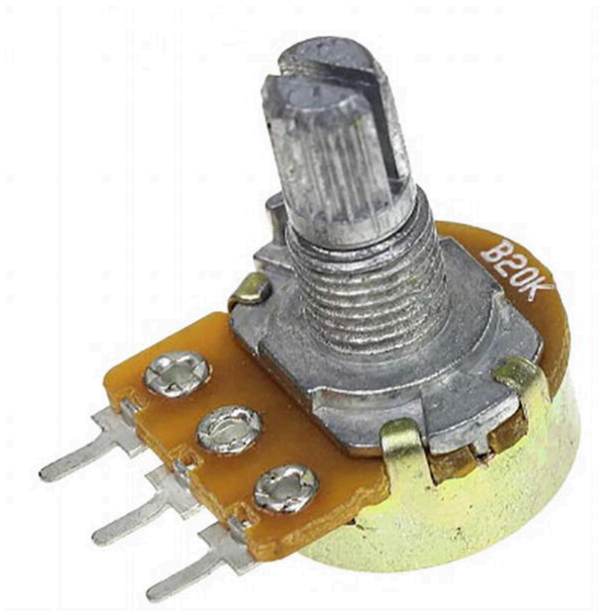


Figure 3-5 A potentiometer

Another type of variable resistor you will encounter is the trimpot or trimmer potentiometer. While potentiometers are designed to be adjusted quite frequently by the end user, a trimmer potentiometer is designed in such a way that once we set the resistance, we usually do not adjust it unless some change in the circuit makes it necessary for us to do so. We usually adjust the trimpot with a screwdriver. Some potentiometers instead of being rotary can also be made to slide.

However, if we consider the mechanism of their operation, all potentiometers from the viewpoint of the microcontroller remain the same; thus, once you can interface a potentiometer, you can interface any other device.

Digital Potentiometer

Though it's a more complex circuit, a type of potentiometer that exists is a digital potentiometer or digipot. Electronically speaking, it performs the same functions as a potentiometer. The advantage of the digipot is that microcontrollers can adjust their resistance using some digital interface protocol using the software.

Since they can be controlled via software unlike their mechanical counterparts, it is possible to adjust the value in ways other than a linear fashion (typically in a logarithmic fashion). This gives digipots additional applications such as scaling and trimming of analog signals. The digipot has a different appearance than a regular potentiometer, and they are packaged to look just like any other IC. The MCP4131 digipot that we'll be using in our projects is shown in Figure 3-6. Figure 3-7 shows the digipot schematic symbol.

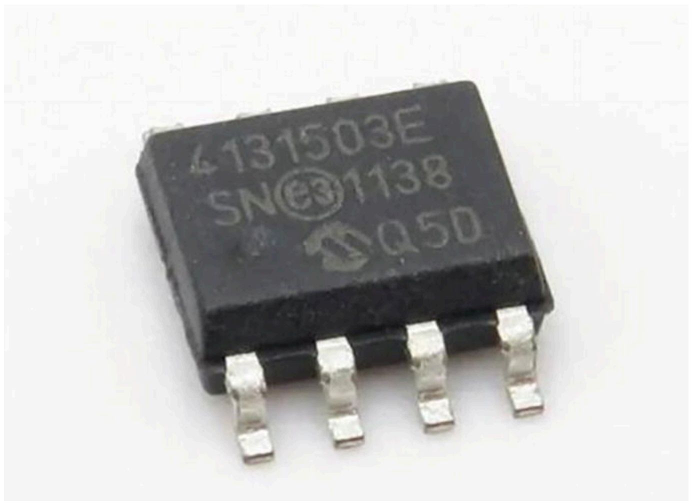


Figure 3-6 MCP4131 digital potentiometer

We can see a schematic symbol for the MCP4131.

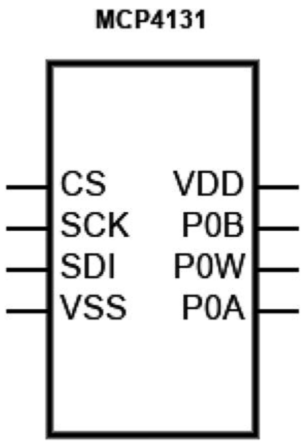


Figure 3-7 Digital potentiometer schematic symbol

Another common digipot family you will encounter is the MCP4x010 devices from Microchip Technology. Their operation is similar to the MCP4131, and once you can use the MCP4131, using these devices will be rather easy.

Photoresistors

The other type of resistor we can look at is photoresistors, also called light-dependent resistors (LDRs) or photocells; the schematic symbol for the device is given in Figure 3-8, and the actual device is shown in Figure 3-9. These are a special type of resistor whose resistance changes in response to the incident light intensity. What this means is that though they are resistors, their resistance will vary based on the amount of light they are exposed to, making them great for both light and dark detection.

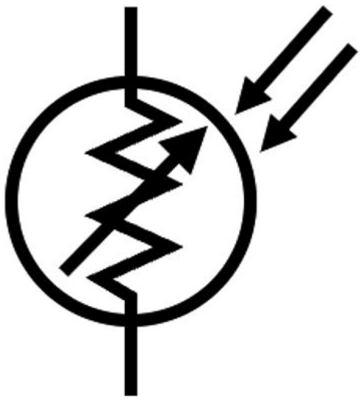


Figure 3-8 Photoresistor schematic symbol

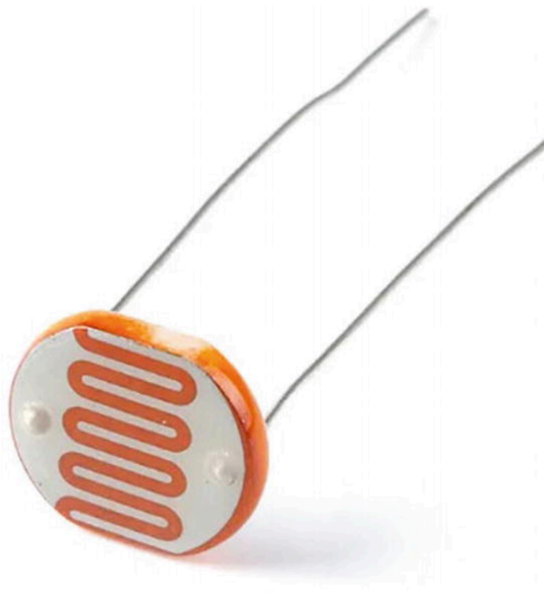


Figure 3-9 A typical photoresistor

Photoresistors used to be made from cadmium sulfide (CdS) and cadmium selenide (CdSe), but some countries have banned cadmium so different materials can be used. Most cheap cells that you buy will, however, be made from cadmium compounds. These devices are intuitively used for light-sensing applications. For example, automatic street lights and security cells detect the dark with "photocells," which usually just consist of a photoresistor in an appropriate housing.

When using these devices, you should know that photoresistors have slower response times compared to other light sensors, limiting their use in applications requiring rapid light level changes. Though they are simple to use and cost-effective, they are less precise than other light sensors, and like other resistors, they are affected by temperature changes. They are, however, great for learning and experimentation, and their low cost makes them attractive for a lot of applications. If you need faster response times, you would typically use a photodiode or photoresistor, which we will look at in subsequent sections.

Capacitors

Capacitors are another passive component that we will look at. They are used to store electrical energy in an electronic circuit. Capacitors come in axial, radial, and SMT packages. They consist of two metal plates separated by an insulator, called the dielectric. This dielectric material is made of many materials, including paper, ceramic, plastic, and even air. Capacitance is measured in farads (F), although microfarads and picofarads are the commonly used units of measurement in everyday usage.

The type of dielectric influences the properties of the capacitor and determines if the capacitor is polarized or nonpolarized. Figure 3-10 shows the polarized cap schematic symbols, and Figure 3-11 shows the nonpolarized cap schematic symbols.

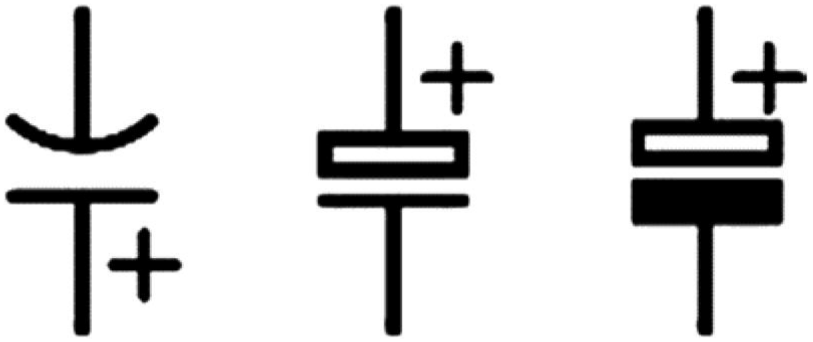


Figure 3-10 Polarized capacitor symbols

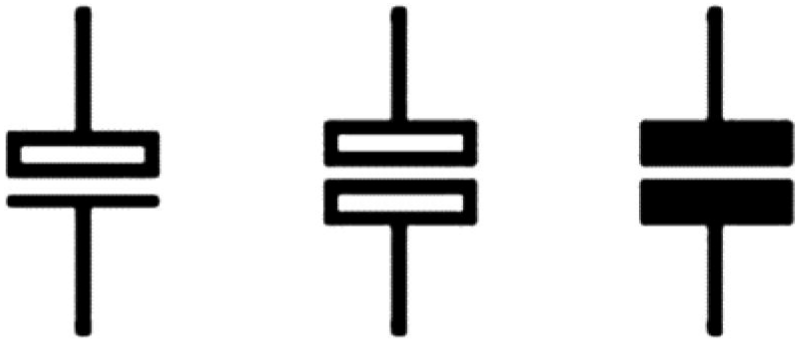


Figure 3-11 Nonpolarized capacitor symbols

The most commonly encountered capacitor is the electrolytic capacitor. This is because they store a relatively large capacitance relative to their size. They are polarized, and care must be taken not to connect them backward. They come in two varieties: tantalum and aluminum. Aluminum capacitors are easily recognizable since they usually come in cylindrical tin cans. Tantalum capacitors have a higher capacitance-to-weight ratio than aluminum capacitors and are usually more expensive. The aluminum capacitor is shown in Figure 3-12.

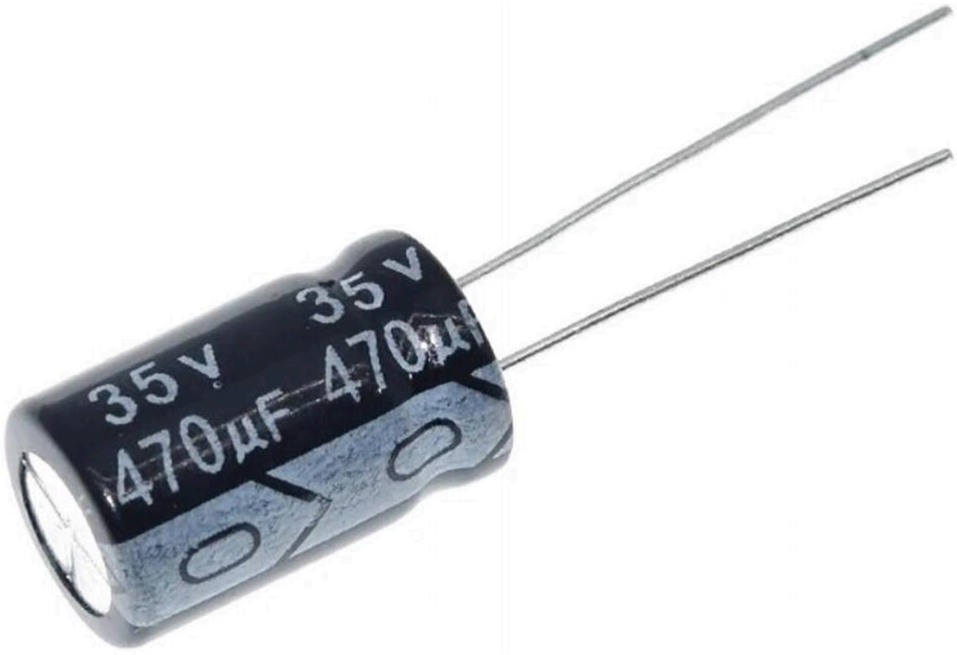


Figure 3-12 Aluminum capacitor

Ceramic capacitors are another commonly encountered type of capacitor in embedded systems design. Unlike electrolytic capacitors, they have the advantage of not being polarized. However, they have a lower capacitance. We see the ceramic capacitor in Figure 3-13.

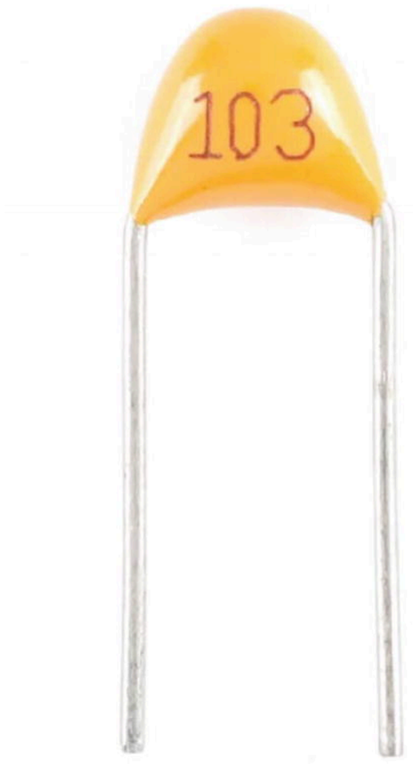


Figure 3-13 Ceramic capacitor

The most prominent use of capacitors in the embedded space is in filtering the output of power supplies. Many microcontrollers require filtering capacitors on their power pins. Decoupling capacitors act as a temporary voltage source for microcontrollers and are very important in suppressing high-frequency noise on the power supply. When used in this way, decoupling capacitors are also known as bypass capacitors since they bypass the power supply. It is important to consult the datasheet to determine the value of bypass capacitors you should use.

There are many occasions where intermittent problems in your circuits can be traced to having a noisy power supply. A power supply is noisy when there are ripples on the power rail. These ripples are essentially fluctuations in the supply voltage. If you look at your DC output from a power supply with an oscilloscope, you will notice these ripples. If they are too large, they can cause a lot of problems in your circuit and may lead to undesired operations and, in some cases, damage to the IC and other sensitive electronics.

Inductors

Inductors are used to resist changes in the electric current flowing through them. The most common use of inductors is in filters as an inductor passes low-frequency signals and resists high-frequency ones. The schematic symbol for the inductor is given in Figure 3-14.

The henry (H) is used to measure inductance. The nanohenry, microhenry, and millihenry are the most commonly encountered units. Inductors are based on a simple principle. A voltage is generated when a conductor is introduced to a magnetic field. The inductor makes use of this property of the voltage being generated by a changing magnetic field in either the same conductor or another conductor under the inductor's influence.

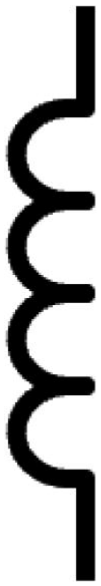


Figure 3-14 Inductor schematic symbol

You will observe that most inductors consist of wires that are wound into a coil. This is because the amount of voltage that can be generated is increased when the wire is wound into a coil. We can further cause the concentration of the magnetic field by using a magnetic core. Usually, we use a soft iron core or ferrite rod. These soft cores don't make the magnetic fields stronger but concentrate and guide the lines of magnetic flux, changing the characteristics of the field rather than its strength.

Another unit you may come across when dealing with inductors is the property of self-inductance L, which determines the ability of the inductor to generate an electromotive force (EMF) in response to changes in the current flowing through it. You see the Greek letter phi Φ being used to represent magnetic flux in equations. And a formula you see is

$$\Phi = B \times A \times \cos(\theta)$$

where B is the magnetic field strength perpendicular to the surface, A is the area of the surface, and theta is the angle between the magnetic field and the normal to the surface. In simple terms, it's a way to mathematically describe how much a magnetic field affects a surface, taking into account the size of the surface and the angle at which the magnetic field hits it.

You should also be aware of Faraday's law of electromagnetic induction, which states that a change in the magnetic flux through a loop induces an electromotive force (EMF or essentially voltage) in that loop, which is a formal way of putting what we said before. These laws are great for gaining an intuitive understanding, but for most practical applications, you would usually use inductors as part of a circuit or module that was already designed. The art of making your inductors, however, is a skill that you can look at, especially if you are into building RF and radio experimentation.

Transformers

A transformer is a device used to step up or down voltages in electronic devices. Transformers require an alternating current to operate. Figure 3-15 shows the transformer schematic symbols, and Figure 3-16 shows an actual transformer.

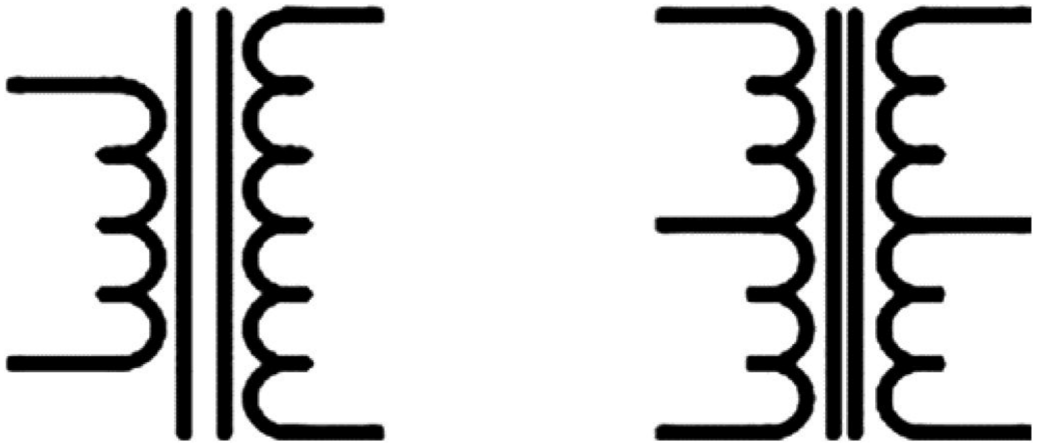


Figure 3-15 Transformer schematic symbols

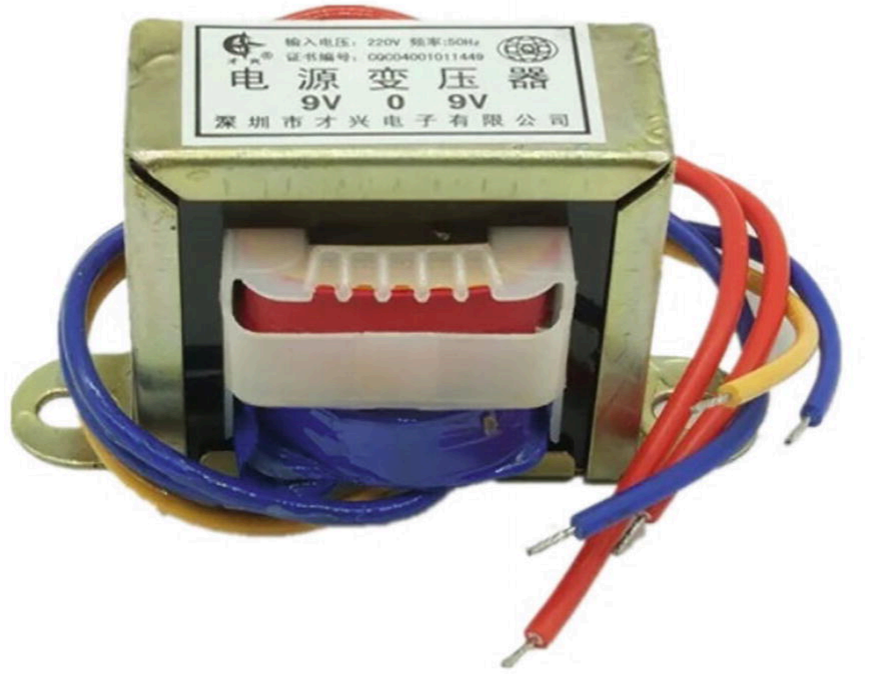


Figure 3-16 A center-tapped transformer

Transformers make use of something we call mutual induction. This is a phenomenon where a changing magnetic field produced by one coil causes a voltage in a nearby conductor. In such a setup of mutual induction, one coil that produces the changing magnetic field is termed the primary coil, and the coil experiencing the induced voltage is called the secondary coil.

Within a transformer, the two coils are usually wound around a shared core, which can be used to transfer electrical energy from one circuit to another one without any direct electrical connection. The primary coil generates a varying magnetic field in the core, which causes a voltage to be induced in the secondary coil. More turns in the coils of the transformer lead to a higher mutual inductance, increasing the induced voltage.

Many times, you will encounter transformers that have multiple taps. A tap is the name we give to a specific point on the winding where a connection is made. It allows for the adjustment of the turns, ratio, and voltage ratio of the transformer. Center-tapped transformers are commonly encountered, especially if you are into audio systems design and designing linear power supplies. The center-tapped transformer has a midpoint in the winding that causes the coil to be split into two halves. This allows the halves to be used independently, allowing for various applications. For example, in voltage regulation, center-tapped transformers are used in power supplies to produce a dual-voltage output, and when doing full wave rectification, the center tap helps in conducting current in both halves of the AC cycle, making it ideal for building linear power supplies.

RC, RL, RLC, and Tank Circuits

We can combine the resistors, capacitors, and inductors to create complex networks that produce unique properties based on their combined behavior. Usually clearing up these component combinations is explained with complex mathematics and can be very confusing to a beginner. While designing such circuits can become rather complex, the principles of their operation can be explained rather simply.

Capacitor-Resistor combinations are called RC circuits, and series RC circuits have characteristics such as time-dependent charging and discharging. The Resistor-Inductor combinations are called RL circuits, and RL circuits have a resistor connected to an inductor in series, have energy storage using magnetic fields, and give transient responses during changes in current.

The magic happens when we combine Resistor (R), Inductor (L), and Capacitor (C) components and create a class of circuits known as RLC circuits. These devices exhibit a property called resonance. Resonance is the name given to the frequency at which these devices interact so that the impedance across the circuit is at its minimum or maximum value. When we talk about impedance, we are referring to the opposition offered by a circuit to the flow of current that is alternating (AC); we represent this property with a symbol “Z”, and it is comprised of this opposition given by both resistance and reactance. Reactance “X” arises from reactive components opposing the changes in current flow in an AC circuit.

So in short, reactance addresses the opposition offered by reactive components, while impedance encompasses both resistance and reactance.

A circuit you are bound to encounter at some point is a “tank circuit”; formally we can refer to it as an “LC” circuit, which is a specific RLC circuit with notable properties. The “tank circuit” derives its name from its ability to store and exchange energy, like a tank filled with electrical charge. We see a typical LC circuit in Figure 3-17.

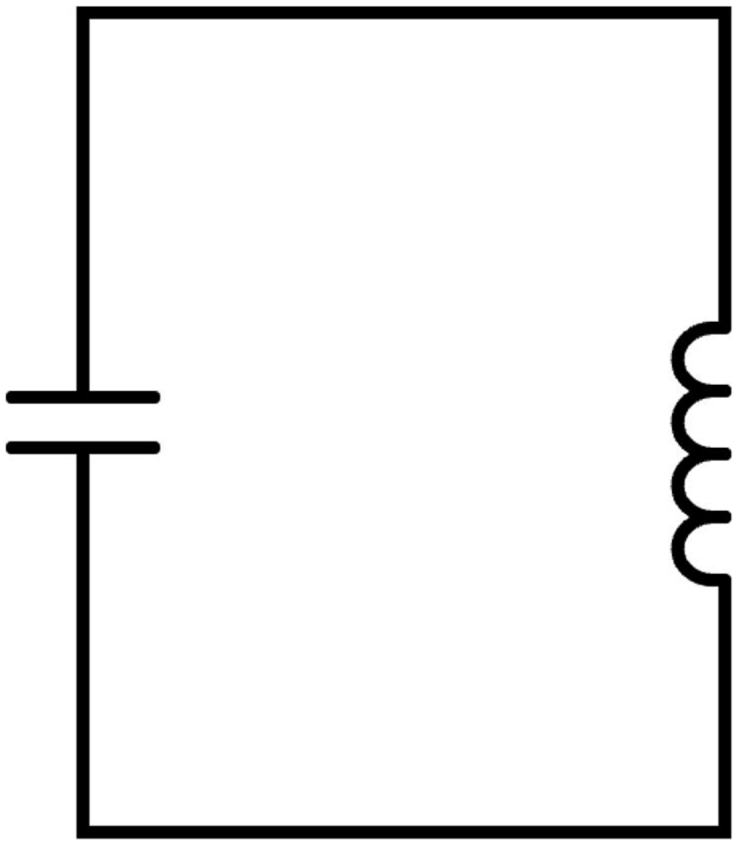


Figure 3-17 LC tank circuit

The LC circuit allows for a unique interplay of stored magnetic and electric energy, creating an oscillator. If you ever had the pleasure of designing your radio circuits, you would have come across this configuration. RF tuning and signal filtering are the two applications you will be sure to come across the LC tank circuit; a lot of RF applications involve the use of the LC tank circuit.

Diode

Now that we have an understanding of the three fundamental passive devices—resistors, capacitors, and inductors—we move on to semiconductor devices. The first semiconductor device we will look at is the diode. A diode is a device used to allow current to flow in a particular direction. When the diode is forward biased, current can flow. When the diode is reverse biased, the current cannot flow. If a certain voltage is applied in the reverse direction, the diode will break down and allow current to flow in the opposite direction. Diodes are extremely important in embedded devices, as they are imperative to suppressing voltage spikes that can be present when driving inductive loads. Figure 3-18 shows the diode schematic symbol, and in Figure 3-19, we see the diode.

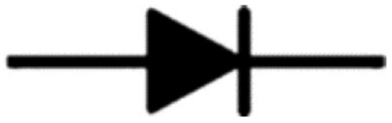


Figure 3-18 Diode schematic symbol

Figure 3-19 A silicon diode

The diode can allow current to flow in only one direction because it has a p-n junction, which controls the direction of current flow. A p-n junction is a boundary formed between a p-type semiconductor, which has an excess of positive charge carriers, and an n-type semiconductor, which has an excess of negative charge carriers. When these two types of semiconductors are brought together, electrons from the n-type region move to the p-type region, creating a depletion zone with no charge carriers. This depletion zone acts as a barrier, establishing a potential difference. The application of a forward voltage reduces the barrier, allowing current flow, while reverse voltage increases the barrier, inhibiting current flow. p-n junctions serve a crucial role in electronic components like diodes. We will also look at how other semiconductor devices exploit the properties of p-n junctions later in this chapter.

When the diode is forward biased, the p-n junction offers very little resistance, and when it is reverse biased, the p-n junction offers little resistance. For a forward-biased diode to allow current to flow, a certain voltage we call a “knee” voltage must be exceeded. This knee voltage is like the turn-on voltage of the diode. In the diode’s forward-biased condition, the voltage across the diode increases until it reaches the knee voltage. When the diode is operated beyond this knee value, a small increase in the forward voltage causes a large increase in the current, thus allowing the diode to conduct current more easily. This knee voltage is around 0.7 volts for common silicon diodes.

Zener Diodes

Zener diodes are devices that operate in the breakdown voltage region of the diode and are used for voltage stabilization, voltage regulation, and as a voltage reference. Figure 3-20 shows the Zener diode schematic symbol. In Figure 3-21, you see the Zener diode. You must be aware that regular diodes may also have glass bodies.

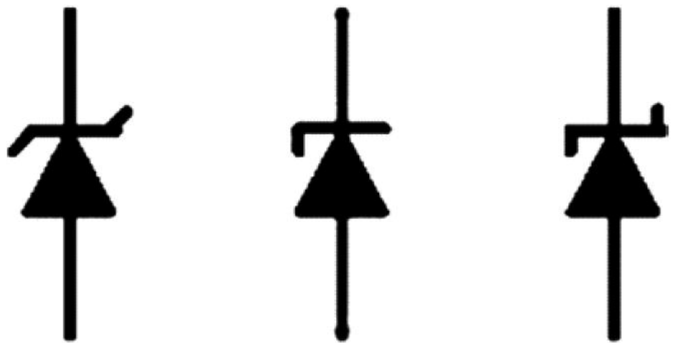


Figure 3-20 Zener diode schematic

In the last section, we talked about knee voltages for forward bias in a diode turning on. We also mentioned that there is also the breakdown voltage, which is the maximum reverse voltage a diode can withstand before it allows a significant reverse current to flow (there is always a small leakage current, but it is usually negligible for practical applications).

This breakdown voltage is exploited by Zener diodes. Zener diodes have differing semiconductor doping done to them, and depending on how heavily the diode is doped, the breakdown voltage can be manipulated, and we exploit this property.

Figure 3-21 Zener diode

Light-Emitting Diode

A light-emitting diode (LED) is a type of diode that emits light when a voltage is applied to it. Diodes come in a variety of colors and sizes. The types encountered are infrared, red, orange, yellow, green, blue, violet, purple, pink, ultraviolet, and white. There are also bi-color LEDs and RGB LEDs. There are surface-mount LEDs and of course standard 3 mm and 5 mm LEDs that are used in most projects. Seven-segment, sixteen-segment, and dot-matrix LEDs are also used in a variety of projects.

Figure 3-22 shows the LED schematic symbols, and Figure 3-23 shows an LED.

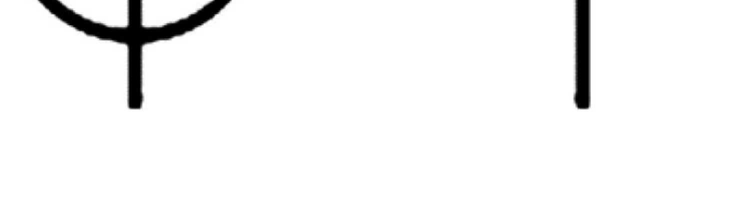


Figure 3-22 LED schematic symbols

Figure 3-23 An LED

LEDs also have a knee voltage point that allows for significant current flow. These LEDs are very energy efficient, producing very little heat, and have long lifespans. This efficiency has led to LEDs becoming the preferred method of illumination in modern society. LEDs have replaced the majority of neon, incandescent, halogen, and compact fluorescent lamps for lighting.

Laser Diodes

Laser diodes are another type of diode you are likely to encounter. Figure 3-24 shows us the typical laser diode. Laser diodes produce coherent, intense beams of light through stimulated emission of photons. They are similar to LEDs, but the light they produce is more concentrated.

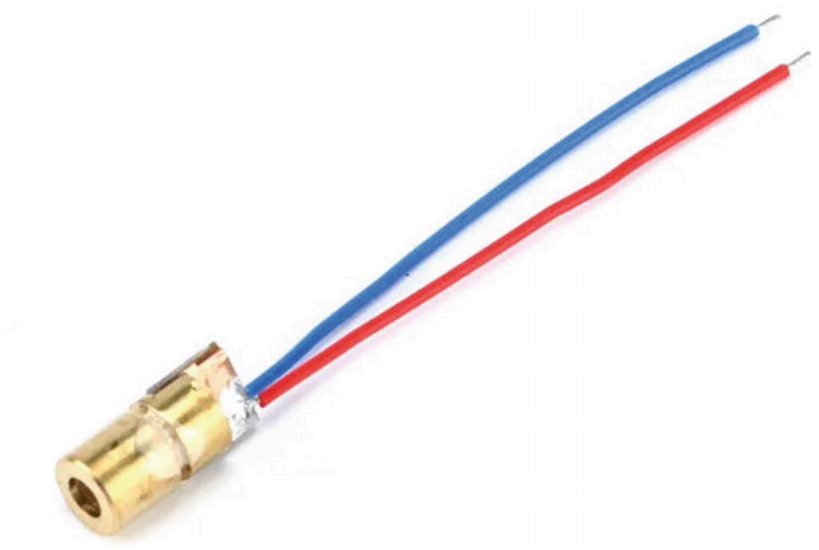


Figure 3-24 A laser diode

Laser diodes are very low cost and weigh very little, making them useful for a variety of projects.

Transistors

Transistors are arguably the most revolutionary devices ever invented. Modern devices would not be possible without the transistor. Transistors are mainly used for switching and rectification in electronic circuits. Transistors come in a variety of types. One of the most fundamental types of transistors is the bipolar junction transistor, which we will discuss. We will also talk about field effect transistors (FETs) including metal oxide semiconductor FETs (MOSFETs), and we will also cover insulated gate bipolar transistors (IGBTs). While silicon transistors are prevalent and are the type we will focus on in this chapter, you should be aware of emerging types such as organic and quantum-dot transistors.

Bipolar Junction Transistors

The first transistor type we will look at is the bipolar junction transistor or BJT. The BJT comes in two varieties and may be either NPN- or PNP based. These names come from the designation of the semiconductor material of which it is constructed. A semiconductor with extra electrons is of the N-type variety, and one with fewer electrons is of the

P-type variety. If that semiconductor is stacked in the order of N-type, P-type, and N-type, then you get the NPN variety. Similarly, if it is stacked in the order of P-type, N-type, and P-type, then the PNP variety is created.

The two types of transistors can be differentiated by the direction of the arrow on the emitter pin in schematic drawings. The PNP-type transistor has the arrow pointing inward, while the NPN variety has the arrow pointing outward (see Figure 3-25).

Transistors are three-pin devices: the collector (C), the base (B), and the emitter (E). The transistor is used extensively for signal amplifying and electronic switching. When used for amplification, the transistors convert a low-power signal into one of higher power. The name given to the type of transistor amplifier is determined by the pin into which the signal to be amplified enters and exits.

The most common type of amplifier is the common-emitter type amplifier. In this mode, the emitter is tied to the ground, the signal entry point is the base, and the exit point is the collector. This type of amplifier is commonly used to amplify audio signals since it performs voltage amplification.

The common collector is the other type of amplifier configuration of the transistor. In this mode, the collector is connected to the ground, and the signal enters the base and exits the emitter. This type of amplifier is used for voltage buffering and current amplification.

The final type of amplifier we will look at is the common-base configuration, which is rarely used in practice. The base is connected to the ground with the emitter as the input and the collector as the output. It has applications as a current buffer.

Figure 3-25 shows the transistor schematic symbols, and Figure 3-26 shows the common TO-92 transistor package.

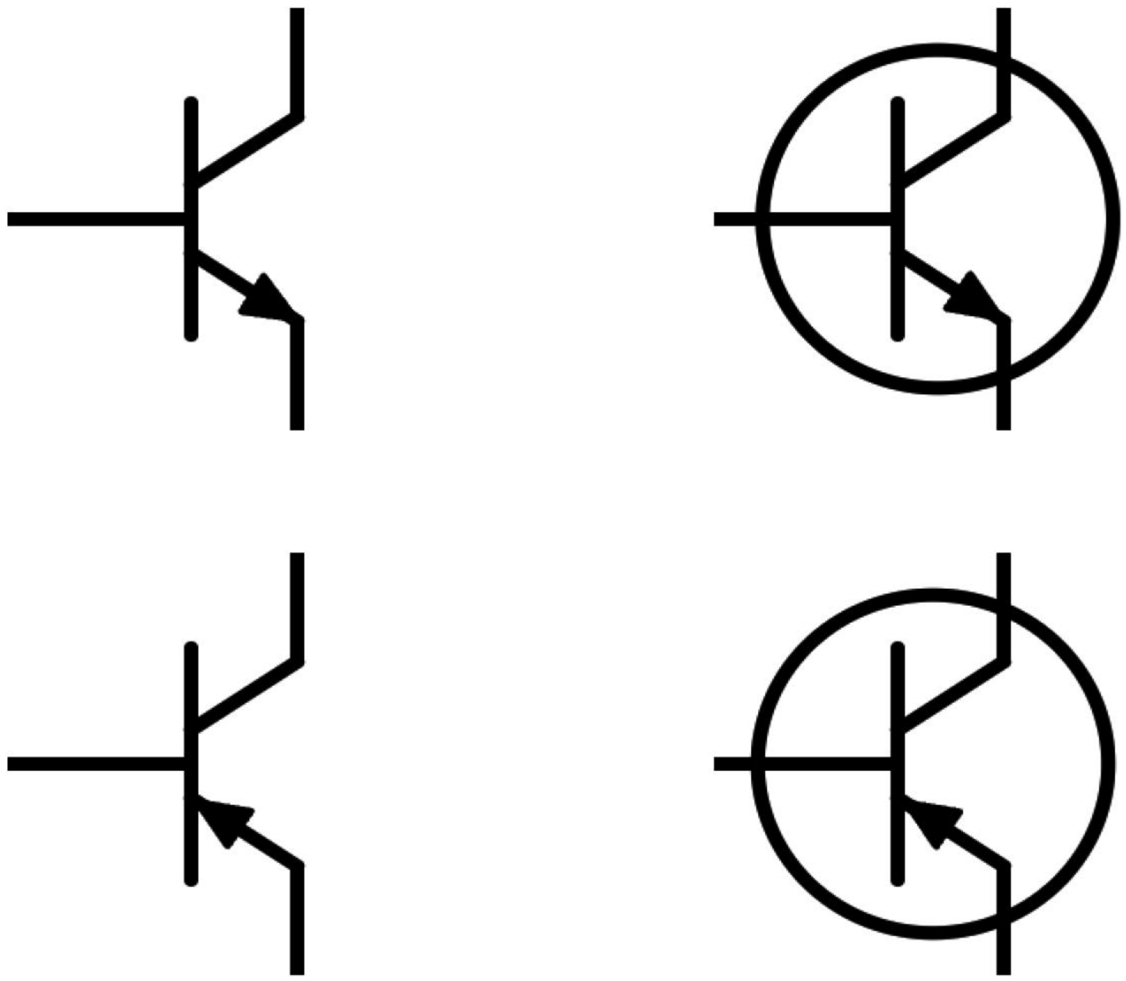


Figure 3-25 Transistor schematic symbols

If you have worked with transistors before, you would have worked with small signal or low-power transistors. However, there are also power transistors that are designed to handle higher power loads than the small signal transistors. Once you start working with higher-power transistors, they generate heat that we need to manage through the use of heat sinks and other cooling mechanisms.

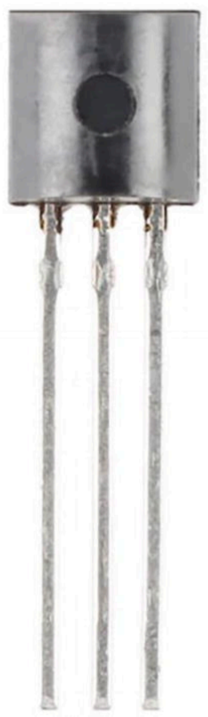


Figure 3-26 Transistor in the TO-92 package

The most popular NPN transistor you will encounter in designs is the 2N2222A device, which can cover any use case where you need a small transistor. The 2N3904 and 2N3906 are a popular pair as well; generally, if I am designing an application that needs small transistors, I start with these for proof of concept. The 2N3055, which is commonly used in power supplies and audio amplifiers, is a great device when you want a bit more power handling capability. The TIP31C is also a good general-purpose power NPN transistor that can be used in power switches and linear amplifiers. If you want a PNP device equivalent to the TIP31C, then the TIP32C is a general-purpose PNP complement to the TIP31C.

Darlington Transistor

A Darlington transistor consists of two transistors connected in such a manner that the current output of the first transistor is further amplified by the second one. The Darlington pair uses two PNP or two NPN transistors, and a complementary Darlington uses one NPN and one PNP transistor.

They act as a single transistor with a high current gain. This property is important in embedded applications, as microcontroller-based circuits can use a small amount of current from the microcontroller to run a larger load. This gives them many uses, such as display drivers and control of motors and solenoids.

Another Darlington transistor that you may have to use is the Photodarlington transistor. This transistor consists of two transistors just like a regular Darlington. However, they differ in that the first transistor acts as a photodetector, and its emitter is coupled with the base of the second transistor. Figure 3-27 shows the Darlington transistor schematic symbols.

The Photodarlington has a high gain but is slower than ordinary phototransistors.

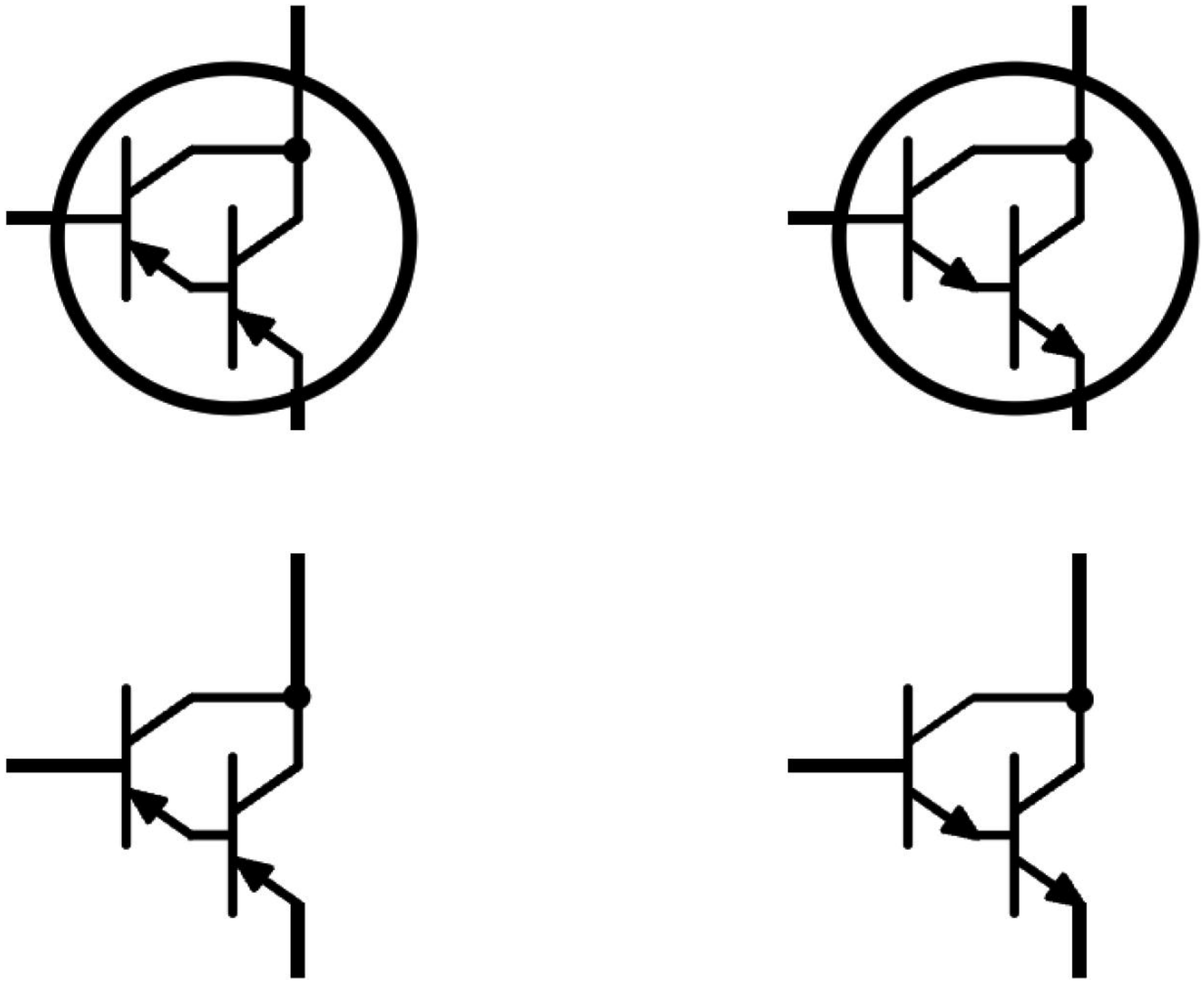


Figure 3-27 Darlington transistor schematic symbols

The ULN2003, which will be used to drive small stepper motors in this book, consists of an array of Darlington transistors. We will discuss this device in a later chapter in the book.

Field Effect Transistor

The field effect transistor or FET is another semiconductor device we need to look at. It is also used in circuits to control the flow of electrical current. FETs can be N-channel or P-channel, and they operate similarly to bipolar transistors. The two types of FETs we will concern ourselves with are the MOSFET and the JFET, which we will discuss in the subsequent sections.

Metal Oxide Semiconductor Field Effect Transistor (MOSFET)

The transistor was the pioneer of modern digital electronics. However, as time progressed, the metal oxide semiconductor field effect transistor (MOSFET) has taken over a lot of applications of the transistor and is used for amplification and switching in modern integrated circuits.

The MOSFET consists of three pins—Gate (G), Source (S), and Drain (D)—which are the equivalent of the base, emitter, and collector, respectively, of the transistor. There is also a fourth pin called the body or substrate, but it’s usually internally connected. MOSFETs come in the N-channel and P-channel varieties. MOSFETs have a major advantage over BJTs, as they require less voltage to turn on. Thus, while transistors are current-based devices, MOSFETs are voltage based. MOSFETs must be handled carefully, as they are easily damaged by static electricity. Figure 3-28 shows the MOSFET schematic symbols.

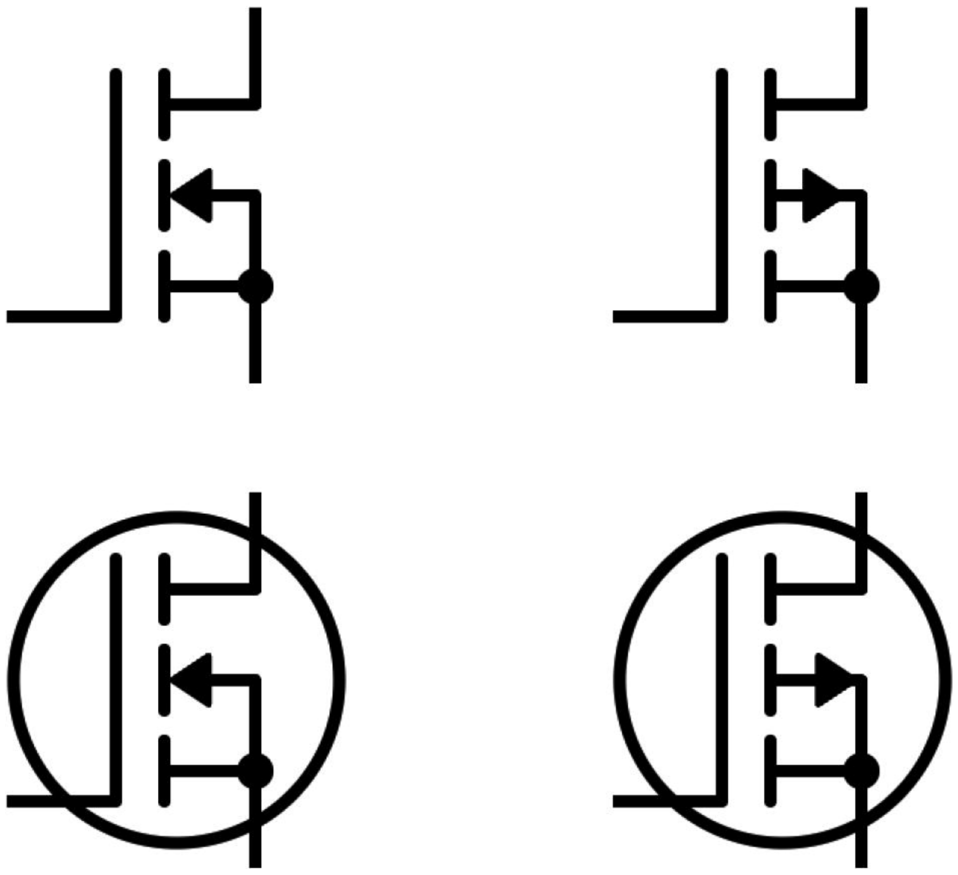


Figure 3-28 MOSFET schematic symbols

MOSFETs are great for controlling high currents; we can put them in parallel, but we also need to use gate drivers for them.

Junction Field Effect Transistor

The junction field effect transistor (JFET) is used for switching, amplification, and as a voltage-controlled resistor. JFETs are not commonly used in normal circuit design but do find use in specialty analog circuits. BJTs or MOSFETs can do most of what is required. The JFET also finds use as a voltage-controlled switch and as a chopper. Figure 3-29 shows the JFET schematic symbol.

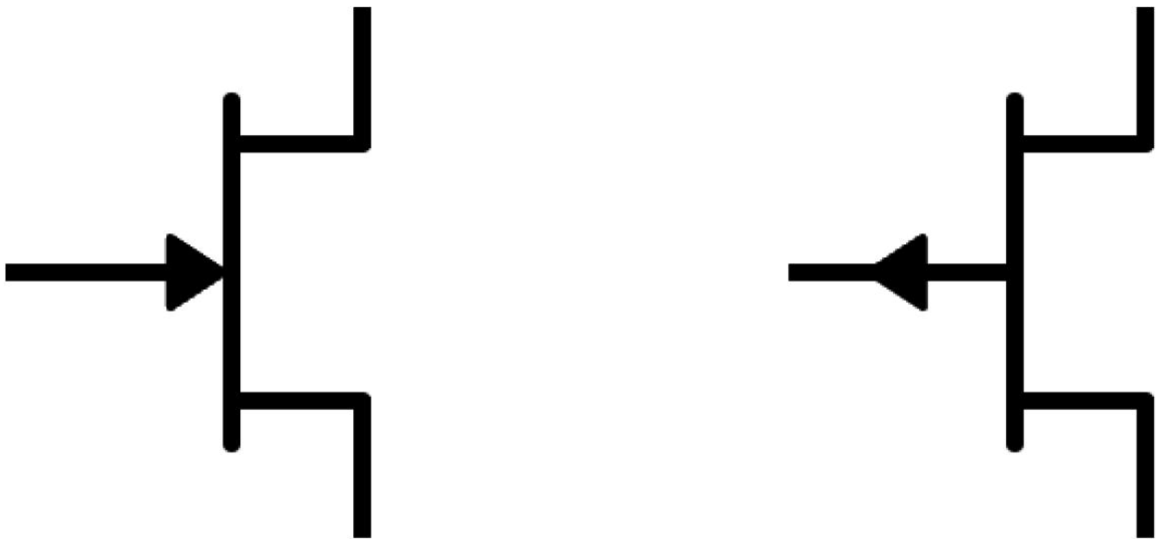


Figure 3-29 JFET schematic symbol

JFETs tend to consume more power than MOSFETs and are typically more voltage-sensitive, which means you must make careful design considerations to avoid damaging them if the voltage limits are exceeded and require careful design considerations. I have rarely needed to use discrete JEFTs in my designs though I have used them integrated into operational amplifier packages that use a JFET on the front end. We will discuss operational amplifiers in the succeeding section.

Operational Amplifiers

The operational amplifier or op-amp is one of the fundamental building blocks of analog electronics. I would even go so far as to say that an op-amp is to analog electronics what a transistor is to digital ones. After you learn about microcontroller technology, I recommend you take an in-depth look at op-amps. With knowledge of op-amps and microcontrollers, you can design very powerful embedded systems. Figure 3-30 shows the op-amp schematic symbol.

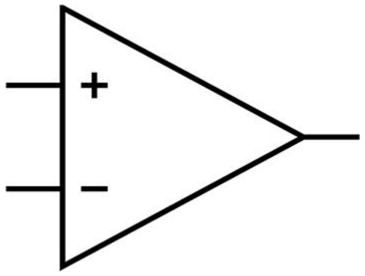


Figure 3-30 Operational amplifier Symbol

As the name implies, the op-amp is used for DC signal amplification. It is also used to filter and condition signals as well as for integration, differentiation, subtraction, and addition.

When looking at op-amp schematic symbols, in addition to the power supply pins (which are usually omitted), you will see two terminals: one with the minus sign and the other with the positive sign. The input with the positive sign is known as the non-inverting input, and the one with the minus sign is called the inverting input. There is a third pin at the vertex of the triangular-shaped op-amp symbol, known as the output port, and this pin can allow a voltage or current to flow into the device, called sinking, or to supply current from the device, called sourcing.

Some common applications of op-amps in embedded systems design are as a buffer for the output of voltage divider voltage references, instrumentation amplifiers for differential pairs, active low-pass and high-pass filters, photodiode amplification, and more. An entire book can be written on op-amps and their applications!

There are hundreds of op-amps to choose from, and I recommend you prototype with the TL081CP, KIA324, MCP6001, and MCP6002. These op-amps are great for rapid prototyping. Once you have a working system, you can determine the best amplifier for your needs. Some microcontroller devices have op-amps built in, which can reduce the need to have a discrete device in your design.

Understanding Waveforms

To better understand the importance of op-amps, we must understand a bit about waveforms. If we take a steady voltage and vary it over some time, we get a waveform. We see waves occurring in nature all the time. We see waves in the ocean, and of course, we know we have the sense of hearing due to sound waves. The lines of flux we were discussing in inductors can be made into a generator that produces a sine wave when we have an inductor (a coil) rotating in a magnetic field. The sine wave is shown in Figure 3-31.

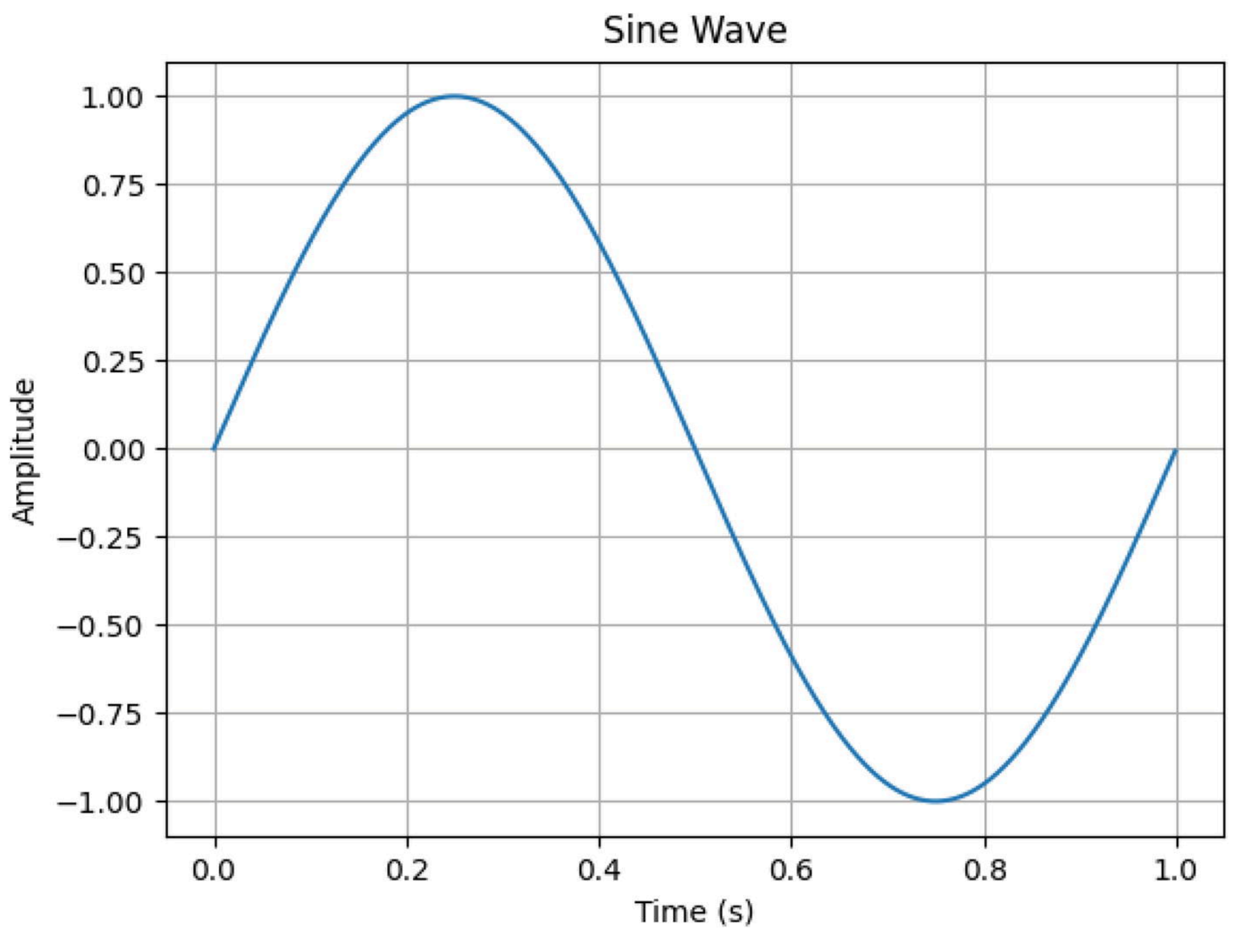


Figure 3-31 Sine wave graph

Besides a sine wave, another type of waveform you will encounter is the square wave. A square wave is produced when we take a voltage and switch it on and off; this type of cyclical switching produces a wave that is symmetrical and is very useful for all types of electronic devices. The square wave as we will learn later on is the heartbeat of digital circuits including microcontroller systems. We see the square wave graph in Figure 3-32.

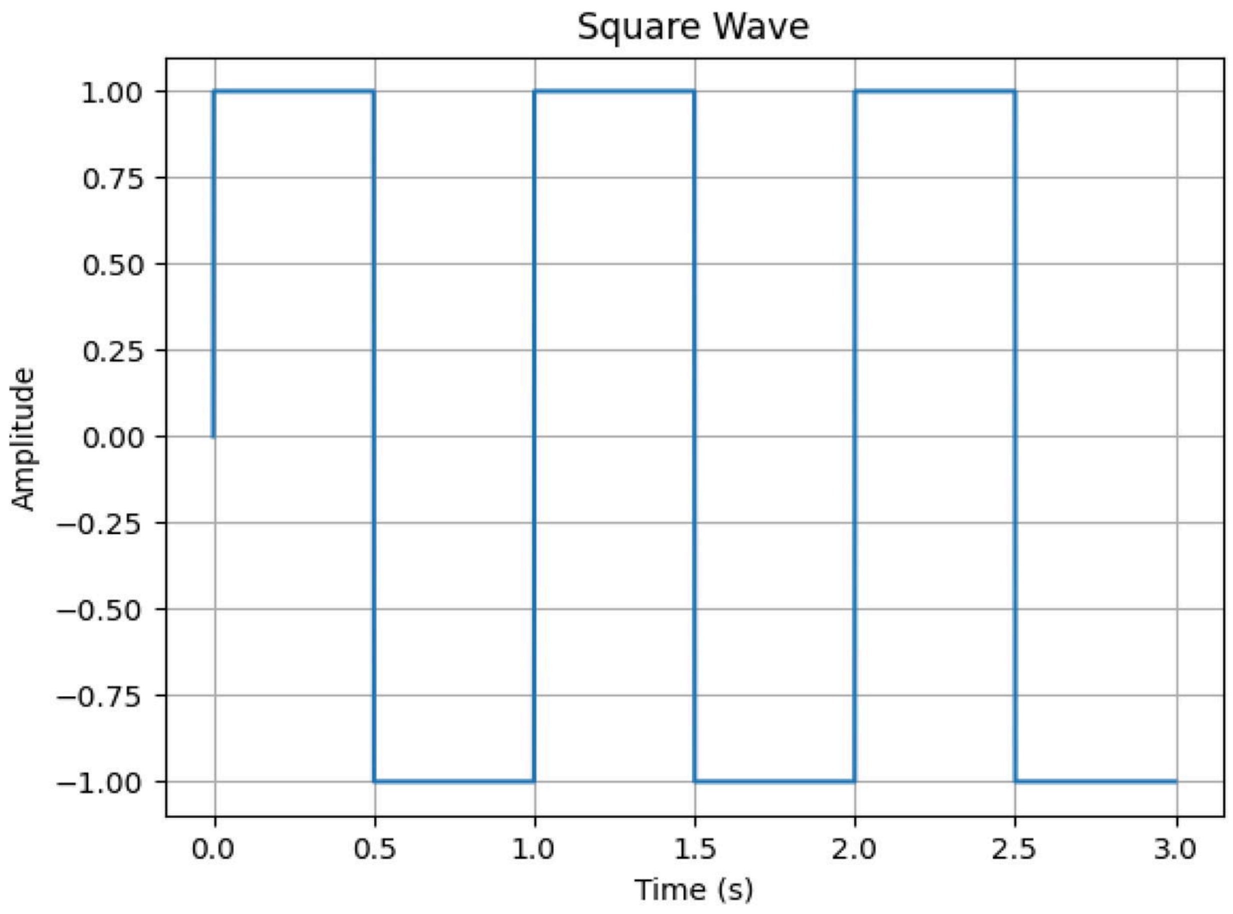


Figure 3-32 Square wave graph

Regardless of the type of waveform that is produced, all waveforms have an amplitude, which is the peak-to-peak distance we measure from the top of one wave to the top of another within a unit of time of what we call the wavelength. Waveforms also have a period, which is the time between one point on a wave to the same point on the next wave. Another measurement we need to consider when dealing with waveforms is the frequency. The frequency is the inverse of the period and is measured in hertz, which is the number of waves per second or cycles per second (in fact, it used to be called cycles per second before). As we progress further in the book, we will see waveforms.

Oscillators

In the last section, we talked about waveforms. Waveforms are generated by an electronic circuit called an oscillator. Oscillators are a circuit configuration that produces a continuous output signal without the need for an input. They utilize positive feedback to sustain oscillations, and we can generate waveforms of various types and frequencies with them. Oscillators work with an amplifying element we call a feedback loop that keeps the oscillation stable. The way a feedback loop works is that a portion of the output signal is fed back into the inputs (something we should be familiar with from op amps). Oscillators also require a resonant circuit, typically an RC or RL circuit that determines the frequency at which the oscillator operates. Active devices like the transistor and op amp we looked at provide amplification within the feedback loop.

Oscillators as we know can produce waveforms, but their stability means they can also be used for the clock or heartbeat of a circuit and they can also be used for signal generation in things like radios and other communications equipment.

There are various oscillator circuits. There are LC oscillators that use the LC resonant "tank circuit" to generate oscillation. Anyone who worked with electronics is typically introduced to the Colpitts, Hartley, and Clapp oscillators based around inductors.

There are also RC oscillators that use an RC network to set the frequency of oscillation. A common oscillator we encounter in this category is the phase-shift oscillator. There are also crystal oscillators, which we will use throughout the book to provide clocking for our microcontroller devices.

We will also encounter special types of oscillators such as Voltage Controlled Oscillators (VCOs) whose frequency is dependent on the input voltage signal. There are also specialized devices like Numerically Controlled Oscillators (NCOs), the operation of which we will discuss in the later chapter.

Integrated Circuits

The op amp is usually in the form of an integrated circuit or IC. This is distinct from the components and circuits we have been looking at so far, which are comprised of discrete devices. ICs are usually in small packages and usually only require a power supply to work. ICs comprise the transistors, diodes, resistors, and capacitors in one package interconnected and fabricated onto a single semiconductor wafer.

An op amp is a type of integrated circuit we call a linear integrated circuit. These are circuits that process analog signals linearly, without the need to alter the waveform. An op amp is used for amplification, though we can get linear ICs that perform a lot of functions including filtering, voltage regulation, signal conditioning, and other functions, some of which require specialized ICs we call Application-Specific Integrated Circuits (ASICs) or can use general ICs like an op amp.

Digital Electronics

The previous sections focused on the branch of electronics known as analog electronics. However, the majority of our signal processing and new electronic devices we have available these days are digital devices. Though we can build digital circuits from discrete components, digital devices typically comprise a complete electronic circuit and like op amps typically are encapsulated on one IC package.

Digital electronics undoubtedly changed the world as we knew it. While analog electronics deals with continuous signals, the contemporary landscape of electronics is predominantly shaped by the realm of digital devices; just look around at the devices surrounding you and you will see the majority, if not all of them, are primarily digital devices. Unlike analog circuits, which manipulate continuous signals, digital circuits process discrete signals represented by binary code—0s and 1s. The shift toward digital electronics has revolutionized signal processing, leading to the proliferation of advanced electronic devices we know today such as smartphones, laptops, and $1 Linux SoCs.

Digital circuits, the foundation of digital devices, can be built using discrete components. However, the trend in modern electronics leans heavily toward integrated circuits (ICs) that encapsulate entire electronic circuits within a single package. This integration allows for compact and efficient digital devices, ranging from microcontrollers to powerful processors, all housed on a single chip. We will begin with simple logic gates and work our way up to more complex circuits.

Logic Gates

At the heart of digital electronics are logic gates. Logic gates are the fundamental building blocks of digital circuits and perform logical operations on one or more binary inputs and produce a single binary output. The evolution of transistors led to the development of logic gates and subsequently digital circuits. You see transistors enabled the miniaturization of electronic systems. These are tiny switches that could control the flow of electrical current, serving as the building blocks for more complex structures.

When we begin combining transistors, eventually we realize that we can combine them in specific configurations that can form the basis of digital computation. These then can become atomic circuit constructs in of themselves, which we call logic gates. Logic gates take one or more binary inputs (0s and 1s) and produce a binary output based on predefined rules rooted in Boolean logic.

Boolean logic is a framework that was developed by a mathematician known as George Boole that provided a framework for expressing logical operations using binary variables. All logic gates manipulate binary signals based on predefined rules implementing basic Boolean logic functions. These logic functions become the foundation for designing digital systems, allowing engineers to construct circuits that could execute various logical functions. The AND, OR, and NOT gates are the primary tools for implementing these logical functions. We will look at these in the consequent section.

The AND Gate

The AND gate is one of the foundational building blocks of digital logic. The AND gate works by treating two logical inputs and outputs as a logical high only if both inputs are high. If only one of the inputs is high, then the output will be a logical low. Figure 3-33 shows the AND gate schematic symbol.

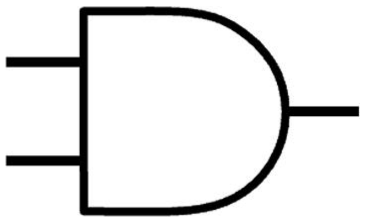


Figure 3-33 AND gate schematic symbol

The OR Gate

The OR gate works by outputting a logical high if either of its inputs is a logical high. If both inputs are a logical high, then the output is also a logical high. The only time the OR gate outputs a logical low is if both inputs are a logical low. Figure 3-34 shows the OR gate schematic symbol.

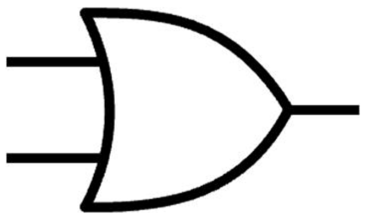


Figure 3-34 OR gate schematic symbol

The NOT Gate

The NOT gate, also known as an inversion gate, produces the exact opposite of its input. If the input is a logical high, then the output will be a logical low, and if the input is a logical low, then the output will be a logical high. Figure 3-35 shows the NOT gate schematic symbol.

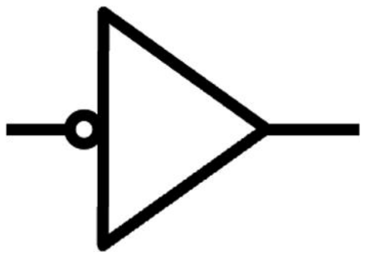


Figure 3-35 NOT gate schematic symbol

The NAND Gate

The NOT AND (NAND) gate is a logical gate that combines a NOT gate and an AND gate. The only distinguishing feature between the NAND gate and the AND gate is the little circle on the end that symbolizes inversion. The NAND gate only gives a logical low if both its inputs are logical highs. Figure 3-36 shows the NAND gate schematic symbol.

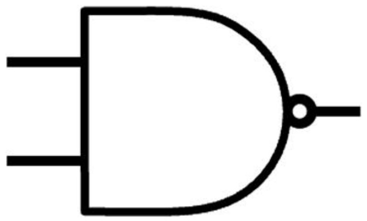


Figure 3-36 NAND gate schematic symbol

The NOR Gate

The NOT OR (NOR) gate is a logic gate that combines a NOT gate and an OR gate. The NOR gate, like the NAND gate, simply inverts the output of an OR gate and has the same distinguishing feature. Figure 3-37 shows the NOR gate schematic symbol.

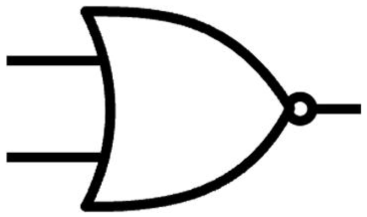


Figure 3-37 NOR gate schematic symbol

The Buffer Gate

The buffer gate is simply two NOT gates combined. The buffer gate might seem useless, but in actuality, it has a lot of applications with logic-level conversion, discussed in the next section. Figure 3-38 shows the buffer gate schematic symbol.

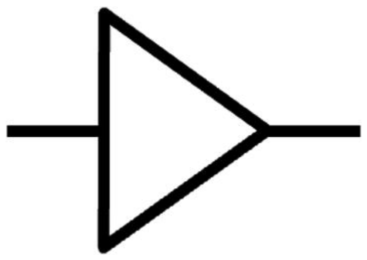


Figure 3-38 Buffer gate schematic symbol

The XOR Gate

The eXclusive OR (XOR) gate is a logic gate that gives a logic low when both inputs are true or when both inputs are false. It gives a logical high when both inputs are logically the opposite. Figure 3-39 shows the XOR gate schematic symbol.

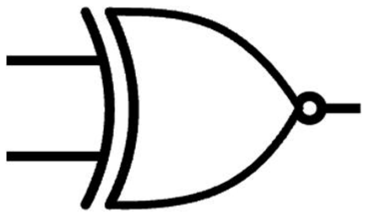


Figure 3-39 XOR gate schematic symbol

Comparators

If I were to think of the simplest way possible to convert analog signals into digital ones, then the use of a comparator would be at the top of that list. Comparators are specialized electronic circuits we use to compare two analog voltage signals and produce a digital output based on the relationship between these signals, essentially acting as a bridge between analog and digital components.

Comparators have two inputs we call the inverting (-) and non-inverting (+) inputs and a digital output that indicates the comparison results. What makes comparators useful is that they can compare continuous voltages and output a binary signal, which indicates whether the non-inverting input voltage is higher or lower than the inverting input voltage.

The comparator checks if one input voltage is greater than or less than the other input voltage. It has threshold detection, which is used to detect when a signal crosses a certain voltage threshold. Comparators have rapid response times and high gain, which allows them to feature high gain to amplify small voltage differences for accurate comparison.

Comparators usually have something called hysteresis, which gives a small amount of positive feedback to prevent output oscillations near the switching point. There is a type of comparator with hysteresis providing noise immunity and stable switching behavior near the threshold. The schematic symbol of the comparator is the same as the op amp. Indeed, it is possible to use an op amp as a comparator; however, if you need to use a comparator for the majority of applications, it's best to use a dedicated device. Comparators are so useful that it is standard on many microcontrollers, and we will explore them further in a later chapter.

Clocked Flip Flops, Latches, and Counters

In the previous sections, we looked at logic gates. In this section, we will look at some of the circuits we get when we combine logic gates. When we combine logic gates we can create synchronous sequential circuits. These circuits are usually driven by some external clocks, and they change states at specific instances determined by the clock signal. The first of such sequential circuits we will look at is the clocked flip flop. The circuit symbol is given in Figure 3-40.

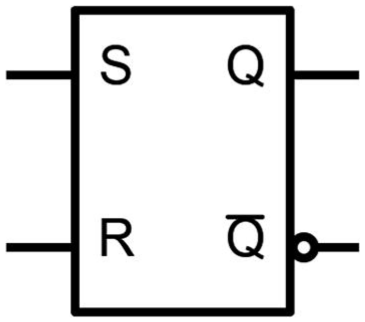


Figure 3-40 Clocked flip flop

This device works as follows. When the output of the circuit Q switches to logic 1, then the Q-bar goes to logic 0. When S (set) is logic 1, then the state is latched even if both S and R (reset) are logic 0. The output states can be changed by applying logic 1 to the Reset terminal. This flip flop acts like a simple 1-bit memory device.

There is also a D-type flip flop, which internally places an inverter between the S and R terminals of the flip flop. The D stands for "Data," and placing this inverter allows us to store either the original or the complement of the input data. There are also JK flip flops that can hold, set, reset, or toggle their state based on their input. D and JK flip flops are also called latches because we can make their outputs be latched to either a logic high or a logic low depending on the input signals.

We can combine flip flops to get a circuit known as a counter, specifically a binary counter. These counters can be used to divide a clock signal by 2, and by combining them, we can use the output of one counter to be the clock of another. If we do this many times over and cascade them, we can get what is known as a divide-by-2 counter, and by cascading multiple counters, we can get a different number of circuits called ripple counters that can be any number of bits like 4, 8, or 16 bits for example. These ripple counters are also called dividers. These counters can also be configured to count up or down and up and down.

Registers and Shift Registers

As we learn more about embedded systems, one word you are sure to see coming up is a register. A register is the name we give to a circuit element that stores a bit of binary data. We can combine several registers and make registers that are several bits wide such as 8-, 16-, or 32-bit registers, for example. Though we usually encounter registers from the context of CPUs, they are not only used in that context. Registers may be used as temporary storage devices for holding data in a digital system. Typically, we use registers when we want to hold data temporarily for processing, moving data, and as a holding place for them before they go on to be processed elsewhere.

For example, a type of register called the data register is used to hold data during arithmetic, logic, and data manipulation operations.

One type of register you will need to familiarize yourself with is the shift register. Shift registers are flip flops connected to allow them to move data either leftward or rightward through the register. These registers may be

- Serial-In Serial-Out (SISO)
- Serial-In Parallel-Out (SIPO)
- Parallel-In Serial-Out (PISO)
- Parallel-In Parallel-Out (PIPO)

Some of these shift registers allow for bidirectional shifting, allowing data to be shifted both to the left and the right, offering great flexibility. There are also universal shift registers that can perform all shift operations: left, right, parallel load, and serial input/output.

Accumulators

In computing, an accumulator is a special register inside the CPU that we need to know about. You see besides CPU accumulators being used within specialized devices such as digital signal processor systems and digital signal controllers, you can also make Proportional-Integral-Derivative (PID) controllers with them, and they aid in digitization within analog-to-digital converter (ADC) circuits. Needless to say, there are endless applications of these devices. The Numerically Controlled Oscillator (NCO) module we will learn about in a later chapter also uses an accumulator to operate.

The accumulator works by allowing for sequential processing. It retains intermediate results during a sequence of calculations, allowing continuous operations without writing back to another memory location, usually the main memory after each step. The accumulator itself is a dedicated register used to temporarily store data during arithmetic and logic operations. The accumulator many times does addition, being a primary register where the result of arithmetic addition operations is stored. Accumulators retain intermediate results, so it reduces the need to continuously access the main memory for every arithmetic operation, enhancing computational efficiency. Within the context of digital signal processing (DSP), this becomes quite valuable, and if you move into that domain, you will explore integration and accumulation of samples over time as well as the sum of sequence operations, which will rely heavily on an accumulator.

Multiplexers and Demultiplexers

While shift registers provide a way to store data for onward transmission, there are devices called multiplexers and demultiplexers used for routing and selection. Using multiplexers and demultiplexers, we can send signals over a single channel or from a single channel to multiple channels.

Multiplexers combine multiple input signals into a single output signal based on select control inputs. Multiplexers usually have multiple data inputs that can be controlled by select lines that determine, which input signal gets transmitted to the output. Multiplexers are used in communication systems and data transmission.

Demultiplexers perform the opposite function of multiplexers, turning a single input signal into multiple output lines based on select control inputs. They have one input and multiple outputs and also have select lines to direct the input signal to the desired output line. We can say that multiplexers route data from multiple sources to a single destination and demultiplexers distribute a single source to multiple destinations.

Buffers and Drivers

In addition to multiplexers and demultiplexers, there are also other digital circuit building blocks we can use to manage signal transmission in digital circuits. Buffers are used to amplify digital signals without changing their logic levels or altering the waveform. Some buffers can drive multiple outputs without affecting the original signal quality.

Sometimes you need to transmit data signals over longer distances or through noisy environments. When you need to do this, you get a line driver that provides higher current capability and voltage levels, improving signal strength and integrity for reliable transmission. This makes it ideal for communication between different parts of a circuit, modules, or systems placed at a distance from each other. Buffers maintain signal integrity, while line drivers focus on robust signal transmission over extended distances.

Logic-Level Conversion

An important concept to understand in the realm of digital electronics, especially when about interfacing microcontrollers, is the logic-level conversion. Before we discuss ways to convert between logic levels, we must first understand the concept of a logic level.

As you know in digital systems, data is represented in binary format, with a 0 representing off or low and a 1 representing on or high. While this knowledge may be sufficient for programming in general, when you use physical hardware, you must understand that low is 0 volts and high is the voltage that the system will recognize as a high signal when compared to the ground.

Early microcontroller systems used 5 volts as the standard because this is the voltage with which the microcontroller and any external modules operate. Recently, however, the trend has been toward using 3.3 v and even 1.8 v as the voltage to power these systems. This presents a problem because a lot of existing modules, like LCDs for example, were made to use 5 volts, whereas newer microcontrollers typically use 3.3 v. The problem also arises if you have a newer module that uses 3.3 v logic and your system runs on 5 v logic.

To solve this problem, logic-level conversion is in order. Systems typically have some tolerance for their logic level. What this means is that if you have a 5 v system, it will recognize a 3.3 v signal as a logic high. However, you cannot drive a 3.3 v logic-level system with 5 volts, as this will damage the module.

To avoid this, there are common ways to convert a 5 v logic-level system to be interfaced with a 3.3 v logic-level system, discussed next.

Run the Entire System on 3.3 v

Although it's not necessarily a logic-level conversion technique, running your system on 3.3 v will eliminate any additional components being purchased, thus reducing your bill of materials (BOM) costs. In addition, running the entire system on 3.3 v will lower overall power consumption. For these reasons, it is recommended that once it is possible, you lower the overall operating voltage of your system.

The PIC16F1719 and newer microcontrollers are capable of being run at 3.3 v or 5 v. In this book, I use 5 v as much as possible, simply because a lot of modules and sensors cater to being used by a 5 v system (although this is slowly changing, newer sensors operate on 3.3 v or even 1.8 v). If you are an Arduino user, you may have built up your electronics arsenal with 5 v components. Another advantage of 5 V is that they are much less susceptible to being disturbed by noise than 3.3 V ones because you need more noise to disturb the operation of the 5 V circuit. However, feel free to run your system at 3.3 v in your end application.

Using a Bus Transceiver

If you cannot run your entire system on 3.3 v, then you may opt to use a bus transceiver. A bus transceiver is designed to act as a bidirectional buffer, allowing data to flow bidirectionally between two buses. Transceiver is short for "transmitter-receiver," and they are used a lot in embedded systems. One such device is the 74LVC245, which provides eight bidirectional buffers; hence, we call it an "octal" bus transceiver. There is also the 74LVC244, which is like a unidirectional version of the 74LVC245.

Bidirectional Logic-Level Shifter

When you are interfacing between logic levels and a very high-speed conversion needs to be done, it is simple to use a dedicated bidirectional logic-level shifter to convert between signal levels. For prototyping purposes, I recommend the ubiquitous logic-level converter modules, as they are designed for breadboarding and work well (see Figure 3-41). For moving to a PCB, I recommend the 74LVC245, because they are simple to use and work well.

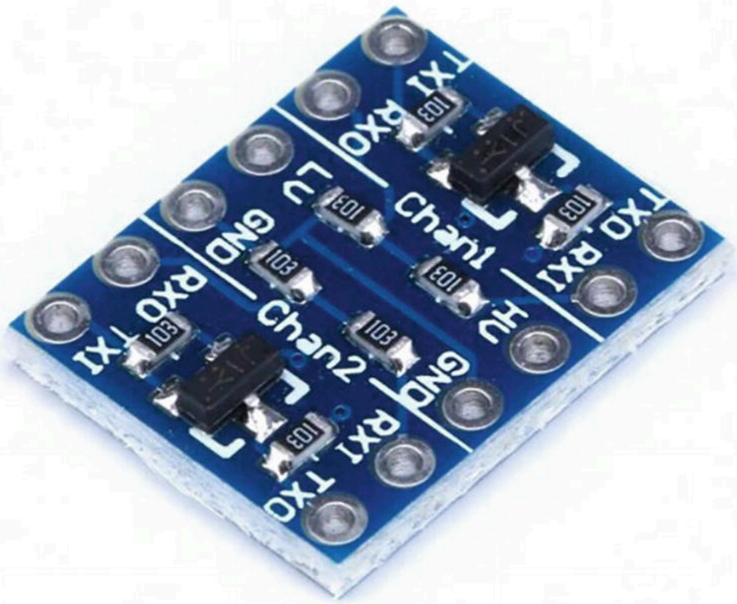


Figure 3-41 Bidirectional logic-level shifter

Use a Voltage Divider

Using a voltage divider is another way to interface between logic-level signals (see Figure 3-42). As mentioned, if your 3.3 v device is transmitting at 3.3 v, then you can directly connect this line to the 5 v device input.

However, on the transmitting end of the 5 v device, it may be necessary to use a voltage divider to convert the higher logic level to a lower one.

A good resistor combination for this type of circuit is a 1k and 2k pair. The output would be close to 3.3 v. The downside of this system is that it is best suited for very slow signals. If you are on a tight budget, then this is the method I recommend.

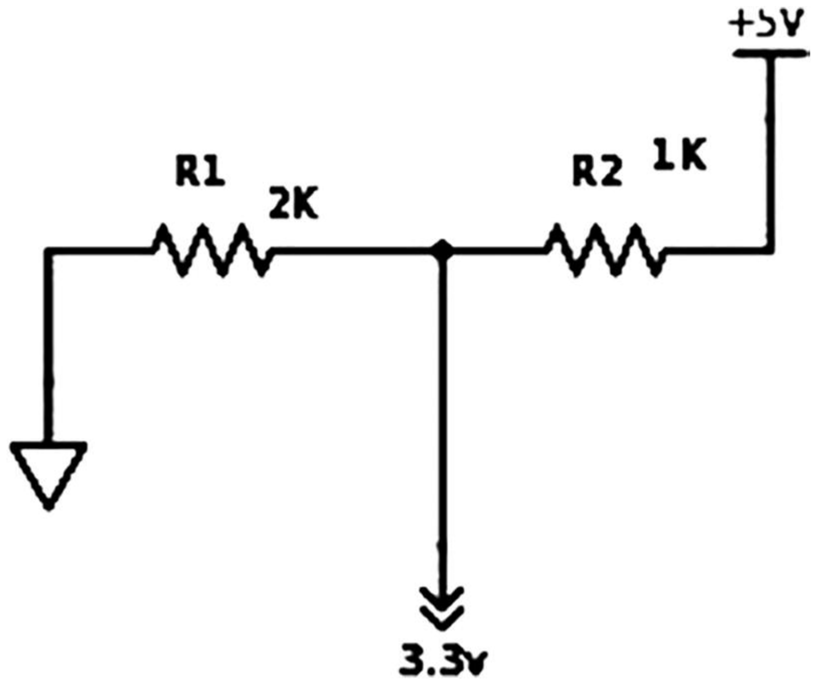


Figure 3-42 Voltage divider level shifter module

Conclusion

This chapter looked at basic electronic components commonly found in embedded systems. We looked at passive components and some of their combinations and then looked at discrete semiconductor devices as well as operational amplifiers. We covered various components as well as basic logic gates and methods of logic-level conversion. This chapter is essential to understanding how to connect devices and sensors to your microcontroller. It was a very basic introduction; however, if you understand the content here, you should be able to construct your circuits. If you need further information, there are books available that give a more detailed description of the components.

A. Subero, *Programming PIC Microcontrollers with XC8*, Maker Innovations Series
https://doi.org/10.1007/979-8-8688-0467-0_4

4. PIC Microcontrollers

Armstrong Subero[1]

(1) Moruga, Trinidad and Tobago

PIC microcontrollers had humble beginnings. Starting as a general device for embedded control applications, PIC microcontrollers have evolved into mixed signal devices capable of being designed into almost any embedded device. In this chapter, we look at microcontrollers in general, and then we look at the specifics of PIC microcontrollers.

Microcontrollers

Microcontrollers have gone from being an obscure term only known to elite electrical engineers to becoming the bread and butter of not only embedded engineers but also enthusiasts, makers, and midnight engineers everywhere. Microcontrollers are essentially microprocessors that have everything needed to make them function packed into just one package. Indeed, if you look at early microcontrollers, you will observe that they have oscillators, program memory, and data memory all in one package, essentially a microprocessor with the bare minimum needed to make them function.

Modern microprocessors have advanced to having ridiculous computing power compared to their humble origins. However, except for adding more cores and increasing core frequency and architecture, the microprocessor has remained the same. The CPU is on a package, and there is external RAM and storage that the microprocessor has access to.

Microcontrollers, on the other hand, in my opinion have undergone radical changes. Rather than just having memory, clocking, and storage on board, many specialized circuitries have been added that make these devices the perfect all-in-one tool. To make the microcontroller function, little more is needed than a reliable power supply! Some microcontrollers even have circuity on board that allows them to run from a single cell (the prevalence of 3.7 lithium-ion batteries has also made battery-powered devices easier to design). All these innovations mean that microcontrollers can be added in places they could not be added before.

The power of microcontrollers lies in their reprogrammability. You can design a system with a microcontroller, and later on, if you need to add new features, you can just add additional lines of code. Modern devices also allow you to update the flash memory remotely. This is a large improvement over the use of discrete logic that was prevalent many moons ago. In that era, once you spun a board, it was pretty much set in stone once it was shipped to a customer.

Today many microcontrollers are suited for multiple designs and applications. You can get microcontrollers from six pins like the

PIC10F220T-I/OT

to a few hundred pins. Some microcontrollers are closer to microprocessors than embedded devices. In my view, what makes a microcontroller special is the peripherals and devices available on the package. You can do a lot with modern microcontrollers. In this book, I chose the Microchip PIC microcontroller device to design our applications.

Platforms such as the Arduino have become prevalent, and when looking at comments online involving PIC microcontrollers, many people are recommending beginners start with the Arduino. I agree that Arduino is a great place for a start and I would prefer if you have some experience with Arduino before you use this book. However, the ability to design with a PIC microcontroller will give you an edge over the Arduino crowd, since as you may know, the AVR line which the Arduino is based upon was initially designed by Atmel, which was acquired by Microchip Technology. Modern PIC microcontrollers are packed with many peripherals on board that you won't find on any Arduino device.

Using a platform like Arduino also abstracts many of the hardware that will be available to you. When you learn to do bare metal programming on a device like a PIC microcontroller, you not only have the ability to choose the right device depending on your application but the skills learned can be applied to the most cutting-edge chips. Many of the principles you will learn (ports, interrupts, timers, etc.) are the same on the most advanced cutting-edge chips. You can design efficient, reliable, and professional devices. You also won't be limited to the devices that are supported by the platform and can use any device you want.

PIC Microcontrollers Overview

Microchip Technology manufactures 8-, 16-, and 32-bit microcontrollers. Microchip Technology has shipped billions of PIC microcontroller devices. These devices are in everything, from flashlights and coffee makers to automotive engine control systems and spacecraft. There are thousands of devices available. In this chapter, we take a look at the portfolio of the 8-bit devices from the portfolio that the XC8 compiler targets. I have used devices in each of the families outlined and have personally seen the evolution of these devices and have encountered them in a lot of industry applications. In the teardown of devices, I have seen PIC microcontrollers in video game controllers, electronic vaping devices, smoke detectors, portable instruments, and medical devices. To give an example, famous high-tech automotive manufacturer Tesla open-sourced a lot of documents of design files, and if you look at the design of the Tesla Roadster, you will see it has a PIC18F device in the Battery Management System (BMS).

The first thing you need to know about PIC microcontrollers is that there are several families of devices. Devices in the same family may share common characteristics including things like the device architecture, programming specifications, and of course peripherals and feature sets. When you select a device from a family, you can expect them to have things like similar clock speed and voltage and power requirements. One thing that you are sure to see being varied is the memory of the device. You will see variations in the program (also called flash memory) and data memory (RAM) within a device family. Devices within a family usually share the same set of peripherals; however, the amount of available peripherals will vary from device to device. For example, a device within a family may provide an onboard DAC module. However, the smaller members of the family may have a single DAC, whereas larger families may have two DACs available for the end user.

One key aspect you need to pay attention to when designing devices is their packaging options. These may differ depending on the device. So one device will be available in a DIP package, which is suitable for prototyping, and in your final design, you usually opt to use a device in a TQFP or QFN package. Some pearls of wisdom I can share from experience are that when learning or getting familiarized with a new family of devices, always start with the largest family within a device (the one with the most pins, RAM, and flash Memory), and then for your end application, you can always select a device with a lesser pin count or memory for your specific application.

When choosing a device family, it is important to take your time and choose one that suits your specific industry. For example, some devices offer peripherals geared toward a specific application including USB or motor control applications, whereas others are more general in their intended application. In the portable medical device industry, for example, you may want to explore devices with higher resolution ADCs, integrated op amps, and maybe capabilities for fault tolerance and low power consumption, whereas a device for power control applications may favor devices with design or architecture in favor of electromagnetic interference (EMI), plentiful high-resolution Pulse Width Modulators (PWMs), and maybe extended temperature ranges.

The trend these days across the industry is to select a device based on a specific solution. You may have a device geared toward a metering solution, professional audio, or IoT solutions. A solution considers not only the MCU itself but also the entire system; it is built around things like sensors, actuators, power management components, and the algorithms you need to run. This is contrasted with an application, in which you focus on specific tasks you want the device to solve. Let's say you work in the automotive industry; the application might be a microcontroller for engine control; however, the solution will involve not only getting the right MCU for the application but also integrating sensors, actuators, communication modules, and maybe safety-critical software algorithms to create a functional engine control system. I highlight this because selecting a microcontroller may be the most important part of designing your embedded system. It is not uncommon to spend a week or two selecting the right MCU for your intended device. Try to find solutions for a device you are trying to build; if there is no ready-made solution for you to reference, you may look at selecting a target MCU for your application and working up from there.

Aside from the core independent peripherals, one of the most important features of PIC microcontrollers is scalability, which is crucial when selecting a device. When I talk about scalability, what I am talking about is the ability to migrate from one family to the next. You may begin your design on an 8-bit device and realize that you need more features; it would be easy for you to move to a 16-bit or 32-bit device knowing you usually have access to the same peripherals and most importantly leverage the same programming tools and software environment. This is important because many times in your design you may experience feature creep from management or even yourself and moving to a new device will not be a problem. This is especially useful when this happens midway or nearing completion of the device you are building.

In this book, we focus on the XC8 compiler that targets 8-bit PIC devices; these are the devices we will focus on in this section. The 8-bit PIC families are further divided into further device families; these are the PIC10F, PIC12F, PIC16F, and 18F families. These devices are then classed into

- Baseline devices
- Midrange devices
- Enhanced midrange devices
- High-performance devices

Generally, though not the rule, if you see a PIC10F or a PIC12F part, these are baseline devices; 16F devices are midrange or enhanced midrange devices, and PIC18F devices are high-performance devices.

This is always the case though as there are baseline PIC16 devices and midrange PIC12 devices.

Baseline PIC Microcontrollers

These are at the bottom of the 8-bit Microchip food chain. The larger pin members of these families in particular are usually not very attractive for many modern designs considering the price-to-performance and features ratio of modern devices. Maybe if you need a very tiny form factor with low cost and extremely simple architecture, these devices may suffice. These devices use what is known as a 12-bit program word, meaning these devices have a 12-bit instruction word.

Baseline PIC microcontrollers consist of family members from PIC10, PIC12, and PIC16. These devices are typically used in applications that need a low pin count and extremely low power requirements and contain small programs. Some members of this family don't even include an internal oscillator, but some do. These devices can have a few bytes of memory. A once popular member of this family, the PIC16F54 has only 25 bytes of data memory and a single timer. There are members of this family that have as few as six pins. The limited memory makes the device architecture suited for Assembly language programming.

You will encounter members of this device family still in use though in very legacy applications. When you hear "20-year-old PICs" still being available for purchase, this is usually the family of devices they are talking about, especially the PIC16C54 and PIC16F54 devices. The PIC16F57 is one member of the baseline family that has found widespread use. The BASIC stamp I uses the PIC16C56, and the BASIC stamp II uses the PIC16F57 as its microcontrollers.

In new and modern devices, in practice, the PIC10F can find good use as a replacement for traditional glue logic. Rather than dropping a discreet single logic device into the application, this family can act as software programmable "smart gates" and signal conditioning. I have also used them to replace 555 timers particularly for pulse generation and as a programmable frequency source. They are more stable than the 555 timer; using the onboard timer makes them very reliable as well, and they generally have better temperature stability than a 555.

Sometimes you can straight up use them as “smart” comparators as many devices in this family have an onboard comparator.

Midrange PIC Microcontrollers

The midrange devices are what I like to call the “golden age” devices of the PIC microcontroller line. When engineers in the industry hear “PIC microcontroller,” they usually think about it. Without a doubt, the most popular and well-known members of this family are the PIC16F877 and the PIC16F84A device. For a long time, this family comprised the bulk of PIC microcontrollers. The midrange family of PIC microcontrollers has members in the PIC10, PIC12, and PIC16 families. Usually, PIC10F3xx, PIC12F6/7xx, and PIC16F6/7/8/9xx are the devices in this family. These devices are used when you need more features than baseline devices. They feature 14-bit-wide instructions compared to the 12-bit-wide instructions of the baseline family.

Devices such as the PIC16F877A, 16F84A, and 16F887 were and still are very popular with microcontroller enthusiasts, and there is a lot of code available for using these devices. The revolutionary PICkit2 starter kit and PICkit3 starter kit include devices in this family. However, unless you are supporting a legacy design, it is advisable to use the enhanced midrange microcontrollers for new designs, which have a lot of useful features.

These devices are particularly useful for learning about 8-bit PIC microcontrollers, particularly if you are a total beginner to microcontrollers. If you are learning about PIC microcontroller architecture, particularly Assembly language programming, these are the devices you want to use. There are so many low-cost tools that support these devices geared toward students and hobbyists. For example, if you want to get a taste of professional embedded development, you can use the CSS PICC “Workshop” compiler, which includes several devices in this family. For simulation, the Proteus VSM Starter Kit for PIC is a low-cost tool that allows you to learn and explore the 8-bit PIC architecture. Legacy tools such as the MPLAB 8 compiler, which still uses MPASM, also support this device; this is important because the majority of example code and books you will find on PIC microcontrollers are centered on these devices.

A lot of “student” and “hobbyist” tools also center on these devices, and as a professional engineer at some time in your career, you are bound to meet these devices in legacy applications. This is because almost all the members of this family are available through-hole PDIP packages that are easy to breadboard with. What hobbyists and amateur engineers can console themselves with is that devices in these parts have been around so long that all the kinks, quirks, and silicon errata are very minimal, making them the ideal platform to learn.

Enhanced Midrange PIC Microcontrollers

The most innovative 8-bit microcontrollers ever produced in my experience fall into this category. Looking at device architecture, you will observe that 8-bit PIC microcontrollers were originally designed with Assembly language programming in mind. Unlike the 8-bit AVR devices, which, looking at the architecture, were designed from the ground up for using the C language, the baseline and midrange have limited memory and architecture that are not suited for C.

The enhanced midrange devices changed this. These devices maintain the 14-bit instructions of the midrange family but have an architecture that is optimized for C architecture. They fall somewhere between the midrange devices and the high-performance devices we will look at in the following section. They are usually named PIC12F1xxx and PIC16F1xxx.

Even Assembly language programmers can rejoice as the device adds 14 new instructions; sometimes you need to add inline assembly for tight control loops and deterministic cycle-by-cycle execution; these additional instructions make life a little easier as these additional instructions combined with optimized architecture lead to fewer clock cycles needed for execution.

In addition to architecture and instruction changes, these devices have a faster maximum clock speed and more memory.

These devices are where cutting-edge 8-bit innovation is taking place. If you are a professional developer or have experience with other microcontrollers, the goodies on these devices make them world learning the microcontroller. The key feature of these devices for me is the core independent peripherals (CIPs) dominant in this device family. The type and amount of core independent peripherals keep changing, but some notable modules are the Configurable Logic Cell (CLC), which has combinational logic and latches integrated; the Complementary Output Generator (COG), which takes a single PWM waveform and converts it into two complementary output signals; and the Numerically Controlled Oscillator (NCO), which uses an accumulator to produce high-resolution waveforms.

We will explore these CIPs later on in more detail, but essentially these are dedicated hardware peripherals with extremely low power consumption that function without the need for CPU intervention. Almost all microcontroller devices regardless of manufacturer have a set of "core" functionality like I/O timers, ADCs, and DACs, for example, but only PICs have these innovative CIPs.

From this family, I tend to use the PIC16F1719 microcontroller, which is the basis of the projects in this book. While you don't specifically need to have this device to work through the book, I do recommend any member of the PIC16F170x/1x family of devices. Ideally you can get a Curiosity High Pin Count (HPC) development board with the PIC16F1719 device to follow along with this book. You can also use a PICkit programmer with a PIC16F1719 DIP-40 IC and a breadboard to follow along as well. These devices demonstrate many features of the enhanced midrange devices and have a nice selection of core independent peripherals and memory to showcase the potential of a "modern" PIC microcontroller. These devices are also PIC16F devices with Peripheral Pin Select (PPS) allowing the remapping of pins.

High-Performance PIC Microcontrollers

These are the highest-performing members of the PIC 8-bit family and are members of the 18F family. From the get-go based on the architecture and memory, you can tell that they were designed with the C compiler in mind. These are industry-standard devices that can be used in a variety of applications.

They feature large flash program memory, an extended instruction set, and of course integrated protocol communications such as USB, CAN, LIN, and Ethernet. They are intended for high-performance 8-bit devices and have hardware multipliers. There is the J series of devices (PIC18FxxJxx) family that offers most of the Ethernet and USB connections and the K devices (PIC18FxxKxx) family that run at up to

64 MHz. The PIC18F45K22 device is a nice device that showcases the features of the PIC18F family.

The PIC 18 “Q” devices are newer PIC18 microcontrollers you are likely to encounter. They are usually denoted PIC18-Qxx or PIC18FxxQxx, and they combine the higher performance of the PIC18 F core with CIPs; they also have powerful features like Direct Memory Access (DMA) controllers and Cyclic Redundancy Check (CRC) modules that you don’t usually see on 8-bit devices, making them powerful and flexible 8-bit devices.

In the legacy realm, if you need to use USB, you may come across the PIC18F4553 for your USB-based projects. Though there are newer versions of the PIC18 family, this chip has a lot of existing code related to its use for USB applications, as it is identical to the PIC18F4550. The exception is that the PIC18F4553 contains a larger resolution analog-to-digital converter. The PIC18F4550 is another device you can look at.

PIC16F1719 Block Diagram

Now that you have learned about the different types of PIC microcontroller families and groups, let’s look at the architecture of the PIC16F1719 (see Figure 4-1).

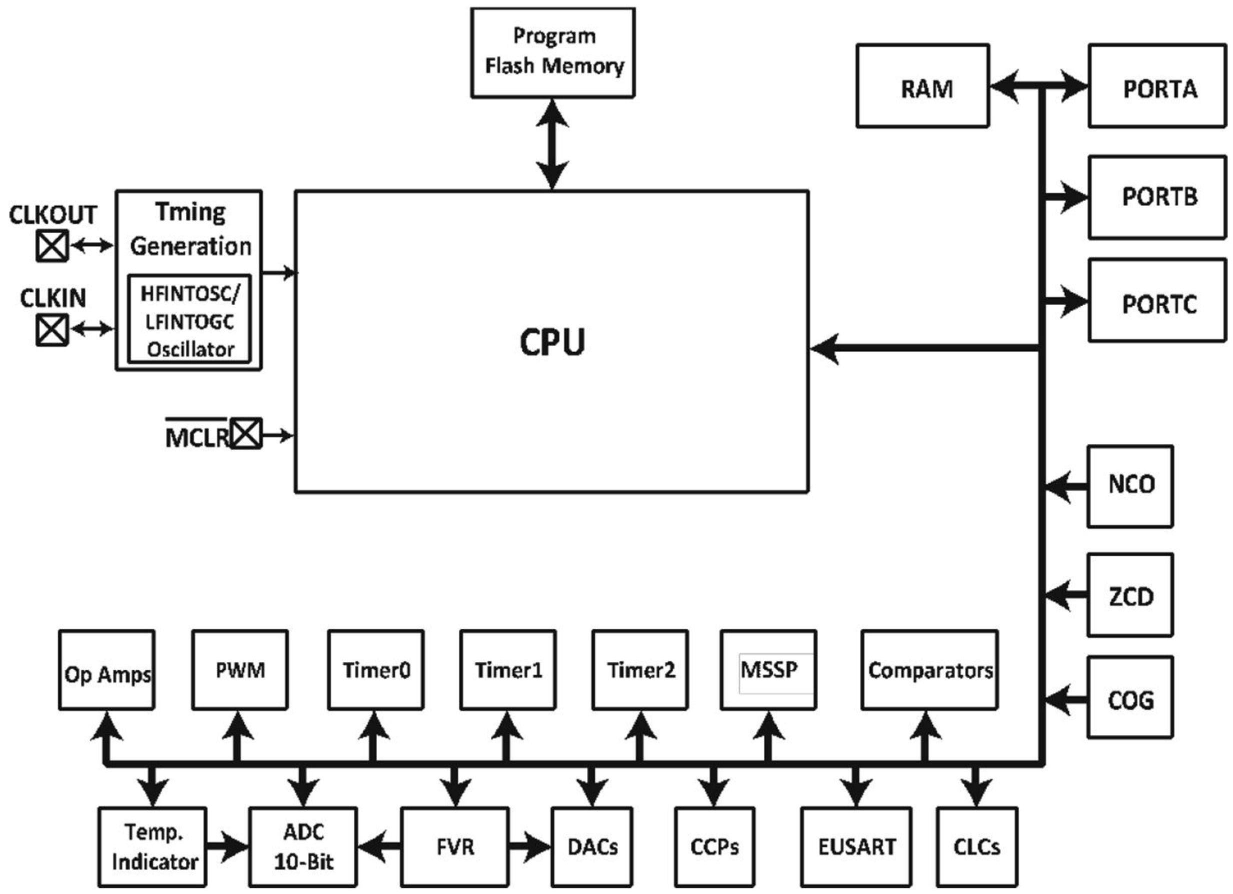


Figure 4-1 PIC16F1719 block diagram (reprinted with permission)

In looking at the block diagram shown in Figure 4-1, we see that the PIC16F1719 is very complex. It consists of a lot of peripherals, which we will discuss in the next few sections.

Program Flash Memory

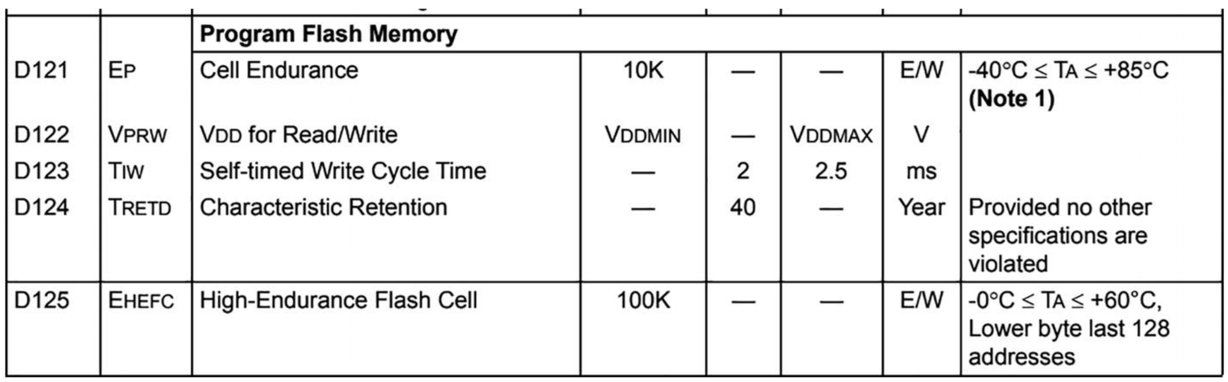
The program flash memory is a memory that stores our program and is made from flash memory technology. Older chips require UV light to erase their memory. However, with the advent of flash-based technology, microcontrollers are extremely low cost and can be reprogrammed in seconds. Flash is also a nonvolatile form of memory and features a long data retention. The PIC16F1717 flash memory has a data retention of about 40 years!

The 8-bit PIC microcontrollers' flash memory size usually consists of several kilobytes, and in the PIC16F1717, it is 14KB of program memory. This can store quite a lot of instructions, as you will see in this book. Microcontroller systems generally never use more than a couple of megabytes of program memory.

Flash Memory Operations and Limitations

Flash memory is made of cells that can store data using a system of transistors. These cells have a floating-gate transistor and a control gate. The presence or absence of electrons in the floating gate determines the state of the memory cell, which is represented in binary. When we program our microcontroller device, data is written to a flash memory cell by pushing electrons to the floating gate using a high voltage. This traps electrons in the floating gate, altering its charge and determining the cell's state. When we erase flash, we remove electrons from the floating gate by applying a high voltage, which allows the trapped electrons to tunnel out of the floating gate, resetting the cell's state. We can then read the data by allowing the control gate to apply a voltage to the cell, and when the memory controller detects current passing through the cell, it determines the state of the cell, whether there are electrons or not, which it reads as a binary value from the cell.

As amazing as flash technology is, the memory cells can be read without consequence; when we perform write and erase cycles, we degrade the memory cells, which reduces their performance and reliability. For this reason, if we look at the datasheet of the PIC microcontroller, we will see a limitation in the number of erase and write cycles.



		Program Flash Memory					
D121	EP	Cell Endurance	10K	—	—	E/W	-40°C ≤ TA ≤ +85°C **(Note 1)**
D122	VPRW	VDD for Read/Write	VDDMIN	—	VDDMAX	V	
D123	TIW	Self-timed Write Cycle Time	—	2	2.5	ms	
D124	TRETD	Characteristic Retention	—	40	—	Year	Provided no other specifications are violated
D125	EHEFC	High-Endurance Flash Cell	100K	—	—	E/W	-0°C ≤ TA ≤ +60°C, Lower byte last 128 addresses

If we look at the table from the device's electrical characteristics, we see that the program flash memory has a cell endurance of 10K erase and write (E/W) cycles. That is pretty high; however, it must be noted that midrange devices like the PIC16F877A and PIC16F887 have erase and write cycles of over 100K E/W cycles! No wonder these are great for beginners who may be reading and erasing the devices.

High Endurance Flash

One of the nice features of the midrange devices was the internal EEPROM. This EEPROM had over one million write endurance. The enhanced midrange devices do not provide this facility; instead, they have a High Endurance Flash (HEF) block that provides 128 bytes of storage with 100K erase and write cycles; this HEF is not as simple to operate as the EEPROM on the midrange devices and has several nuances, which we will look at in a later chapter.

In my opinion, the lowering of the erase/write cycles from 100K to just 10K and replacing the simpler-to-access EEPROM with HEF are the only changes made to the enhanced midrange devices that weren't appreciated. When designing instrumentation devices in particular, it is important to usually update calibration data, which can be stored on the HEF of the enhanced midrange devices and was previously stored in the EEPROM of the midrange devices.

The reason is the internal EEPROM in the midrange devices allowed for byte-by-byte erases and did not stall the MCU during reading and writing. The HEF on the enhanced midrange devices stall the MCU for a few milliseconds, and data must be erased in blocks before writing. This might not seem like much, but when developing an interrupt-driven application, while writing to the HEF, you are unable to respond to interrupts; thus, for safety-critical applications and designing with functional safety in mind, the use of the HEF is not recommended. The external EEPROM might add to your BOM cost, but it will allow for system reliability. Additionally, you need to set up linker options to use the HEF, which is a bit more cumbersome than the EEPROM on the midrange devices.

Timing Generation

If you look at the block diagram in Figure 4-1, you see a block entitled "Timing Generation." This is the oscillator module. The oscillator has a fail-safe clock monitor and can use external oscillators and internal oscillators. What the fail-safe clock monitor does is it is designed to detect a failure in the external oscillator and switch to the internal oscillator automatically.

When the device is reset, the default internal oscillator frequency selection is 500 kHz. The external oscillators can be an external clock or a crystal oscillator. For external clocks, the clock can be 0 MHz to 0.5 MHz; this is the External Clock Low Power Mode (ECL); the External Clock Medium Power Mode (ECM) gives an operation for 0.5 MHz to 4 MHz, and ECH or External Clock High Power Mode operates from 4 MHz to 32 MHz.

For the crystals, the device can be LP for 32kHz low power crystal mode, XT for Medium Gain Crystals for up to 4 MHz, and HS for High Gain Crystal for 4 MHz to 20 MHz.

This internal oscillator block contains the HFINTOSC and LFINTOSC, which are the high-frequency internal oscillator and low-frequency internal oscillator, respectively. There is also an MFINTOSC (medium-frequency internal oscillator).

The LFINTOSC operates at 31 kHz and is not calibrated. The MFINTOSC operates at 500 kHz and is factory calibrated. The HFINTOSC derives its speed from the MFINTOSC and runs at a speed of up to 16 MHz.

The maximum speed of the PIC16F1717 is 32 MHz, which can be obtained by using the Phase Locked Loop (PLL). PLLs are used to generate some multiple of the input frequency. The one onboard the PIC16F1717 is a 4x PLL, which means it will give an output frequency four times the input frequency. The PLL can be used with both external and internal clock sources.

PLLs have a period before they match the frequency and phase that are expected from them, and when this is done, the PLL is said to be locked. The PLL on the PIC16F1717 has a lock time of 2 ms.

It is important to note that the HFINTOSC and MFINTOSC, although calibrated, fall within a margin of 2% of the stipulated frequency. Thus, if you need extremely accurate timing, an external oscillator would be required. However, in this book, the internal oscillator would suffice.

A Closer Look at Microcontroller Clocks

The clock in a microcontroller system is similar to a heart in the human body. Without proper clocking, the system will inevitably fail to run, and having a poor clock mechanism in place will cause undesirable operation. Selection of the clock for your system is crucial. The clock you select for your system will determine the type of applications you can build. If you are building a time-sensitive application, then the internal microcontroller clock will not be sufficient for such an application.

A faster clock will provide more performance, but it also includes the risk of times limiting the voltage you can run the microcontroller at, and of course, it adds greater power consumption. In general, for power low voltage applications (3.3 v and below), you are generally not permitted to run the microcontroller at full clock speed, since many clocks have a threshold voltage that is needed for reliable operation.

Some peripherals are also sensitive to the frequency of the clock you use. For example, in the dated (though still very much alive) 8051 architecture that took 12 clock cycles to execute its instructions, you would see the seemingly oddball 11.05952 MHz crystal oscillator used for its design. This frequency would give the microcontroller a cycle frequency of 921.6 kHz, which is perfect for generating an accurate baud rate for a UART module. The PIC microcontroller architecture is not immune from this, and as we will see in a later chapter, the microcontroller clock can influence the accuracy of the baud rate of the UART module.

Thus, in general, once your board space and cost constraints allow it, it is best to use a quartz crystal to run the microcontroller. Keep in mind, however, that these crystals also draw power, which, as designers, is a common issue we face, which is power consumption vs. performance vs. cost. When selecting a crystal, opt for one that allows for the lowest speed that your application will allow. The crystal oscillators are very stable and low cost; however, they are sensitive to vibrations, so depending on your application, this may be something to consider when using a crystal oscillator.

Something that is usually overlooked when designing microcontroller circuits is electromagnetic interference or EMI. As a best practice, you want to design your application in such a way that will reduce the EMI generated by the circuit, and the faster you run your circuit, the more EMI you will generate, and as previously stated, the slower you run your MCU, the less power it will consume.

Flash Wait States

Since we are on the topic of microcontroller clocks, something you will encounter as you inevitably move some of your designs to larger and more powerful devices is the idea of flash wait states. When we are designing deeply embedded applications with 8-bit MCUs and relatively low clock speeds, you usually do not have to think about the speed of flash memory. The flash memory in our MCU is superb for storing program code as it is not volatile but it can be a bottleneck as the microcontroller CPU clock usually runs at a higher speed than flash and the CPU will have to wait for data to be ready as it fetches instructions or data from flash memory. The CPU thus must wait a certain number of clock cycles for the memory to respond to the instruction or fetch the required data.

The wait cycles of the CPU are known as "flash wait states." By introducing wait states, the MCU gives the flash memory enough time to fetch the required data and make it available to the microcontroller. Higher CPU clock speeds might necessitate more wait states to ensure that accurate data is retrieved from flash memory. Thus, unless we need the extra CPU processing capability, we avoid running the microcontroller at a speed that is faster than needed to make our application more efficient.

On many low-power and cost-sensitive applications, sometimes you may see an RC oscillator being used to provide clocking for the microcontroller. In general, I would say avoid using RC oscillators as they are unreliable, though if your application requires it, you may consider it as an option.

!MCLR

The PIC MCU has a not master clear pin. This pin is used to reset the microcontroller. When the MCLR line is connected to the ground "low" state, the PIC is held in a state of reset until the MCLR line is in a "high" state. The !MCLR pin is used to reset the PIC microcontroller. Even though the device has a weak internal pull-up, I have found in practice that when designing circuits, do not leave this pin floating. This pin must be connected to VDD if not in use. The circuit used for the !MCLR is shown in the following. The !MCLR pin can be controlled by the configuration bits, which can be used to enable or disable the function.

Ports

If you look at the microcontroller block diagram, you'll notice several ports marked as PORTA to PORTE. On a microcontroller, several pins are sticking out of it. Pins are used to interface the microcontroller to the outside world. However, inside the microcontroller, these pins are controlled by registers within the microcontroller, which are represented by ports.

Onboard Peripherals

The PIC microcontroller consists of several digital peripherals. These peripherals are either digital or analog. Microchip recently introduced a lot of core independent peripherals. Core independent peripherals require no intervention from the CPU to maintain operation. Let's look at these peripherals.

Analog-to-Digital Converter

The analog-to-digital converter (ADC) is used to convert analog signals to digital ones. The ADC converter onboard the PIC16F1717 has a resolution of 10 bits. What this means is that it can take a signal and break it into 1023 "steps," with the value of a step being the input voltage divided by the number of steps. For example, if we use a 4.096 v voltage reference, then we have a resolution of 4 mV per bit. For accurate ADC reading, it is advised to have a clean power supply and a stable voltage reference.

Digital-to-Analog Converter

The digital-to-analog converter (DAC) does the exact opposite of the ADC. The DAC converts an analog signal to a digital one. The DAC is typically used to generate sound and waveforms. The PIC16F1717 has two DACs. DAC1 has a resolution of 8 bits, and DAC2 has a resolution of 5 bits.

Capture/Compare/Pulse Width Modulation Module

The capture/compare/PWM (CCP module) is an important module on the PIpwmC microcontroller. Capture mode is used to measure a particular number of falling or rising edges of a timer and essentially allows the timing of an event. Compare mode allows the comparison of the value of the timer to a preset comparison value. Pulse Width Modulation (PWM) mode generates a square wave of varying frequency and duty cycle, which can be determined by the user.

Pulse Width Modulation Module

In practice, I have found that one tends to use PWM more often because of its use in applications such as lighting and motor control. Even Microchip has realized the importance of PWM and provides a dedicated PWM module in addition to the regular CCP modules. The PWM module on the PIC16F1717 has a resolution of 10 bits.

Timers

Though you may see the word "Timers" in the block diagram of the microcontroller, the timers on board the PIC16F1717 can also function as counters and perform timer/counter functions. Timers are used for time measurement, pulse generation, and counting pulses and are also very accurate for time delays. Hence, although you may see these modules simply referred to as "timers," bear in mind that they perform timer/counter functions.

The PIC16F1717 has four 8-bit timers and one 16-bit timer. Timers 0, 2, 4, and 6 are 8-bit, and timer 1 is 16-bit.

Comparators

The comparator on the PIC16F1717 compares two voltages and gives a digital output to indicate which is larger. The comparator has a minimum hysteresis of 20 mV and a maximum of 75 mV. It also has a response time of under 100.

Fixed Voltage Reference

The Fixed Voltage Reference (FVR) is used to provide a stable reference voltage to the comparator, DAC, or ADC. By doing this, the cost of paying for an external voltage reference is eliminated.

Temperature Indicator

There is a temperature indicator onboard the PIC16F1717 that has a range from -40 to 85 degrees Celsius. This temperature indicator is useful when you do not have space on the board for a temperature sensor or you want to reduce system costs.

EUSART

The Enhanced Universal Synchronous Asynchronous Receiver Transmitter (EUSART) module is used for serial communications, and a lot of external modules require this interface to communicate with the microcontroller.

CLC

The Configurable Logic Cell (CLC) is a module on the microcontroller that provides a bit of onboard sequential and combinational logic functions.

MSSP

The Master Synchronous Serial Port (MSSP) module provides two modes of operation to be configured for use—either for Serial Peripheral Interface (SPI) function or Inter-Integrated Circuit (I2C) function.

NCO

The Numerically Controlled Oscillator (NCO) is used to provide a very precise and fine resolution frequency at a duty cycle. The NCO on the PIC16F1717 uses the overflow of a 20-bit accumulator to achieve this function.

ZCD

The Zero Cross Detection module detects the point when there is no voltage present on the AC waveform. The ZCD module can be used to detect the fundamental frequency of a waveform and for sending digital data over AC circuits, such as what is done in X10 systems. The Zero Cross Detection on the PIC16F1717 has a response time of 1 uS.

COG

The Complementary Output Generator (COG) module takes one PWM signal and converts it into two complementary signals.

Operational Amplifiers

Operational Amplifiers (OPA) or op amps are the building blocks of analog systems. The PIC16F1717 includes onboard op amps. They have a gain bandwidth product of 2 MHz and a slew rate of 3 V per uS, assuming a VDD of 3.0 V.

The Enhanced Midrange CPU Core

Now that you have a basic understanding of the onboard peripherals of the microcontroller, let’s take a look at the 8-bit CPU core (see Figure 4-2).

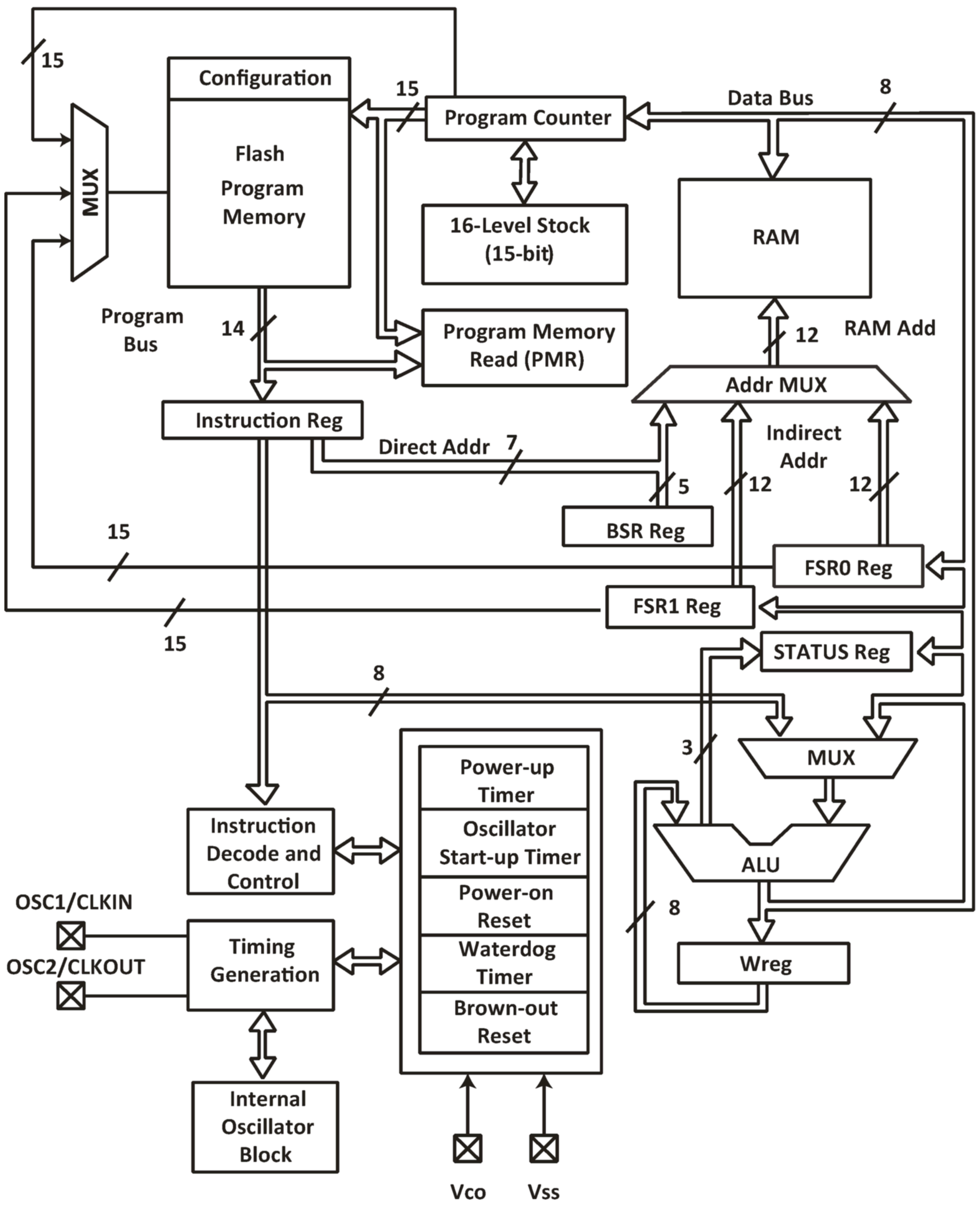


Figure 4-2 PIC16F1717 core diagram (reprinted with permission)

The PIC16 CPU is a RISC-like CPU that has a Harvard architecture; what this means is the program memory and registers are kept separate. The way the CPU operates is it takes four instructions to execute one clock cycle. The instruction clock of the CPU runs at one-fourth the speed of the oscillator. So a PIC16 MCU running with a 4 MHz oscillator will have an instruction clock of 1 MHz. Usually, this is written as Million Instructions Per Second, so we can also say it runs at 1 MIPS. Let's look at what some of these blocks in the core are responsible for. We will not discuss every detail of the architecture; however, the components that can be configured in software are discussed in the following sections.

Power-Up Timer

The power-up timer is responsible for providing a short delay to allow time for the power supply to reach the required value. After the time has passed, the program can begin to execute. The reason for this is that it is a precaution to prevent adverse effects on program execution. The power-up timer takes between 40 and 140 ms to do this.

Oscillator Start-Up Timer

The oscillator start-up timer (OST) provides a delay (in addition to the one offered by the power-up timer) to allow the clock to become stable before program execution begins. The OST counts for 1024 cycles and is independent of the frequency of the microcontroller.

Power-On Reset

While the power-up timer and oscillator start-up timer are working, the power-on reset timer holds the device in reset until the power and clock stabilize. The power must reach an acceptable level before the device is released from reset.

Watchdog Timer

The watchdog timer (WDT) automatically resets the processor after a given period as defined by the user. This is extremely important to allow an application to escape from an endless loop. To keep the program running, the WDT must be cleared or else the program will not run as intended. This is done via a special instruction we will look at in a later chapter. It is therefore important to turn off the WDT when configuring the microcontroller if it is not in use.

Brown-Out Reset

Sometimes you have a condition where you have a change in the power supply for the MCU, likely a drop in the voltage from the nominal operating voltage. This is known as a brown-out condition. A brown-out condition is one in which there is a drop in the voltage of the power supply. The supply does not fall to zero volts, just a little under its operating voltage. When this happens, the MCU can get data loss or corruption, especially if it's writing or processing data during the voltage drop. Not only can devices get their data corrupted, sometimes this also causes a "freeze," which is when the device is stuck in an unusable state.

The PIC microcontroller has brown-out circuitry to detect this condition. The brown-out reset is used to detect a brown-out condition within the microcontroller. The brown-out reset circuitry holds the microcontroller in reset until the power supply returns to an acceptable level. The brown-out reset has a response time of between 1 and 35 uS on the PIC16F1719 before it activates.

The device follows a particular startup sequence to ensure reliable operation. This startup sequence consists of assorted reset logic modules to put the device into a known state. First, a module called power-up timer (PWRT) runs to completion once it's enabled, then the oscillator start-up timer runs to completion, and then !MCLR must be released if it's enabled.

Conclusion

In this chapter, we briefly examined the PIC microcontroller, which is the main topic of this book. We looked at the different families in the 8-bit PIC microcontroller line. We also looked at the block and core diagrams of the PIC16F1719 device and its associated components. Learn the information presented in this chapter and learn it well.

A. Subero, *Programming PIC Microcontrollers with XC8*, Maker Innovations Series
https://doi.org/10.1007/979-8-8688-0467-0_5

5. Input and Output

Armstrong Subero[1]

(1) Moruga, Trinidad and Tobago

In the previous chapter, we looked at the internals of PIC microcontrollers and briefly discussed what features are available for us to develop our applications with them. When looking at the block diagram, one of the things we saw was the ports. As was described, the ports are registers that give us access to the pins of the microcontroller. The ports on the PIC16F devices can be used for input or output and can provide access to the many peripherals onboard the microcontroller. In this chapter, we will discuss input and output. Using just input and output, we will look at interfacing to LEDs, switches, and even seven-segment displays. We will also look at techniques for buffering and increasing the current output of our MCU pins, allowing us to safely drive high power loads.

I/O Fundamentals

Without a doubt, the most important peripherals on a microcontroller device are the ones that allow us to perform input and output on the device. These input and output pins are abbreviated as “I/O” and are the simplest peripherals to control on your microcontroller. Like any other computing device, microcontrollers perform input, processing, and output. The input and output parts of that cycle come from the I/O pins available on the device. These are normally called General-Purpose Input/Output (GPIO) pins.

Once you have I/O on your device, a plethora of options open up to you. You can control displays, actuators, input devices, and an endless supply of sensors and devices. Once you get the hang of it, even if you do not have access to other peripherals on the device such as communication protocols, for example, you can implement nearly any protocol with something known as “bit banging.” With bit banging, we can use the software we write to precisely toggle our I/O pins to emulate communication protocols without the need for hardware modules. While these may not be recommended for high-speed or real-time communication, sometimes there may be a silicon bug or you may need access to an additional module that is not implemented in hardware on your device.

The I/O on a microcontroller is binary; it can be set to either a high state or a low state. The values that are considered high or low are dependent on the current supply voltage that is currently being used by the device. If the device is being run from 5 v, then a logic high will be 5 v and a logic low will be 0 v. Similarly, if the device is running from a 3.3 v supply, the high would be 3.3 v and the low would be 0 v. Let's move on to looking at some registers that are needed to effectively use I/O on-picture microcontrollers.

TRIS Register

The first register related to I/O control we need to examine is the TRI-state (TRIS) register. The TRIS register gets its name from the fact that it can be in three states configured for output high, output low, or input. The TRIS register makes a port either an input or an output. To make a port an output, we write a 0 to the corresponding TRIS register of that port, and to make it an input, we write a 1. For this reason, we refer to the TRIS register as a data direction register.

We can set the entire PORTB to an output port with the following:

```
TRISB = 0;
```

We can also make the entirety of PORTB an input port:

```
TRISB = 1;
```

PIC microcontrollers have a limited number of pins, and we must effectively manage the precious I/O on these devices. The DIP version of the PIC16F1719 used for prototyping has a total of 40 pins of which 36 of those can be used as I/O pins. Thus, I/O must be properly designated. It is sometimes necessary to assign one pin on a particular port to serve as either an input or an output function. For example, to set one individual bit, for example, BIT 0, on PORTB as an output, we do the following:

```
TRISBbits.TRISB0 = 0;
```

Similarly, if we wanted to set bit one as an input, we write the following:

```
TRISBbits.TRISB1 = 1;
```

As you can see from these assignments, Microchip makes it very easy to access individual I/O on PIC microcontrollers and configure their associated registers. There is no need to mess with esoteric pointer functions or deal with mind-boggling bitwise logic. Accessing individual pins for I/O operations on PIC microcontrollers is easiest on a microcontroller. This is a fact you will appreciate when you move on to more complicated devices.

It is imperative to remember that data will not move from the port register to the pins on the microcontroller unless you give the appropriate TRIS register some value. Now we will look at some other important registers important to I/O.

Port Register

We have referred to the ports of the PIC microcontrollers on several occasions. On the PIC16F1719, there are a total of five ports ranging from A to E. These ports are 8-bit wide, and therefore, generally these ports have from bits 0 to 7. The exception to this on the PIC16F1719 is PORTE, which is a 4-bit-wide input port and a 3-bit-wide output port. The PORT register essentially reads the voltage levels of the pins on the device. This PORT register is a latch for the data to be output.

Output Latch Registers

The Output Latch Registers (LAT) are an important and overlooked type of register related to I/O. Before the enhancements made by Microchip to the PIC16F family, only the 18F family contained LAT registers. The LAT registers are an improvement on the PORT registers for writing data to output. The LAT register improves on problems associated with the PORT register for writing data to the pins.

Note that it is advisable to output data to ports using the LATx register and to read data using the PORTx register.

This is because, as previously stated, the LAT register improves on any problems that may occur while simply using the PORT register for outputting data. The reason you use the PORT register for reading data from the input pins is that the PORT register reads the actual voltage level on the pin, whereas a read of the LAT register simply reads the same without regard for the voltage level of the associated pin.

Analog Select Registers

The analog select registers (ANSEL) are used to enable or disable analog input functions on a particular pin. When using a particular I/O pin for output, it is not necessary to adjust the ANSEL register corresponding to that bit. However, if you want to use a particular I/O pin as an analog input pin, you must set the corresponding ANSEL register.

Weak Pull-Up

Now the ports on the PIC16F1717 have internal pull-up resistors. These are important as they reduce component count by eliminating the need for an external resistor. The weak pull-up can be used as seen in Listing 5-1.

```
// First we must enable weak pull-ups globally
 OPTION_REGbits.nWPUEN = 0;

// Then we configure it for the individual pin
WPUBbits.WPUB0 = 1;
```

Listing 5-1 Example of Weak Pull-Up

Once this is completed, we can connect a switch to the microcontroller without the need for an external pull-up resistor.

There are also options for other registers associated with the PORT, including those for input level control, open-drain, and slew rate control.

Blinking an LED

Now we are ready to begin blinking out LED! Making an LED blink is the "hello world" of embedded programming. While it may seem trivial, blinking an LED has a lot of use. The most important reason we blink an LED is to ensure that our system is working as intended. When we successfully blink an LED, our programmer, MCU, and software toolchain are configured properly. It also means that we have our clock set up properly. You can tell if your clock is set correctly or not based on the rate at which your microcontroller is blinking the LED.

The LED can also be used as a representation of any load such as a motor, a buzzer, or transistors, for example. To make the LED blink, it's a fairly simple process:

- Outside our loop we set clocks and our device configuration bits.
- We set the TRIS register for our microcontroller to be an output.
- We create an infinite loop.
- The PORT register is set high.
- We let the microcontroller burn a few clock cycles.
- The PORT register is set low.
- We let the microcontroller burn more clock cycles.
- This cycle repeats inside the loop indefinitely.

Let's look at each of these in stages.

Device Configuration Bits

Before we do anything with our microcontroller, I wanted to discuss the need to set the configuration bits of the device. These configuration bits, also called configuration words, are used to set options for the mode in which the device will operate and is loaded at power-up. These bits are erased by our device programmer.

Some of the important configuration bit options are as follows:

- Oscillator
- Watchdog timer
- Power-up timer
- !MCLR enable
- Code protect
- Brown-out reset

If you look at these configuration bit options, you may recognize them as being features of the device core that we looked at in the previous chapter. We set these device configuration bits with the #pragma directive. For the vast majority of the book, the device configuration bit options we will look at are given in Listing 5-2. If they are different, you will see that reflected in the associated project for the chapter.

```
// CONFIG1
```

```
#pragma config FOSC = INTOSC    // Oscillator Selection Bits
#pragma config WDTE = OFF       // Watchdog Timer Enable (WDT
disabled)
#pragma config PWRTE = OFF      // Power-up Timer Enable (PWRT
disabled)
#pragma config MCLRE = OFF      // MCLR Pin Function Select
(MCLR/VPP pin function is MCLR)
#pragma config CP = OFF         // Flash Program Memory Code
Protection

#pragma config BOREN = OFF      // Brown-out Reset Enable
#pragma config CLKOUTEN = OFF   // Clock Out Enable
#pragma config IESO = ON        // Internal/External
Switchover Mode
#pragma config FCMEN = OFF      // Fail-Safe Clock Monitor
Enable (Fail-Safe Clock Monitor is enabled)

// CONFIG2
#pragma config WRT = OFF        // Flash Memory Self-Write
Protection (Write protection off)
#pragma config PPS1WAY = ON     // Peripheral Pin Select one-
way control
#pragma config ZCDDIS = ON      // Zero-cross detect disable
#pragma config PLLEN = OFF      // Phase Lock Loop enable
#pragma config STVREN = ON      // Stack Overflow/Underflow
#pragma config BORV = LO        // Brown-out Reset Voltage
Selection
#pragma config LPBOR = OFF      // Low-Power Brown Out Reset
#pragma config LVP = OFF        // Low-Voltage Programming
Enable
```

Listing 5-2 Our Device Configuration Bit Options

As we look at the configuration bit options, we see that there are two configuration word bits: a Configuration Word 1 (CONFIG1) and a Configuration Word 2 (CONFIG2). These configuration words are situated at different memory addresses and allow for different configuration settings. Each configuration bit can be set by configuring them to "ON" and can be disabled by configuring them to "OFF".

If you look at the configuration words, you will observe that there is an option for flash memory code protection. When this bit is set to on, the microcontroller will not be able to be read by an external programmer, and it is a way to protect your code should your device ever go into production.

Our Device Header and Source Files

Putting this all together we can set up the header file for our application. The device header file will contain our configuration bit settings and include files and function prototypes for our device file. We will call this device header file "PIC16F1719_Internal.h" to let us know that we are using an internal clock. This device header file will be needed by every project going forward. As we move forward, we will not only create device header files but also header files for peripherals on the microcontroller. The "PIC16F1719_Internal.h" header file is shown in Listing 5-3.

```
/*
 * File: PIC16F1719_Internal.h
 * Author: Armstrong Subero
 * PIC: 16F1719 w/Int OSC @ 32MHz, 5v
 * Program: Header file to setup PIC16F1719
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 * Program Version:
 * 2.1
 *      * Updated for PIC16F1719
 *      * Updated Compiler and IDE versions
 * 2.0
 *      * Separated file into Header and C source file
 *      * Changed comments and layout
 *
 * Program Description: This program header will allow setup
of configuration bits and provides routines for setting up
internal oscillator and includes all devices and MCU modules
 *
 * Created on Friday 17th, November 2023, 9:56 PM
 ************************************************/

/************************************************
 *Includes and defines
 ************************************************/
// PIC16F1717 Configuration Bit Settings

// CONFIG1
#pragma config FOSC = INTOSC    // Oscillator Selection Bits
#pragma config WDTE = OFF       // Watchdog Timer Enable (WDT
disabled)
#pragma config PWRTE = OFF      // Power-up Timer Enable (PWRT
disabled)
#pragma config MCLRE = OFF      // MCLR Pin Function Select
#pragma config CP = OFF         // Flash Program Memory Code
Protection
#pragma config BOREN = OFF      // Brown-out Reset Enable
#pragma config CLKOUTEN = OFF   // Clock Out Enable
#pragma config IESO = ON        // Internal/External
Switchover Mode
```

```
#pragma config FCMEN = OFF      // Fail-Safe Clock Monitor
Enable

// CONFIG2
#pragma config WRT = OFF        // Flash Memory Self-Write
Protection
#pragma config PPS1WAY = ON     // Peripheral Pin Select one-
way control
#pragma config ZCDDIS = ON      // Zero-cross detect disable
#pragma config PLLEN = OFF      // Phase Lock Loop enable
#pragma config STVREN = ON      // Stack Overflow/Underflow
Reset Enable
#pragma config BORV = LO        // Brown-out Reset Voltage
Selection
#pragma config LPBOR = OFF      // Low-Power Brown Out Reset
#pragma config LVP = OFF        // Low-Voltage Programming
Enable

//XC8 Standard Include
#include <xc.h>
#include <stdio.h>
#include <stdlib.h>

//Other Includes
#include <stdint.h>
#include <stdbool.h>
#include <stddef.h>
#include <math.h>

//For delay routines
#define _XTAL_FREQ 32000000

//MCU Modules Includes

//Internal oscillator setup
void internal_32(); // 32 MHz
void internal_16(); // 16 MHz
void internal_8();
void internal_4();
void internal_2();
void internal_1();
void internal_31(); // 31 kHz
```
Listing 5-3 PIC16F Internal Header File

If we look at the file, it looks like your standard C header file, except we see the "include <xc.h>" line. Whenever we are working with the XC8 compiler, we need to include this file, and it is specific to the XC8 compiler. We also see a define "#define

_XTAL_FREQ 32000000". This line is needed to let the compiler know what frequency we are running the device at. When we use delays and other timing-sensitive functions, this macro is needed for these functions. We also have several function prototypes for using internal clocks. We can create a new source file called "PIC16F1719_Internal.c". This code file is given in Listing 5-4.

```
/*
 * File: PIC16F1719_Internal.c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/Int OSC @ 32MHz, 5v
 * Program: Library file to configure PIC16F1719
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 * Program Version:
 * 1.3 * Updated for PIC16F1719
 *     * Updated compiler and IDE versions
 * 1.2 * Added additional comments
 *
 * Program Description: This Library allows you to setup a
PIC16F1719
 *
 * Created on Friday 17th, November 2023, 9:52 PM
 */

/*******************************************
 *Includes and defines
 *******************************************/
#include "PIC16F1719_Internal.h"

/**********************************************************
 * Function: internal_32()
 *
 * Returns: Nothing
 *
 * Description: Sets internal oscillator to 32MHz
 *
 * Usage: internal_32()
 **********************************************************/
//Set to 32MHz
void internal_32(){
    //Clock determined by FOSC in configuration bits
    SCS0 = 0;
    SCS1 = 0;

    //Frequency select bits
    IRCF0 = 0;
    IRCF1 = 1;
    IRCF2 = 1;
    IRCF3 = 1;

    //SET PLLx4 ON
    SPLLEN = 1;
}

/**********************************************************
 * Function: internal_16()
 *
 * Returns: Nothing
 *
 * Description: Sets internal oscillator to 16MHz
 *
 * Usage: internal_16()
 **********************************************************/
//Set to 16MHz
void internal_16(){
    //Clock determined by FOSC in configuration bits
    SCS0 = 0;
    SCS1 = 0;

    //Frequency select bits
    IRCF0 = 1;
    IRCF1 = 1;
    IRCF2 = 1;
    IRCF3 = 1;

    //SET PLLx4 OFF
    SPLLEN = 0;
}

/************************************************************
 * Function: internal_8()
 *
 * Returns: Nothing
 *
 * Description: Sets internal oscillator to 8MHz
 *
 * Usage: internal_8()
 ************************************************************/
//Set to 8MHz
void internal_8(){
    //Clock determined by FOSC in configuration bits
    SCS0 = 0;
    SCS1 = 0;
    //Frequency select bits
    IRCF0 = 0;
    IRCF1 = 1;
    IRCF2 = 1;
    IRCF3 = 1;

    //SET PLLx4 OFF
    SPLLEN = 0;
}

/***********************************************************
 * Function: internal_4()
 *
 * Returns: Nothing
 *
 * Description: Sets internal oscillator to 4MHz
 *
 * Usage: internal_4()
 ***********************************************************/
//Set to 4MHz
void internal_4(){
    //Clock determined by FOSC in configuration bits
    SCS0 = 0;
    SCS1 = 0;

    //Frequency select bits
    IRCF0 = 1;
    IRCF1 = 0;
    IRCF2 = 1;
    IRCF3 = 1;

    //SET PLLx4 OFF
    SPLLEN = 0;
}

/***********************************************************
 * Function: internal_2()
 *
 * Returns: Nothing
 *
 * Description: Sets internal oscillator to 2MHz
 *
 * Usage: internal_2()
 ***********************************************************/
//Set to 2MHz
void internal_2(){
    //Clock determined by FOSC in configuration bits
    SCS0 = 0;
    SCS1 = 0;

    //Frequency select bits
    IRCF0 = 0;
    IRCF1 = 0;
    IRCF2 = 1;
    IRCF3 = 1;

    //SET PLLx4 OFF
    SPLLEN = 0;
}

/*************************************************************
 * Function: internal_1()
 *
 * Returns: Nothing
 *
 * Description: Sets internal oscillator to 1MHz
 *
 * Usage: internal_1()
 *************************************************************/
//Set to 1MHz
void internal_1(){
    //Clock determined by FOSC in configuration bits
    SCS0 = 0;
    SCS1 = 0;

    //Frequency select bits
    IRCF0 = 1;
    IRCF1 = 1;
    IRCF2 = 0;
    IRCF3 = 1;

    //SET PLLx4 OFF
    SPLLEN = 0;

/************************************************************
 * Function: internal_31()
 *
 * Returns: Nothing
 *
 * Description: Sets internal oscillator to 31kHz
 *
 * Usage: internal_31()
 ********************************************************/
//Set to 31kHz(LFINTOSC)
void internal_31(){
    //Clock determined by FOSC in configuration bits
    SCS0 = 0;
    SCS1 = 0;

    //Frequency select bits
    IRCF0 = 0;
    IRCF1 = 0;
    IRCF2 = 0;
    IRCF3 = 0;

    //SET PLLx4 OFF
    SPLLEN = 0;
}
```

Listing 5-4 PIC16F Internal Source File

Looking at the source file for the device, we see several options for configuring internal clocks, from running at the full 32 MHz down to running at 31 kHz. Depending on your target application, you may need to adjust these clocks. The majority of projects in this book run the device at either 16 or 32 MHz, and if we look at the source code in this book, we will see that I use heavily commented code; thus, line-by-line explanations are not included. I do, however, explain the most important aspects of the code as needed.

Our Main File

Once we have our include and header files for the device configuration setup, our next step is to include a source file for the device. Our source file is given in Listing 5-5.

```
/*
 * File: main.c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/int OSC @ 32MHz, 5v
 * Program: 01_Blink
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 * Program Version: 1.2
 *
 * Program Description: This Program Allows PIC16F1719 to
Turn           on an LED
 *
 * Hardware Description: An LED is connected via a 1K resistor
to PIN D1
 *
 * Created November 4th, 2016, 1:00 PM
 * Last Updated: November 17th, 2023, 10:00 PM
 */

/*******************************************************
 Change History:
 Revision     Description
 v1.2         Updated and recompiled with MPLABX v6.15 using
XC8 v2.45
 v1.1         Changed from PIC16F1717 to PIC16F1719 and
recompiled with
              MPLABX v5.15 using v2.05 of the XC8 compiler
******************************************************/

/*******************************************************
 *Includes and defines
 ******************************************************/

#include "PIC16F1719_Internal.h"

/*******************************************************
 * Function: void initMain()
 *
 * Returns: Nothing
 *
 * Description: Contains initializations for main
 *
 * Usage: initMain()
******************************************************/

void initMain(){
    // Run at 32 MHz
    internal_32();

    // Set PIN D1 as output
    TRISDbits.TRISD1 = 0;
}

/******************************************************
 * Function: Main
 *
 * Returns: Nothing
 *
 * Description: Program entry point
 *****************************************************/

void main(void) {
    initMain();
    while(1){
        // Toggle PIND1
        LATDbits.LATD1 = !LATDbits.LATD1;

        // Delay 500 milliseconds
        __delay_ms(500);
    }

    return;
}
```

Listing 5-5 LED Blinking Main File

Our C source file contains two functions: a main function and an initialization function. The initialization function will set our clocks and configure our pins. In it, we configure our MCU to run at 32 MHz, and we set PIN D1 to be an output pin.

Our main function is called our initialization function and also consists of an infinite loop that toggles our LED. The key to toggling the LED is the "LATDbits.LATD1 = !LATDbits.LATD1" line, which ensures that the current state of PIN D1 is negated. If it is high, it becomes low, and vice versa. Once that is done, we introduce a half-a-second delay before the code execution continues. Note that we could have used HEX or binary numbers to manipulate the bits on the port (a sort of byte access way of doing things), but I have found it clear and intuitive to use individual bit access to manipulate the microcontroller pins.

Later on, we will look at other architecture patterns in our embedded systems design, but this way of having an infinite main loop is called a Super Loop architecture. This is a simple-to-use way of structuring your embedded applications, and it's one that you will see regardless of the device you are using. If you ever used an Arduino, for example, you may be familiar with this type of structure where you have a "setup" and a "loop" function. This is a similar structure and is one that will be used for the majority of our simple applications going forward.

The LED Blink Program Hardware

Once we ensure that our code compiles, we can flash it to the microcontroller device. To set up the hardware for our device, we connect an LED to PIN RD1 of our device through a 1k resistor as shown in Figure 5-1. The 1k resistor is needed in series with the LED to limit the current flow to the port when the LED is on.

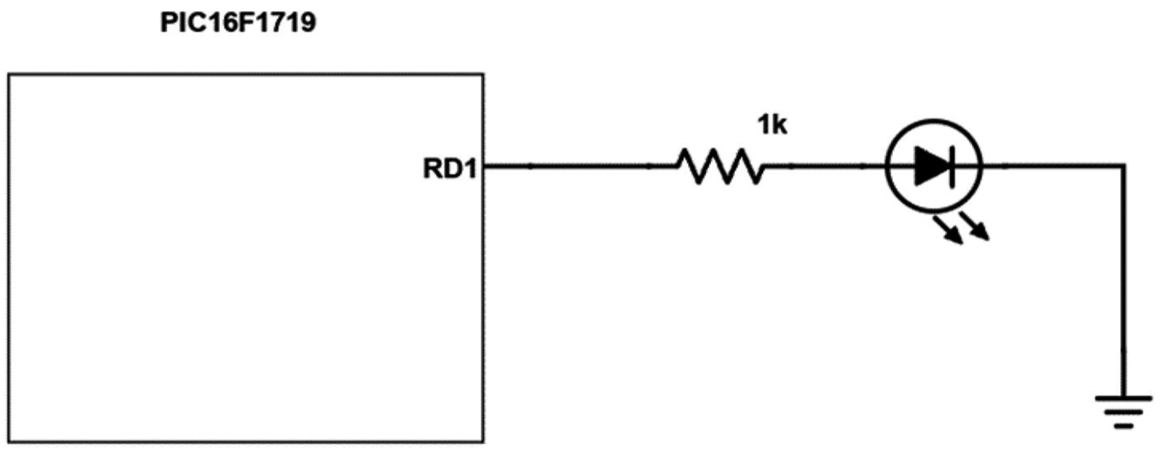


Figure 5-1 Connecting out LED to a port pin

Once the LED is connected, you should see it toggling. While this may seem like a simple application, there is a lot that you can learn from this. This application being powered proves that the device is at least on, and in many applications, at least one LED is connected to an I/O pin as a "heartbeat LED" to get our attention and let us know our application is working. By flashing the LED at different rates, you can also use it to indicate the different status of your embedded device. While this program works, it is not very efficient as it uses a software delay function that uses a lot of CPU cycles.

On every device there is a specific amount of current draw that the device can provide on each pin and we must not exceed this current draw. If we exceed this current draw, we risk damaging the microcontroller. The absolute maximum current any I/O pin on the PIC16F1719 can source is 50 mA. So as anyone who tried to connect a motor directly to the I/O pin of a microcontroller will tell you, you need to pay attention to the amount of current you are trying to draw from a pin.

A Closer Look at Bit Access

For an absolute beginner, it might be confusing to have to wonder how the bit access is done on the microcontroller device. If we look at our microcontroller, we will see that it is made up of registers. The registers on the PIC16 microcontroller are comprised of 8 bits. The register in the microcontroller looks something like this:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Those bits can be the 8 bits of a port, latch, or any other register inside the microcontroller. The bit we call Bit 7 is known as the Most Significant Bit or MSB, and Bit 0 is called the Least Significant Bit or LSB. So we can represent this register in our C program with the hexadecimal address. For example, let's say we wanted to represent TRISA; we can do so in C by defining a memory-mapped address like this:

```
// Example address for TRISA
#define PORTA   (*(volatile unsigned char*)0x85)
```

This syntax uses typecasting and dereferencing pointers to access our memory-mapped hardware register. You will see this method of accessing memory-mapped registers a lot on differing microcontroller hardware platforms.

Inside that TRISA register, we can then set bit masks for individual bits within the TRISA register. Bit masks are a technique we use to manipulate specific bits within the memory location. Let's say we have a generic register we call "REG". If we wanted to manipulate the third bit in that register—let's say we wanted to set bit 3 in that register—we can do something like this:

```
REG |= (1 << 3);
```

Within that code, the construct "(1<<3)" creates a mask, setting bit 3 to 1, and by using the bitwise OR operation, we set that bit in the register. Similarly, we can clear a bit with bit masks. The way we typically do so with memory-mapped register addresses is as follows:

```
REG &= ~(1 << 3);
```

This creates a mask with bit 3 setting it to 0 and setting the rest to 1, and the bitwise AND operation clears bit 3 in the register. Now within PIC microcontroller programming, we usually have macros for individual bit access so we have something like this:

```
#define TRISA0  0b00000001 // Bit 0
#define TRISA1  0b00000010 // Bit 1
#define TRISA2  0b00000100 // Bit 2
// ... (continue for other bits)
```

This will allow us to access the individual bits in the TRISA register without having to do so with bitwise operations, allowing for easy bit manipulation. Therefore, with PIC microcontrollers, we can access register names and bit macros easily as these are all provided by the MPLAX IDE and XC8 compiler combination.

GPIO Switching and Buffering Techniques

As we learned in the previous section, GPIO pins have a limit to the amount of current that they can source or sink. For that reason, many times you will need to be able to drive more current than can be afforded to the device. For currents from around 50 mA to 3 A, you can consider using a bipolar transistor. If you remember, the BJT is a current-controlled device, and as such, you can use it as a current-controlled switch as shown in the figure.

Low-Power Low-Side Switching

The first type of switching we can look at is low-power low-side switching. With low-power low-side switching, we control the power supply to a load by switching the side of the load that is connected to the ground. What this does is interrupt the current path to the load by controlling the connection to the lower potential of the circuit.

Since the switching power requirements are usually low in this design, we usually use a general-purpose NPN transistor to perform the switching. This circuit configuration is given in Figure 5-2.

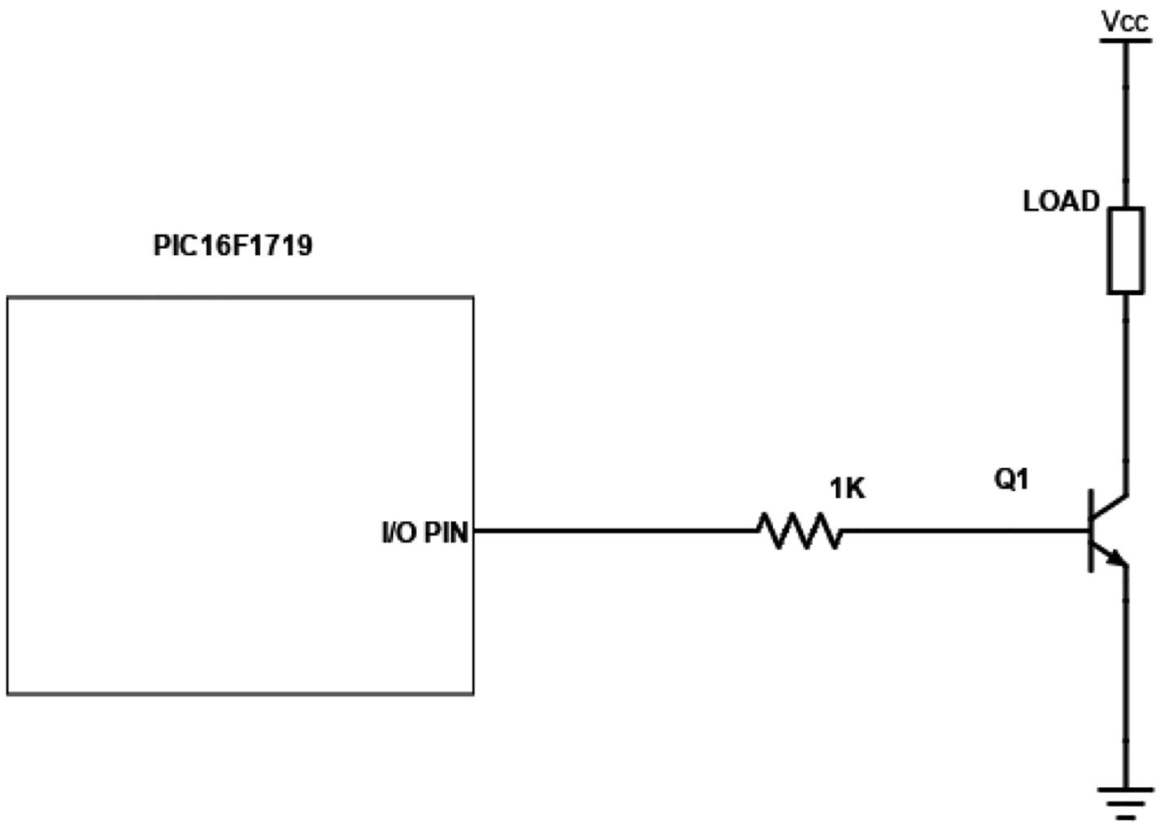


Figure 5-2 Low-power low-side switching

This is a low-cost, easy method of switching, though care must be taken if you are designing circuits that are sensitive to ground disturbances.

Low-Power High-Side Switching

There is also low-power high-side switching. When we use low-power high-side switching, what we essentially are doing is controlling power delivery to our load by switching the positive voltage supply. We typically use this type of switching when we want efficient control of power distribution, but we also want to minimize power consumption in our microcontroller.

Like low-side switching, in this setup, we have a high-side switch; typically we use a general-purpose PNP transistor and place it between the supply voltage and our load. This gives us voltage isolation where ground potential differences need to be managed, and we can switch loads that require more current than our microcontroller can provide. The setup for this circuit is given in Figure 5-3.

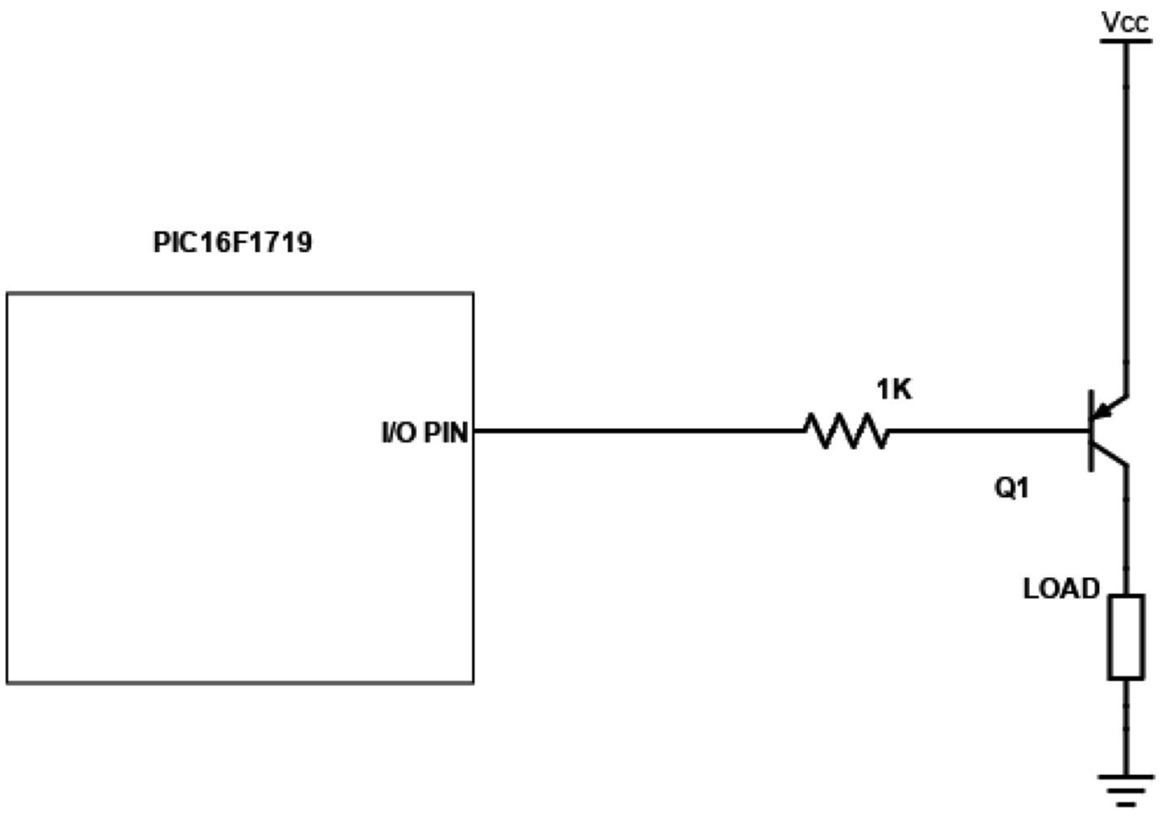


Figure 5-3 Low-power high-side switching

Using a small transistor is good for many applications. Sometimes you have to switch later loads; in such cases, you need to look at high-power switching techniques.

High-Power Low-Side Switching

Sometimes though you need more current than a BJT is capable of wielding. Once you start using currents above 3–5 A, it might be time to consider using a MOSFET device. MOSFETs are voltage-controlled devices and can switch large currents. They also allow for higher switching frequencies than BJTs, which makes them attractive as buffers. MOSFETs usually require about >12 volts to run properly unless you are using a logic-level MOSFET. In any case, you can turn on loads using a MOSFET as shown in the figure. Remember when selecting a MOSFET that you should look at the RDS ON value; in general, you want a lower RDS ON to give less heating on power losses in the device. This circuit configuration is shown in Figure 5-4.

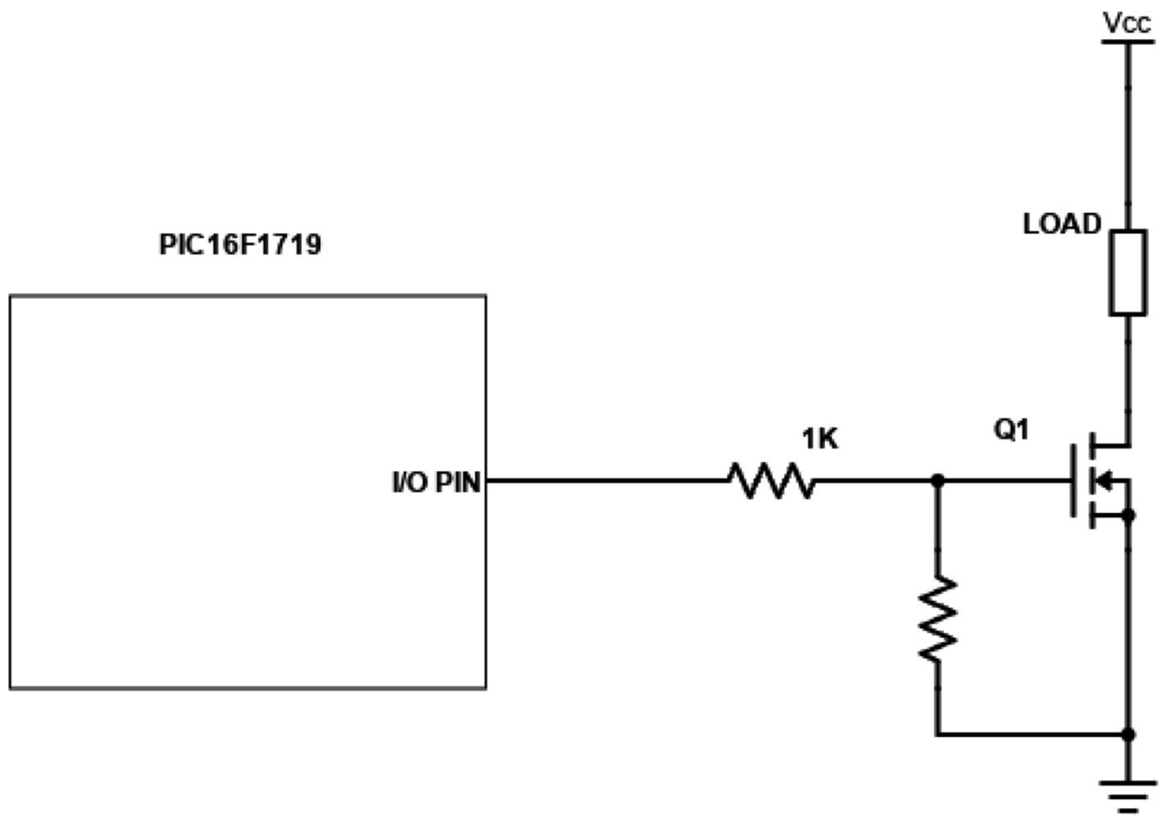


Figure 5-4 High-power low-side switching

When using MOSFET drivers, something to consider is the packaging of the device. While some MOSFET datasheets give the current rating at up to 150 amps, the standard TO-220 device package MOSFETs usually come in can handle around 75 amps, though if you are using currents that high, you are better off putting the MOSFETs in parallel and allowing them to share the load. However, if you are doing that, you also need to consider gate capacitance requirements, and you will require a MOSFET driver at that point as well.

The majority of the time you will use low-side switching for your high power requirements. You can also perform high-side high-power switching, though P-Channel MOSFETs typically have poorer characteristics and less selection than N-Channel MOSFETs; we will explore this option though in a later chapter.

Using IC Buffers and Drivers

While there are a variety of ICs you can use to increase the drive capacity of your microcontroller, to keep the devices, you need for this book, we will focus on the ULN2003A device.

The ULN2003A is an IC that contains seven identical NPN Darlington transistors with diodes to prevent inductor kickback in a single package. This device can provide up to 50 volts and 500 mA current. What makes these devices nice for interfacing with PIC microcontrollers is that they have a 2.7k base resistor on their input so that you can connect it directly to the microcontroller pin. Note that if you prefer to have a device that can work with all eight pins of an 8-bit microcontroller port, then the ULN2803A device might be one to look at. It is identical to the ULN2003A for practical purposes except it provides one additional pair of Darlington drivers. Later when we look at using a stepper driver in an upcoming chapter, we will look at using the ULN2003A device, but it is a device to keep in mind when you need to do higher voltage load driving.

Using a Pushbutton

Now that we used the microcontroller output capabilities, we can now look at how we can use the device to perform input functions. We will demonstrate this with the use of what we call pushbuttons or momentary switches. Using pushbuttons, we can control the electronic system by manually pressing a button.

The pushbutton works because when you push the plunger, the electrical circuit is completed by bringing two conductive surfaces together. The type of pushbutton we typically use for testing embedded devices is tactile pushbuttons, which are small momentary switches with a nice tactile feeling when pressed. If you ever took apart appliances, toys, tools, and other items in the consumer space, you are sure to have encountered these pushbuttons.

To use a pushbutton, we will need to configure the pin it is connected to as an input; by using the TRIS register and then by reading the state of the PORT pin it is connected to, we will be able to read the button. The program for using the pushbutton is given in Listing 5-6.

```
/*
 * File: main.c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/int OSC @ 32MHz, 5v
 * Program: 02_Pushbutton
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 * Program Version: 1.2
 *
 * Program Description: This Program Allows PIC16F1719 to Turn
on an LED
 *
 * Hardware Description: An LED is connected via a 1K resistor
to PIN D1 and a switch connected to PIN D2
 *
 * Created November 4th, 2016, 1:00 PM
 * Last Updated: November 17th, 2023, 10:05 PM
 */

/*****************************************************
 Change History:
 Revision     Description
 v1.2         Updated and recompiled with MPLABX v6.15 using
XC8 v2.45
```

```
 v1.1         Changed from PIC16F1717 to PIC16F1719 and
recompiled with
              MPLABX v5.15 using v2.05 of the XC8 compiler
*****************************************************/

/*****************************************************
 *Includes and defines
 ****************************************************/

#include "PIC16F1719_Internal.h"

/****************************************************
 * Function: void initMain()
 *
 * Returns: Nothing
 *
 * Description: Contains initializations for main
 *
 * Usage: initMain()
 ************************************************/

void initMain(){
    // Run at 32 MHz
    internal_32();

    // Set PIN D1 as output
    TRISDbits.TRISD1 = 0;

    LATDbits.LATD1 = 0;

    // Set PIN D2 as input
    TRISDbits.TRISD2 = 1;
    ANSELDbits.ANSD2 = 0;
}

/***************************************************
 * Function: Main
 *
 * Returns: Nothing
 *
 * Description: Program entry point
 ************************************************/

void main(void) {
    initMain();

    while(1){
        // Toggle LED on PUSH Button
        LATDbits.LATD1 = ~PORTDbits.RD2;
    }

    return;
}
```

Listing 5-6 Pushbutton Main File

If we look at the program, we see that we set bit 2 of PORTD as an input with the TRIS register. Because of the way the port is structured when using any pin as an input pin, we manipulate the ANSEL bit associated with the pin. ANSEL stands for Analog Select, and it allows us to configure our points to operate in analog mode. By setting or clearing the bits in the ANSEL register associated with a pin, the pin can use analog or digital mode and allow for analog functionality.

The schematic for the program is given in Figure 5-5.

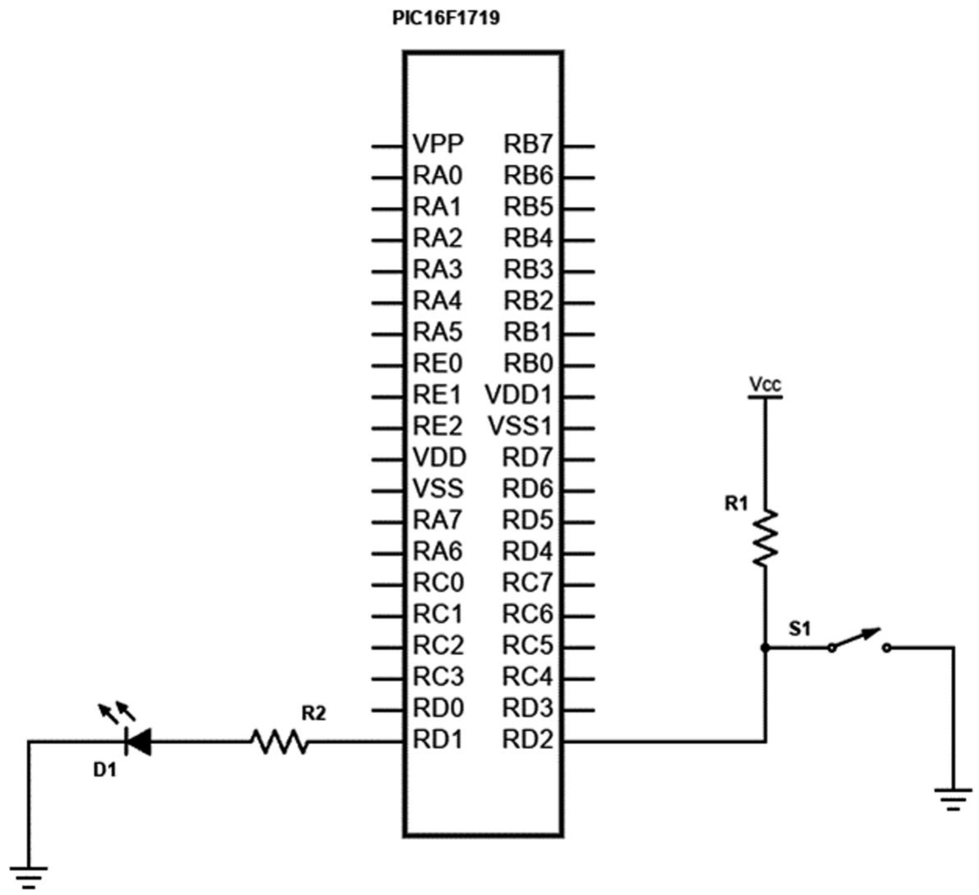


Figure 5-5 Pushbutton schematic diagram

The switch is connected in a pull-up configuration. With pull-up configuration, a resistor connects the input pin to the positive supply voltage, and when the input is left disconnected, the resistor pulls the voltage of the pin toward Vcc, giving us a default HIGH logic state. When we press the pushbutton, the resistor pulls the pin's voltage toward 0 V, overriding the pull-up resistor's action. We could also have used a pull-down configuration for the switch. In a pull-down configuration, the input pin is connected to the ground through the resistor, and the switch has a default LOW logic state.

Seven-Segment Displays

When designing embedded systems, the trend has been toward using LCDs and OLEDs. When designing low-cost, robust classical embedded systems, particularly test instruments, and rugged designs, you can't beat a seven-segment LED display. These are segmented LEDs that are intended to display the characters from A to F and digits from 0 to 9. Why these devices are particularly nice for designers is that they are usually very breadboard-friendly.

Seven-segment displays are packages that traditionally contain seven LEDs. If you count the decimal point segment available on most seven-segment displays, it is eight in reality. Each one of the LEDs in this package is referred to as a segment, hence the name seven-segment display. We can see the pinout of one in Figure 5-6.

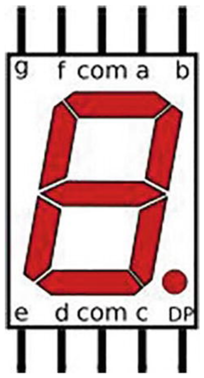


Figure 5-6 Seven-segment display pinout

As you see, each pin is associated with the letters A–G, and there is also a pin for the decimal point marked as DP. There are two pins marked COM. This is short for "common" and will connect to the ground.

To display numbers on the display, the segments associated with that pin are turned on. For example, to display the number 8, all of the segments on the LED would be on. These pins would then be connected to a particular port on the microcontroller. The numbers would then be sent to the microcontroller port that the seven-segment display is connected to.

Since the seven-segment display is, after all, a group of LEDs in one package, you need to connect resistors to each segment of the display to ensure that you do not damage them.

Note that if you are using the common anode variety, the hexadecimal numbers sent to the port will be slightly different. This is because in the common anode variety, all the LEDs are connected to power instead of ground. Let's look at a program for controlling a seven-segment display in Listing 5-7.

```
/*
 * File: Main. c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/Int OSC @ 16MHz, 5v
 * Program: 28_Seven_Segment
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 * Program Version: 1.2
 *                  * Cleaned up code
 *                  * Added additional comments
 *                  * Upgraded from PIC16F1717 to PIC16F1719
 *
 * Program Description: This Program allows the PIC16F1719 to
use a seven segment display
 *
 * Hardware Description: A seven-segment display is connected
to PORTD
 *
 *
 * Created April 1st, 2017, 12:26 PM
 * Updated December 5th, 2023, 6:45 AM */

/*******************************************************
 *Includes and defines
 *******************************************************/

#include "PIC16F1719_Internal.h"

// main loop variables
uint16_t digit, digitCount;
uint16_t i = 0;

/*******************************************************
 * Function: uint16_t seg(unsigned int num)
 *
 * Returns: Activation of Segments as Hex
 *
 * Description: Returns seven segment activations for each
digit
 *
 * Usage: x = seg(digit)
 *****************************************************/
uint16_t seg(unsigned int num)
{
   switch(num)
   {
    case 0 : return 0x3F;
    case 1 : return 0x06;
    case 2 : return 0x5B;
    case 3 : return 0x4F;
    case 4 : return 0x66;
    case 5 : return 0x6D;
    case 6 : return 0x7D;
    case 7 : return 0x07;
    case 8 : return 0x7F;
    case 9 : return 0x6F;
   }
}

/*********************************************************
 * Function: void initMain()
 *
 * Returns: Nothing
 *
 * Description: Contains initializations for main
 *
 * Usage: initMain()
 ********************************************************/

void initMain(){
    // Run at 16 MHz
    internal_16();

    // Set PORTD to output
    TRISD = 0;
}

/*********************************************************
 * Function: Main
 *
 * Returns: Nothing
 *
 * Description: Program entry point
 *************************************************************/

void main(void) {
    initMain();

    while(1)
    {
      i++;

      if (i > 9999)
      {
         i = 0;
```

```
            }

            // extract and display thousands digit from i
            digit = (i / 1000) % 10;
            digitCount = seg(digit);
            PORTD = digitCount;

            __delay_ms(2);

        }

        return;
}
```

Listing 5-7 Controlling a Seven-Segment Display

When we run this program, we see that the LED will count steadily from 0 to 9. We have a function "uint16_t seg(unsigned int num)" that takes in a digit value and returns the corresponding hex value for turning on the LED. Within the main loop, we simply take the thousands digits from the count and display it. The schematic for this project is given in Figure 5-7.

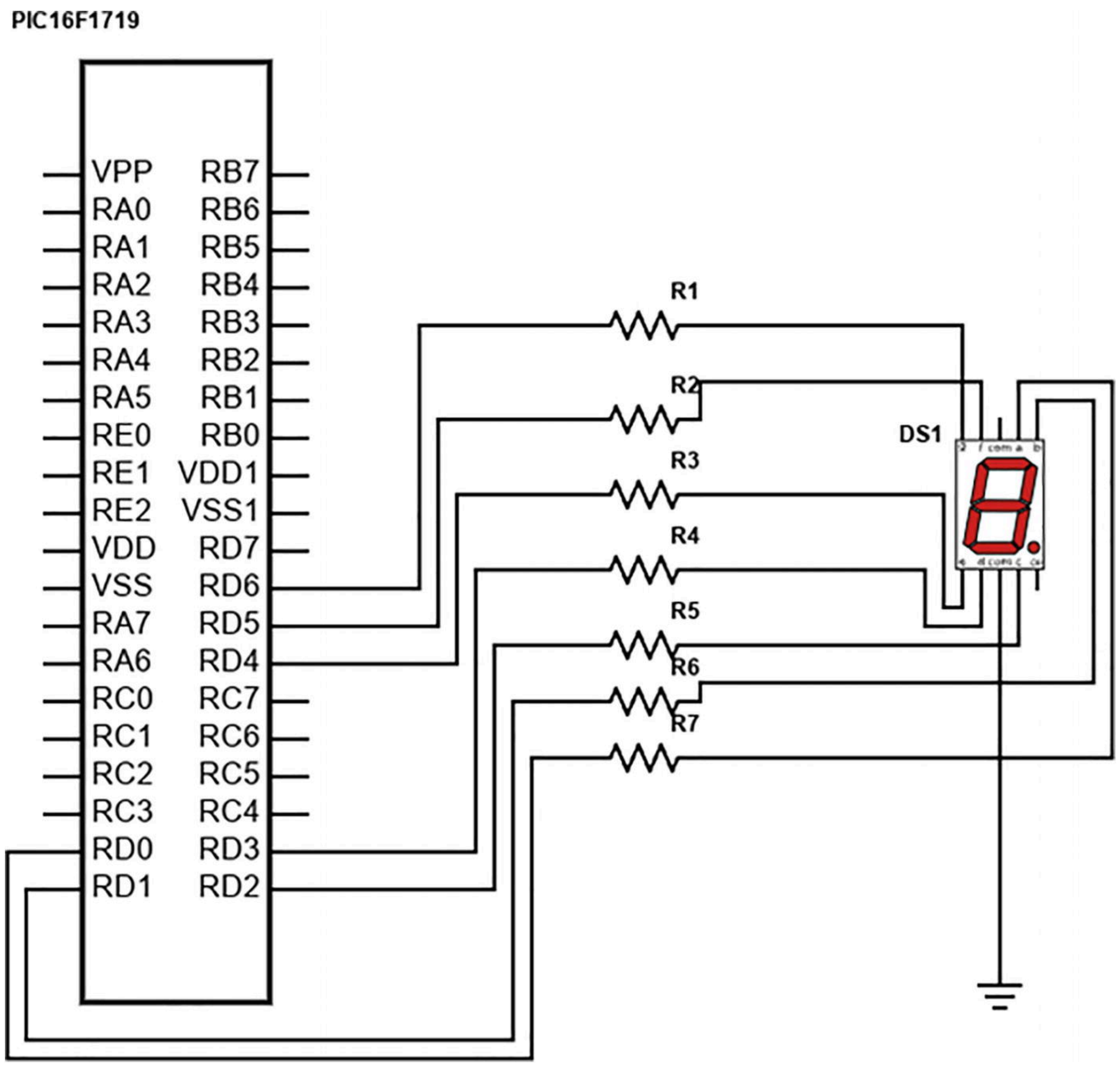


Figure 5-7 Connecting seven segments to the microcontroller

Once we have connected the LED as given in Figure 5-7, you will see the LED counting from 0 to 9.

Multiplexing Seven-Segment Displays

There are times when you'll want to use more than one seven-segment display in your application. One seven-segment display typically uses a full port of your microcontroller (see Figure 5-6). Driving two seven-segment displays would use about 16 pins of your microcontroller! This is almost half of the I/O pins you have at your disposal on the PIC16F1719 device. To avoid this, you can either use a higher pin-count microcontroller, which costs more and adds to the total cost of your system, or use multiplexing.

Display multiplexing is the process of using displays in such a way that the entire display is not on at the same time. What this means for seven-segment displays is that only one digit is on at a time. However, the microcontroller switches between updating these two displays so quickly that users cannot detect it. To multiplex these displays, we use transistors to turn the displays on and off.

The major advantage of multiplexing is that it uses less I/O on the microcontroller. Let's look at how we can multiplex seven-segment displays on the PIC16F1719 in Listing 5-8.

```
/*
 * File: Main.c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/Int OSC @ 16MHz, 5v
 * Program: 29_Seven_Segment_Mul
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 * Program Version: 1.2
 *                   * Cleaned up code
 *                   * Added additional comments
 *                   * Upgraded from PIC16F1717 to PIC16F1719
 *
 * Program Description: This Program allows the PIC16F1719 to
use multiplexed seven segment displays
 *
 * Hardware Description: Two seven-segment displays are
connected to PORTD
 *
 *
 * Created April 1st, 2017, 12:26 PM
 * Updated December 5th, 2023, 7:44 AM
 */

/*************************************************************
 *Includes and defines
 *************************************************************/

#include "PIC16F1719_Internal.h"

#define SEG0 LATBbits.LATB0
#define SEG1 LATBbits.LATB1

// main loop variables
uint16_t digit, digitCount0, digitCount1;
uint16_t i = 0;

/*************************************************************
 * Function: uint16_t seg(unsigned int num)
 *
 * Returns: Activation of Segments as Hex
 *
 * Description: Returns seven segment activations for each
digit
 *
 * Usage: x = seg(digit)
 ************************************************************/
uint16_t seg(unsigned int num)
{
   switch(num)
   {
    case 0 : return 0x3F;
    case 1 : return 0x06;
    case 2 : return 0x5B;
    case 3 : return 0x4F;
    case 4 : return 0x66;
    case 5 : return 0x6D;
    case 6 : return 0x7D;
    case 7 : return 0x07;
    case 8 : return 0x7F;
    case 9 : return 0x6F;
   }
}

/************************************************************
 * Function: void initMain()
 *
 * Returns: Nothing
 *
 * Description: Contains initializations for main
 *
 * Usage: initMain()
 ************************************************************/

void initMain(){
    // Run at 16 MHz
    internal_16();

    // Set PORTD to output
    TRISD = 0;
    LATD = 0;

    // Configure multiplexing on PORTB
    TRISBbits.TRISB0 = 0;
    TRISBbits.TRISB1 = 0;
    PORTB = 0;
}
```

```
/*************************************************************
 * Function: Main
 *
 * Returns: Nothing
 *
 * Description: Program entry point
 ************************************************************/

void main(void) {
    initMain();

    while(1)
    {
      i++;

      if (i > 9999)
      {
         i = 0;
      }

      // Take hundreds digit
      digit = (i / 100) % 10;
      digitCount0 = seg(digit);
      SEG0 = 0;
      SEG1 = 1;
      LATD = digitCount0;
      __delay_ms(5);

      // Ensure both displays are off before switching
      LATD = 0x00; // Turn off both displays
      __delay_ms(1); // Short delay between switching

      // Take Thousands digit
      SEG0 = 1;
      SEG1 = 0;
      digit = (i / 1000) % 10;
      digitCount1 = seg(digit);
      LATD = digitCount1;

      __delay_ms(5);

      // Ensure both displays are off before switching
      LATD = 0x00; // Turn off both displays
      __delay_ms(1); // Short delay between switching
    }

    return;
}
```

Listing 5-8 Multiplexing Seven-Segment Displays

In this program, we have two displays multiplexed together on PORTD, and the switching is handled by RB0 and RB1. We can look at the schematic in Figure 5-8 to see how this is done.

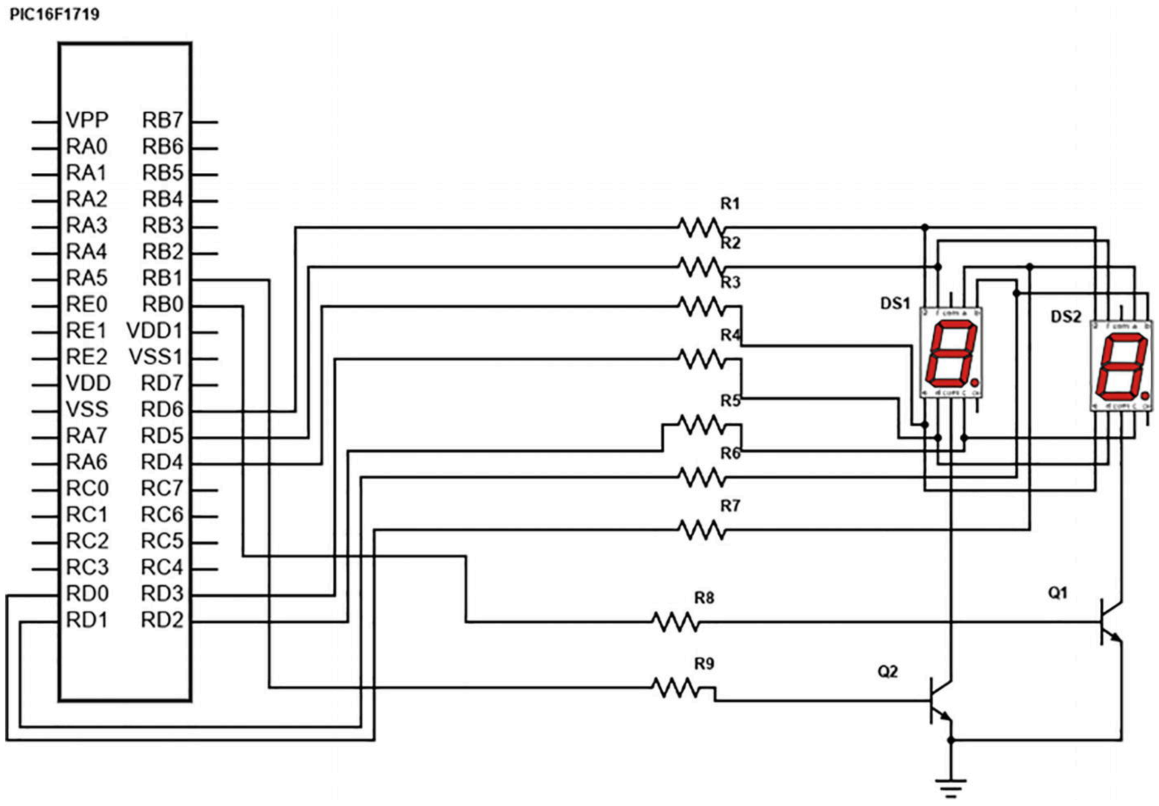


Figure 5-8 Connecting multiplexed seven-segment displays to the microcontroller

As we can see in the hardware schematic, the connection is the same as before when we connected a single seven-segment display to the microcontroller. We just added two NPN transistors to control switching the LEDs.

Project: Countdown Timer

Although we haven't covered much, we can still build a useful project. We will use the knowledge we have gained so far to make a basic countdown timer. The idea is to build a timer that can count down from a value of up to 99 seconds based on what is set by the user. We will use one button for incrementing the time, one for decrementing the time, and another button to begin the countdown. The program is given in Listing 5-9.

```
/*
 * File: Main.c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/Int OSC @ 16MHz, 5v
 * Program: 30_Countdown_Timer
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 * Program Version: 1.2
 *                  * Cleaned up code
 *                  * Added additional comments
 *                  * Upgraded from PIC16F1717 to PIC16F1719
 *
 * Program Description: This Program allows the PIC16F1719 to
use multiplexed
 * Seven-segment displays to make a countdown timer when RC1
is pressed the count increments, when RC2 is pressed the count
decrements, pressing C0 starts the decrement. When the
decrement is finished, pressing C0 again will reset the flag
allowing the process to be repeated.
 *
 * Hardware Description: Two seven-segment displays are
connected to PORTD and three pushbuttons are connected to C0,
C1 and C2
 *
 *
 * Created April 1st, 2017, 12:26 PM
 * Updated December 5th, 2023, 8:44 AM
 */

/*******************************************************
 *Includes and defines
 ******************************************************/

#include "PIC16F1719_Internal.h"

#define SEG0 LATBbits.LATB0
#define SEG1 LATBbits.LATB1

// main loop variables
uint16_t digit, digitCount0, digitCount1;
uint16_t i = 0;

/****************************************************
 * Function: uint16_t seg(unsigned int num)
 *
 * Returns: Activation of Segments as Hex
 *
 * Description: Returns seven segment activations for each
digit
 *
 * Usage: x = seg(digit)
 ****************************************************/
uint16_t seg(unsigned int num)
{
   switch(num)
   {
    case 0 : return 0x3F;
    case 1 : return 0x06;
    case 2 : return 0x5B;
    case 3 : return 0x4F;
    case 4 : return 0x66;
    case 5 : return 0x6D;
    case 6 : return 0x7D;
    case 7 : return 0x07;
    case 8 : return 0x7F;
    case 9 : return 0x6F;
   }
}

/****************************************************
 * Function: void initMain()
 *
 * Returns: Nothing
 *
 * Description: Contains initializations for main
 *
 * Usage: initMain()
 ************************************************/

void initMain(){
    // Run at 16 MHz
    internal_16();

    // Set PORTD to output
    TRISD = 0;
    LATD = 0;

    // Configure multiplexing on PORTB
    TRISBbits.TRISB0 = 0;
    TRISBbits.TRISB1 = 0;
    PORTB = 0;

    TRISCbits.TRISC0 = 1;
    TRISCbits.TRISC1 = 1;
    TRISCbits.TRISC2 = 1;

    ANSELCbits.ANSC2 = 0;
```

```
}

/*****************************************************
 * Function: Main
 *
 * Returns: Nothing
 *
 * Description: Program entry point
*****************************************************/

void main(void) {
    initMain();

    uint8_t c0_pressed = 0; // Flag to track if C0 button is
pressed

    while(1)
    {
        // Check if C0 button is pressed to toggle the flag
for decrement
        if (PORTCbits.RC0 == 0) {
            // Toggle the flag to indicate C0 is pressed
            c0_pressed = !c0_pressed;

            // Add a small debounce logic
            __delay_ms(50); // Adjust delay as required for
debounce
        }

        // If C0 is pressed, decrement the value
        if (c0_pressed == 1) {
            if (i > 0) {
                i--;
            }
        }

       // Check if C1 button is pressed to increment the value
        if (PORTCbits.RC1 == 0) {
            i++;
        }

       // Check if C2 button is pressed to decrement the value
        if (PORTCbits.RC2 == 0) {
            if (i > 0) {
                i--;
            }
        }

      if (i > 9999)
      {
         i = 0;
      }

      // Take hundreds digit
      digit = (i / 100) % 10;
      digitCount0 = seg(digit);
      SEG0 = 0;
      SEG1 = 1;
      LATD = digitCount0;
      __delay_ms(5);

      // Ensure both displays are off before switching
      LATD = 0x00; // Turn off both displays
      __delay_ms(1); // Short delay between switching

      // Take Thousands digit
      SEG0 = 1;
      SEG1 = 0;
      digit = (i / 1000) % 10;
      digitCount1 = seg(digit);
      LATD = digitCount1;

      __delay_ms(5);

      // Ensure both displays are off before switching
      LATD = 0x00; // Turn off both displays
      __delay_ms(1); // Short delay between switching

    }

    return;
}
```

Listing 5-9 Countdown Timer Project

The program works by using a flag to track RC0; when RC0 is pressed, the countdown begins; at the finish of the countdown, RC0 must be pressed again to reset the flag. RC1 and RC2 are used to increment and decrement the count. The schematic is the same as before except for added pushbuttons, which we see in Figure 5-9.

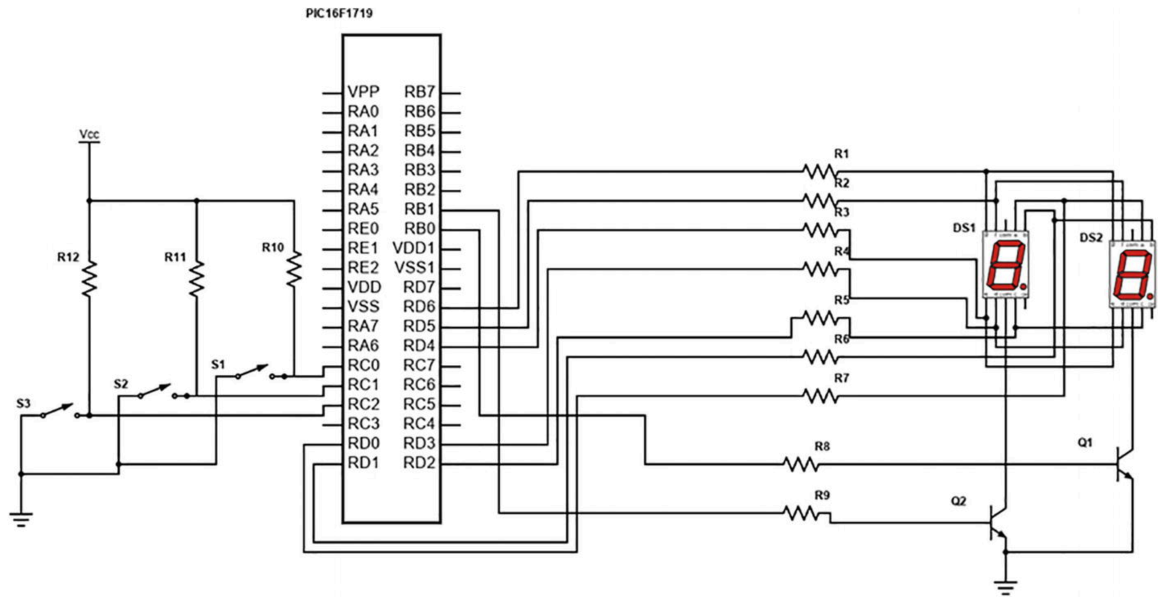


Figure 5-9 Countdown timer schematic

The resistors on the switches are 10k pull-up resistors, and 1k resistors are used on the seven-segment display.

Conclusion

This chapter looked at how to use input and output on a PIC microcontroller while looking at its applications of driving LEDs, switches, and seven-segment displays. We also covered different switching techniques of PIC microcontroller GPIO pins and learned about how to set up our device for a basic project. We covered display multiplexing, and you saw how to build a simple project, a countdown timer, using what you have learned so far.

A. Subero, *Programming PIC Microcontrollers with XC8*, Maker Innovations Series
https://doi.org/10.1007/979-8-8688-0467-0_6

6. Interrupts, Timers, Counters, and PWM

Armstrong Subero[1]

(1) Moruga, Trinidad and Tobago

Once we have gotten past the initial state of getting our programs to perform input and output, the next obvious step would be to introduce some of the most important peripherals inside the microcontroller: interrupts, timers, counters, and Pulse Width Modulation.

Introduction to Interrupts

Interrupts are one of the simplest concepts related to microcontrollers. Let me explain interrupts as simply as possible by referring to everyday life.

Imagine that you need to wake up at 6 a.m. There are two ways to know when you have to wake up. One way is to keep checking the clock until it's 6:00. However, if you do that, then you will not be able to focus on the task at hand, which is getting sleep. The other thing you can do is set your clock to an alarm to alert you that it's 6:00. In both scenarios, you will be aware of when it's 6:00; however, the second method is more effective as you can focus on the task at hand, which is getting sleep.

Taking this analogy back to microcontrollers, the first method of continually checking your clock is the polling method.

The second method of the clock alerting you when the time has reached is the interrupt method. Interrupts are assigned priorities. For example, if you are on the phone with someone and your significant other calls you, you could interrupt your call, speak with your significant other, and then resume your call. This is because your significant other is more important to you and is therefore assigned a greater priority. In the same way, the microcontroller assigns priorities to each interrupt. An interruption with a higher priority takes precedence over a lower-priority one. The microcontroller can also mask an interrupt, which is the name given to the way a microcontroller ignores a particular interrupt call for service.

The interrupt service routine, also known as the interrupt handler, is essentially the piece of code that the microcontroller executes when an interrupt is invoked. The time the microcontroller takes to respond to the interrupt and begin executing its code is known as the interrupt latency. For the PIC16F1719, the interrupt latency is between three to five instruction cycles.

The interrupt could have many sources, that is to say, things that can cause the CPU to be interrupted, and this can include timers and other onboard peripherals, even an external pin.

Timers

Once you have got the hang of interrupts, you realize that you begin to see their power when they are used in conjunction with timers. Timers can be thought of as configurable clocks internal to the microcontroller. Timers count pulses that may be either from an internal or external clock. The timers on board the microcontroller can count either regular clock pulses (thus making it a timer) or irregular clock pulses (in this mode, it is a counter). The PIC16F1719 has five timers. These are Timer0, Timer1, and Timers 2, 4, and 6. Timer0 is ubiquitous among 8-bit PIC microcontrollers. Since timers can also be used as counters, we sometimes refer to them as timers/counters.

The timer needs a clock pulse to tick. This clock source can either be internal or external to the microcontroller. When we feed internal clock pulses, the timer/counter is in timer mode. However, when we use an external clock pulse, the timer is in counter mode. Timer0 on the PIC16F1719 can be used as an 8-bit timer/counter, Timer1 is a 16-bit timer with what is known as gate control. The gate control mechanism on the timer can allow the timer to count freely or the count can be disabled depending on the state of the gate.

The input clock we get from the timer goes through a prescaler, which divides the clock input before it enters into the timer/counter device. The prescaler value will be dependent on the timer. Timer0 has a prescaler value from 1:2 to 1:256. We also have a counter associated with the timer; the counter can be preloaded with a value, and once every prescaler pulses, the timer value of the counter will increment. Some timers also have a postscaler. The postscaler increments every time the counter overflows and is reset to zero. We will explore some of the timers available on the PIC microcontroller in the following sections.

Timer0

The first timer we will look at is Timer0. All PIC microcontrollers have Timer0, which is usually abbreviated as TMR0. When we use Timer0, we can set it to increment every instruction clock, and we set the prescaler with the OPTION_REG register. To not have any prescaler on Timer0, we need to disable it by setting the Prescaler Assignment Bit (PSA) bit of the OPTION_REG. The Prescaler options we can set are as follows:

- 000: 1:2 (1/2 Division)
- 001: 1:4
- 010: 1:8

- 011: 1:16
- 100: 1:32
- 101: 1:64
- 110: 1:128
- 111: 1:256 (1/256 Division)

There is also the Timer0 Clock Source Select (T0CS) bit, which will determine whether the timer is incremented on an internal instruction or using the T0CKI pin on the microcontroller. Let's write a program to use Timer0, which we get in Listing 6-1.

```
/*
 * File: main.c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/int OSC @ 32MHz, 5v
 * Program: 03_Timer0
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 * Program Version: 1.2
 *
 * Program Description: This Program Allows PIC16F1719 to
flash an LED at 1 Hz on Timer0 overflow
 *
 * Hardware Description: An LED is connected via a 1K
resistor to PIN D1 and a switch connected to PIN D2
 *
 * Created November 4th, 2016, 1:00 PM
 * Last Updated: November 18th, 2023, 12:44 AM
 */

/*******************************************************
 Change History:
 Revision     Description
 v1.2         Updated and recompiled with MPLABX v6.15 using
XC8 v2.45
 v1.1         Changed from PIC16F1717 to PIC16F1719 and
recompiled with
              MPLABX v5.15 using v2.05 of the XC8 compiler
*******************************************************/

/*******************************************************
 *Includes and defines
 *******************************************************/

#include "PIC16F1719_Internal.h"

//counter variable
```

```
int count = 0;

/*******************************************************
 * Function: void initMain()
 *
 * Returns: Nothing
 *
 * Description: Contains initializations for main
 *
 * Usage: initMain()
 ******************************************************/

void initMain(){
    // Run at 32 MHz
    internal_32();

    // Set PIN D1 as output
    TRISDbits.TRISD1 = 0;

    LATDbits.LATD1 = 0;

    // Set PIN D2 as input
    TRISDbits.TRISD2 = 1;
    ANSELDbits.ANSD2 = 0;

    ///////////////////////
    /// Configure Timer0
    //////////////////////

    // Select timer mode
    OPTION_REGbits.TMR0CS = 0;

    // Assign Prescaler to TIMER0
    OPTION_REGbits.PSA = 0;

    // Set Prescaler to 256
    OPTION_REGbits.PS0 = 1;
    OPTION_REGbits.PS1 = 1;
    OPTION_REGbits.PS2 = 1;

    // Zero Timer
    TMR0 = 0;
}

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

```
 * Function: Main
 *
 * Returns: Nothing
 *
 * Description: Program entry point
 ****************************************************/

void main(void) {
    initMain();

    while(1){
       // When the timer overflows TMR0 interrupt flag will
be   equal to 1
        while (INTCONbits.TMR0IF != 1);

        // Reset flag after overflow
        INTCONbits.TMR0IF = 0;

        // Increment count
        count++;

        // Value = fclk / (4 * 256 * 256 * fout)
        //                                   |-- Frequency
out (in Hz)
        //                      |-- Prescaler value
        // Value =  32 000 000 / (262 144)
        // Value =  122.07 for 1 s

        // Turn on LED for 1 second on timer overflow
        if (count == 122){
            LATDbits.LATD1 = 1;
            count = 0;
        }

        // Else turn LED off
        else {
            LATDbits.LATD1 = 0;
        }
    }

    return;
}
```

Listing 6-1 Using Timer0

In this program, we set Timer0 to timer mode using TM0CS; then we assign the prescaler to Timer0 and set the prescaler value to 256. We poll the Timer0 interrupt flag, and when it's set, we reset the flag and increment a counter variable. Each time the timer overflows, we increment the count, and when the count reaches a value of 122, which equates to one second, we turn on our LED. Using the setup of the program, we can calculate the value we need for count by the following formula:

Value = Clock Speed / (4 * Prescaler Value * 256 * Desired Frequency in Hz)

Using this formula, we can adjust the timer interrupt to fit a variety of interrupt times.

Timer0 Counter Mode

We can just as easily set the Timer to counter mode by setting the TMR0CS bit, and that will enable us to use the Timer to count external pulses. We can demonstrate this process by using a switch connected to the timer counter pin and using an LED as an indication when we reach the value we want. We see the Timer0 counter mode in Listing 6-2.

```
/*
 * File: main.c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/int OSC @ 32MHz, 5v
 * Program: 04_Counter
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 * Program Version: 1.2
 *
 * Program Description: This Program Allows PIC16F1719 to
turn on an LED using Timer0 counter
 *
 * Hardware Description: An LED is connected via a 1K
resistor to PIN D1 and a switch is connected to PIN B0
 *
 * Created November 4th, 2016, 1:00 PM
 * Last Updated: November 18th, 2023, 1:01 AM
 */

/*************************************************************
 Change History:
 Revision     Description
 v1.2         Updated and recompiled with MPLABX v6.15 using
XC8 v2.45
 v1.1         Changed from PIC16F1717 to PIC16F1719 and
recompiled with
              MPLABX v5.15 using v2.05 of the XC8 compiler
************************************************************/
```

```
/**********************************************************
 *Includes and defines
 **********************************************************/

#include "PIC16F1719_Internal.h"

/*********************************************************
 * Function: void initMain()
 *
 * Returns: Nothing
 *
 * Description: Contains initializations for main
 *
 * Usage: initMain()
 *********************************************************/

void initMain(){
    // Run at 32 MHz
    internal_32();

    ///////////////////////
    /// Configure Ports
    //////////////////////

    // Set PIN D1 as output
    TRISDbits.TRISD1 = 0;

    // Turn off LED
    LATDbits.LATD1 = 0;

    // Set PIN B0 as input
    TRISBbits.TRISB0 = 1;

    // Configure ANSELB0
    ANSELBbits.ANSB0 = 0;

    ///////////////////////
    /// Configure Timer0
    //////////////////////

    // Select counter mode
    OPTION_REGbits.TMR0CS = 1;

    // Assign Prescaler to TIMER0
    OPTION_REGbits.PSA = 1;
    bool state = GIE;
    GIE = 0;
    PPSLOCK = 0xAA;
    PPSLOCKbits.PPSLOCKED = 0x00; // unlock PPS

    T0CKIPPSbits.T0CKIPPS = 0x08;   //RB0->TMR0:T0CKI;

    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    PPSLOCKbits.PPSLOCKED = 0x01; // lock PPS
    PPSLOCK = 0x55;

    GIE = state;

    // Zero Timer
    TMR0 = 0;

    // Enable timer0 interrupts and clear interrupt flag
    INTCONbits.TMR0IE = 1;
    INTCONbits.TMR0IF = 0;
}

/**********************************************************
 * Function: int ReadTimer(void)
 *
 * Returns: int readVal;
 *
 * Description: Returns the value of Timer0
 *
 * Usage: int x;
 *        x = ReadTimer();
 **********************************************************/

uint8_t ReadTimer0(void)
{
    // Read value variable
    uint8_t readVal;

    // Set variable to timer0 value
    readVal = TMR0;

    // return value
    return readVal;
}
```

```
/***********************************************************
 * Function: Main
 *
 * Returns: Nothing
 *
 * Description: Program entry point
 **********************************************************/

void main(void) {
    initMain();

    // count variable
    uint8_t count;

    while(1){

        // read timer with count
        count = ReadTimer0();

        // if counter has value of 5
        if (count == 5){
            // turn LED on
            LATDbits.LATD1 = 1;

            // short delay to see LED on
            __delay_ms(2000);

            // zero timer
            TMR0 = 0;
        }

        else
        {
            // keep LED off
            LATDbits.LATD1 = 0;
        }

    }

    return;
}
```

Listing 6-2 Using Timer0 in Counter Mode

If we look at the program, we will see that we have a "ReadTimer0" function that returns the value of the timer. We use this value to increment a count variable; when the variable reaches a value of 5, we turn our LED on and then zero our timer so the process can start again.

Peripheral Pin Select

If we look at our initMain function, you will see the following block of code:

```
PPSLOCK = 0xAA;
PPSLOCKbits.PPSLOCKED = 0x00; // unlock PPS

T0CKIPPSbits.T0CKIPPS = 0x08;   //RB0->TMR0:T0CKI;

PPSLOCK = 0x55;
PPSLOCK = 0xAA;
PPSLOCKbits.PPSLOCKED = 0x01; // lock PPS
PPSLOCK = 0x55;
```

This block of code performs what is known as Peripheral Pin Select or PPS. On older PIC microcontrollers, the I/O ports had fixed functions about which peripherals were accessed by which pins. When you were laying out a printed circuit board (PCB), this could have been a major problem, so to fix this problem, on newer PIC microcontrollers, the PPS feature was introduced to avoid this problem.

This PPS module also makes it possible to reconfigure which pins are accessed by the microcontroller at runtime and also allows an easier path to migration of legacy designs to newer ones. PPS is a very powerful feature. To use the PPS feature, we need to ensure the PPS lock is properly configured. To do this, we manipulate the PPSLOCKbits, which we clear to unlock the PPS and we set it to lock the PPS. In this case, we use the PPS module to assign RB0 to the T0CKI bit of the Timer0 register. To determine which pins and modules can be selected using PPS, it is best to consult the datasheet for the device.

Pay attention to the PPS functionality as we will see it being used throughout the book as from time to time we need to assign specific pins to peripherals using PPS.

External Interrupts

In addition to using a Timer0 interrupt and using it in timer counter mode, we can also configure the Timer0 for using external interrupts. External interrupts on the PIC16 microcontroller provide a mechanism for the microcontroller to promptly respond to external events or signals, just as we used internal interrupts to respond to timer overflows. These events, often generated by sensors or other devices, can trigger an interrupt request through dedicated pins like INT or RB0/INT. When an external event occurs, the microcontroller halts its current execution and transfers control to a specific interrupt service routine (ISR), which we explored in the last section. Configuration options include specifying the edge (rising or falling) or level triggering of the external interrupt. We use the external interrupt in Listing 6-3.

```
/*
 * File: main.c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/int OSC @ 32MHz, 5v
 * Program: 06_Timer0_Ext_Interrupt
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 * Program Version: 1.2
 *
 * Program Description: This Program Allows PIC16F1719 to
demonstrate use of an external interrupt
 *
 * Hardware Description: An LED is connected via a 1K
resistor to PIN D1 and a switch is connected to PIN B0
 *
 * Created November 4th, 2016, 1:00 PM
 * Last Updated: November 18th, 2023, 1:44 AM
 */

/*********************************************************
 Change History:
 Revision     Description
 v1.2         Updated and recompiled with MPLABX v6.15 using
XC8 v2.45
 v1.1         Changed from PIC16F1717 to PIC16F1719 and
recompiled with
              MPLABX v5.15 using v2.05 of the XC8 compiler
********************************************************/

/********************************************************
 *Includes and defines
 *******************************************************/

#include "PIC16F1719_Internal.h"

/*******************************************************
 * Function: void initMain()
 *
 * Returns: Nothing
 *
 * Description: Contains initializations for main
 *
 * Usage: initMain()
 ********************************************************/

void initMain(){
    //////////////////////
    // Configure Ports
    /////////////////////

    // Run at 16 MHz
    internal_32();

    // Set PIN D1 as output
    TRISDbits.TRISD1 = 0;

    // Turn off LED
    LATDbits.LATD1 = 0;

    // Set PIN B0 as input
    TRISBbits.TRISB0 = 1;

    // Configure ANSELB0
    ANSELBbits.ANSB0 = 0;

    /////////////////////////
    /// Configure Interrupts
    ////////////////////////

    // Set Interrupt pin to pin B0
    INTPPSbits.INTPPS = 0b01000;

    // lock   PPS
    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    PPSLOCK = 0x01;

    // Trigger on falling edge
    OPTION_REGbits.INTEDG = 0;

    // Clear external interrupt flag
    INTCONbits.INTF = 0;

    //  Enable external interrupt
    INTCONbits.INTE = 1;

    // Enable global interrupt
    ei();
}

/*******************************************************
 * Function: Main
 *
 * Returns: Nothing
 *
 * Description: Program entry point
 *******************************************************/

void main(void) {
    initMain();

    while(1){
        // Toggle LED Free running
        LATDbits.LATD1 = ~LATDbits.LATD1;
        __delay_ms(150);
    }

    return;
}

/*******************************************************
 * Function: void interrupt isr(void)
 *
 * Returns: Nothing
 *
 * Description: Use External Interrupt
 ******************************************************/
void __interrupt() isr(void)
{
    // Clear interrupt flag
    INTCONbits.INTF = 0;

    // Toggle led
    LATDbits.LATD1 = 1;
    __delay_ms(3000);
}
```

Listing 6-3 Using External Interrupts

If we look at our program, the initMain() function initializes the necessary configurations; we set our clock frequency to 32 MHz, configure PIN D1 as an output for the LED, and configure PIN B0 as an input for the external switch, which is what we use for our external interrupt. Additionally, it configures the external interrupt by setting the interrupt pin to B0, triggering on a falling edge, clearing the interrupt flag, enabling the external interrupt, and enabling global interrupts. In the main() function, the program enters an infinite loop where the LED connected to PIN D1 is toggled with a delay of 150 milliseconds.

The interrupt service routine (isr()) is executed when the external interrupt occurs. In this case, it clears the interrupt flag and toggles the LED, creating a visible response to the external event. A delay of three seconds is added to make the LED toggle noticeable so we observe the interrupt in action.

The CCP Module

The Capture/Compare/Pulse Width Modulation (CCP) module is a crucial component within PIC microcontrollers, offering versatile functionalities to manage different tasks within embedded systems. The CCP module is 16-bit wide and can be configured as a capture register, a compare register, or a PWM (Pulse Width Modulation) register, making it highly adaptable to diverse applications.

When set to capture mode, the CCP module allows the user to capture the value of Timer1 when there's a transition or change in the state of the input pin. This captured value is then stored in the CCPRx register. This capability is valuable in scenarios where precise timing or monitoring of external events is necessary.

In compare mode, the CCP module continuously compares the value in the CCPRx register with either CCPR1 or CCPR2. This comparison can be utilized to trigger specific actions or interrupt when a predetermined condition is met. This functionality is particularly useful for time-critical operations or synchronization in embedded systems.

However, PWM functionality remains one of the most extensively used features of the CCP module in everyday embedded systems designs. PWM allows for the generation of analog-like signals by rapidly switching digital output between high and low states. This rapid switching, controlled by the duty cycle, enables precise control over the amount of power delivered to connected devices, such as motors, LEDs, or speakers.

The PIC16F1719, for instance, incorporates two CCP modules: CCP1 and CCP2. These modules can be configured independently to perform any of the three functions: capture, compare, or PWM. This flexibility empowers developers to tailor the microcontroller's behavior to suit the requirements of their specific applications, enhancing the efficiency and effectiveness of the embedded systems design. Without a doubt, the mode you will be working with for the vast majority of your applications is the PWM mode, so that is what we will focus on in this chapter.

Understanding PWM

We can now delve into using the Pulse Width Modulation (PWM) feature of PIC microcontrollers. PWM signals may sound complicated at first, but we can break down the technique into the words that compose it. They are as follows:

- Pulse – A PWM signal consists of a series of pulses.
- Width – A PWM signal has characteristics that are dependent on the width of each pulse.
- Modulation – We can encode information by adjusting the width of these pulses.

PWM is a valuable technique because it allows us to digitally control the amount of power that's delivered to electrical devices. For example, if we have a 5-volt motor, at 5 volts, the motor will be running at full power; if we lower the voltage to 2.5 v, then the motor will drop to 50% power, and at 0 v, the motor will be at 0% power. Using this principle, by adjusting the average power we deliver to a motor, we can control its speed. LED brightness control will work in the same way; a pulse represents how long an LED will be on within a given time frame, so by adjusting the period the LED is on in each time frame, we can perceive different levels of brightness.

This ability to control the switching of a voltage on and off at varying intervals to give a certain average analog voltage will create an average voltage that appears analog. Thus, the ratio of the signal being on to being off is referred to as the duty cycle of the PWM signal. A 50% duty cycle means the signal will be on half of the time and off half of the time in one cycle.

By using PWM, we can maximize power efficiency, allowing us to give precise power control to a load, which leads to reduced average power consumption. Let's say we're driving a motor; when we reduce the speed with PWM, we give the motor an intermittent power supply, allowing us to consume less average power than providing a continuous full power supply.

Something that is often overlooked within electronic circuits is heat dissipation. All electronic circuits generate some type of heat. The more heat that is generated, the more expensive our BOM cost becomes. If a component heats up a lot, we may have to add something like a heat sink to actively cool it, or worse we may have to have active forced air cooling with a fan, which not only increases cost, but it means we will have to get a larger enclosure, and our power budget will also increase as the fan also consumes power. PWM can help with minimizing heat dissipation, especially in the case of power regulation. A class of regulators known as switch mode power supplies can utilize the power of PWM to step up or step down voltage efficiently. This rapid switching allows power conversion with minimal losses, reducing a lot of the problems associated with heat.

Using PWM on the PIC Microcontroller

We can now delve into using PWM on the PIC16F1719 microcontroller. The procedure will be the same regardless of the device that you are using once it's a modern enhanced midrange device. Remember when configuring PWM that everything depends on clocking, so the clock you use could affect the frequency of the PWM signal being generated.

When configuring PWM, we can use the following steps:

- Ensure you set the desired oscillator settings of your microcontroller.
- Configure the direction of the port pins you intend on using as outputs and disable the analog functionality to avoid interference with the digital signals.
- Next, we need to set the timer to be configured for PWM operation, set the timer prescaler, and enable the timer

- We need to see the period of the timer so that we will achieve our desired PWM frequency.
- Then we configure our CCP module for PWM operation
- Ensure that we map the PWM output signals to the desired pins using the PPS feature.

While this sounds like a lot of steps, in practice, it is a lot easier. We can look at an example using Timer6 to allow for PWM functionality. Take a look at Listing 6-4.

```
/*
 * File: main.c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/int OSC @ 32MHz, 5v
 * Program: 15_PWM
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 * Program Version: 1.2
 *
 * Program Description: This generates a PWM signal on pins
RB0 and RB1
 *
 * Hardware Description: An Oscilloscope probe is connected
to pins RB0 and RB1
 *
 * Created November 4th, 2016, 1:00 PM
 * Last Updated: November 18th, 2023, 4:36 AM
 */

/************************************************************
 *Includes and defines
 ***********************************************************/
#include "PIC16F1719_Internal.h"

/***********************************************************
 * Function: void initMain()
 *
 * Returns: Nothing
 *
 * Description: Contains initializations for main
 *
 * Usage: initMain()
 **********************************************************/

void initMain(){
    // Run at 32 MHz
    internal_32();
    ////////////////////
    // Configure Ports
    ///////////////////
    TRISDbits.TRISD1 = 0;

    // Set PIN B0 as output
    TRISBbits.TRISB0 = 0;

    // Set PIN B1 as output
    TRISBbits.TRISB1 = 0;

    // Turn off analog
    ANSELB = 0;

    /////////////////////
    // Configure Timer6
    /////////////////////

    // Select PWM timer as Timer6 for CCP1 and CCP2
    CCPTMRSbits.C1TSEL = 0b10;
    CCPTMRSbits.C2TSEL = 0b10;

    // Enable timer Increments every 125 ns (32 MHz clock)
1000/(32/4)
    // Period = 256 x 0.125 us = 31.25 us

    //                          Crystal Frequency
    //    PWM Freq  = -------------------------------------
---
    //                  (PRX + 1) * (TimerX Prescaler) * 4

    //    PWM Frequency = 32 000 000 / 256 * 1 * 4
    //    PWM Frequency = 31.250 kHz

    // Prescale = 1
    T6CONbits.T6CKPS = 0b00;

    // Enable Timer6
    T6CONbits.TMR6ON = 1;

    // Set timer period
    PR6 = 255;

    //////////////////////////
    // Configure PWM
    /////////////////////////
    // Configure CCP1

    // LSB's of PWM duty cycle = 00
    CCP1CONbits.DC1B = 00;

    // Select PWM mode
    CCP1CONbits.CCP1M = 0b1100;

    // Configure CCP2

    // LSB's of PWM duty cycle = 00
    CCP2CONbits.DC2B = 00;

    // Select PWM mode
    CCP2CONbits.CCP2M = 0b1100;

    //////////////////////////////
    // Configure PPS
    /////////////////////////////

    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    PPSLOCKbits.PPSLOCKED = 0x00; // unlock PPS

    // Set RB0 to PWM1
    RB0PPSbits.RB0PPS = 0b01100;

    // Set RB1 to PWM2
    RB1PPSbits.RB1PPS = 0b01101;

    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    PPSLOCKbits.PPSLOCKED = 0x01; // lock PPS
}

/**********************************************************
 * Function: Main
 *
 * Returns: Nothing
 *
 * Description: Program entry point
 **********************************************************/

void main(void) {
    initMain();
    while(1){
        // Run at 75% duty cycle @ 31.250 kHz

        // Forward
        CCPR1L = 192;
        CCPR2L = 192;
    }
     return;
}
```

Listing 6-4 Using Pulse Width Modulation

When we look at the code, we see the steps we outlined in action. First, we ensure that we select the internal oscillator to run our device. We then configure the port direction with the TRIS registers and disable analog functionality with "ANSELB = 0". To have a timer for our device, we select Timer6 (TMR6) with the "CCPTMRSbits" and set the timer prescaler and enable the timer.

We set Timer6 period to 255, which will give us a frequency of 31.25 kHz, and set CCP1 and CCP2 for PWM operation. If you look at the code, you will see that we also configure the PPS bits with "RB0PPSbits" and "RB1PPSbits" as well as set the duty cycle for the PWM signals. Setting both CCPR1L and CCPR2L to 192 sets it to a 75% duty cycle since the maximum value will be 255. In case you are wondering how we arrive at our PWM frequency, we use a simple formula:

PWM Frequency = Crystal Frequency / (PRx + 1) X (Timerx Prescaler) x 4

where PWM Frequency is the desired frequency of the PWM signal, Crystal Frequency is the microcontroller clock frequency in Hz, PRx is the value of the PRx register, which sets the period of the timer, and Timerx Prescaler is the prescaler value used for the timer.

So for our code example, we can plug our values in and we get the following:

PWM Frequency = 32 MHz / (255 + 1) x (1) 4
= 32 000 000 / 1024
= 31 250 Hz (31.250 kHz)

By adjusting these values, we can obtain a lot of different values for the PWM register. As we progress throughout the book, we will be using PWM to perform a lot of functionality.

Project: Using PWM with RGB LED Lighting

One use of PWM is in lighting applications. In this project, we look at using PWM to drive a tri-color LED. A tri-color (RGB LED) consists of three LEDs in a single package. By varying the intensity of each of these three colors, any color can be generated.

We will use the PWM to change the average voltage flowing through each individual LED and let our persistence of vision do the rest. In Listing 6-5, we see the program for driving a tri-color LED with the PWM module.

```
/*
 * File: Main.c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/Internal OSC @ 16MHz, 5v
 * Program: I10_RGB_LED
 * Compiler: XC8 (v1.41, MPLAX X v3.55)
 * Program Version: 1.0
 *
 *
 * Program Description: This demonstrates using a RGB LED
with PIC16F1719
 *
 * Hardware Description:  A RGB LED is connected as follows:
 *                        Red   - RB0
 *                        Green - RB3
 *                        Blue  - RB2
 *
 *
 * Created Tuesday 18th, April, 2017, 11:53 AM
 */

/*******************************************************
 *Includes and defines
 ******************************************************/

#include "PIC16F1719_Internal.h"

/*
 Value for PWM1
 */
void PWM1_LoadDutyValue(uint16_t dutyValue) {
    // Writing to 8 MSBs of pwm duty cycle in CCPRL register
    CCPR1L = ((dutyValue & 0x03FC) >> 2);

    // Writing to 2 LSBs of pwm duty cycle in CCPCON register
    CCP1CON = (CCP1CON & 0xCF) | ((dutyValue & 0x0003) << 4);
}

/*
  Value for PWM2
 */
void PWM2_LoadDutyValue(uint16_t dutyValue) {
    // Writing to 8 MSBs of pwm duty cycle in CCPRL register
    CCPR2L = ((dutyValue & 0x03FC) >> 2);
    // Writing to 2 LSBs of pwm duty cycle in CCPCON register
    CCP2CON = (CCP2CON & 0xCF) | ((dutyValue & 0x0003) << 4);
}

/*
 Value for PWM3
 */
void PWM3_LoadDutyValue(uint16_t dutyValue) {
    // Writing to 8 MSBs of PWM duty cycle in PWMDCH register
    PWM3DCH = (dutyValue & 0x03FC) >> 2;

    // Writing to 2 LSBs of PWM duty cycle in PWMDCL register
    PWM3DCL = (dutyValue & 0x0003) << 6;
}

/*
 Value for RGB LED
 */
void RGB_LoadValue(uint16_t red, uint16_t green, uint16_t
blue)
{

    PWM1_LoadDutyValue(red);
    PWM2_LoadDutyValue(green);
    PWM3_LoadDutyValue(blue);
}

/*******************************************************
 * Function: void initMain()
 *
 * Returns: Nothing
 *
 * Description: Contains initializations for main
 *
 * Usage: initMain()
 ******************************************************/

void initMain(){
    // Run at 16 MHz
    internal_16();

    ////////////////////
    // Configure Ports
    ///////////////////
    // Set PIN B0 as output
    TRISBbits.TRISB0 = 0;

    // Set PIN B1 as output
    TRISBbits.TRISB1 = 0;

    // Set PIN B2 as output
    TRISBbits.TRISB2 = 0;

    // Turn off analog
    ANSELB = 0;

    /////////////////////
    // Configure Timer6
    /////////////////////

    // Select PWM timer as Timer6 for CCP1 and CCP2
    CCPTMRSbits.C1TSEL = 0b10;
    CCPTMRSbits.C2TSEL = 0b10;

    // Enable timer Increments every 250 ns (16MHz clock)
1000/(16/4)
    // Period = 256 x 0.25 us = 64 us

    //                          Crystal Frequency
    //    PWM Freq  = ---------------------------------------
--
    //                  (PRX + 1) * (TimerX Prescaler) * 4

    //    PWM Frequency = 16 000 000 / 256 * 1 * 4
    //    PWM Frequency = 15.625 kHz

    // Prescale = 1
    T6CONbits.T6CKPS = 0b00;

    // Enable Timer6
    T6CONbits.TMR6ON = 1;

    // Set timer period
    PR6 = 255;

    //////////////////////////
    // Configure PWM
    //////////////////////////

    // Configure CCP1
    // LSB's of PWM duty cycle = 00
    CCP1CONbits.DC1B = 00;

    // Select PWM mode
    CCP1CONbits.CCP1M = 0b1100;

    // Configure CCP2

    // LSB's of PWM duty cycle = 00
    CCP2CONbits.DC2B = 00;

    // Select PWM mode
    CCP2CONbits.CCP2M = 0b1100;

    // Configure PWM 3

    // PWM3EN enabled, PWM3POL active high
    PWM3CON = 0x80;

    // PWM3DCH 127
    PWM3DCH = 0x7F;

    // PWM3DCL 192
    PWM3DCL = 0xC0;

    // Select timer6
    CCPTMRSbits.P3TSEL = 0b10;

    //////////////////////////////
    // Configure PPS
    /////////////////////////////

    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    PPSLOCKbits.PPSLOCKED = 0x00; // unlock PPS

    // Set RB0 to PWM1
    RB0PPSbits.RB0PPS = 0b01100;

    // Set RB1 to PWM2
    RB1PPSbits.RB1PPS = 0b01101;

    // Set RB2 to PWM3
    RB2PPSbits.RB2PPS = 0x0E;

    PPSLOCK = 0x55;
```

```
    PPSLOCK = 0xAA;
    PPSLOCKbits.PPSLOCKED = 0x01; // lock PPS

}

/*************************************************************
 * Function: Main
 *
 * Returns: Nothing
 *
 * Description: Program entry point
 ************************************************************/

void main(void) {
    initMain();

    // All channels initially 0
    PWM1_LoadDutyValue(0);
    PWM2_LoadDutyValue(0);
    PWM3_LoadDutyValue(0);

    while(1){

        // Red
        RGB_LoadValue(512,   0,  0);
        __delay_ms(1000);

        // Green
        RGB_LoadValue(0,   512,  0);
        __delay_ms(1000);

        // Blue
        RGB_LoadValue(0,   0,   512);
        __delay_ms(1000);

        // Yellow
        RGB_LoadValue(192,   192,   0);
        __delay_ms(1000);

        // Purple
        RGB_LoadValue(192,   0,   192);
        __delay_ms(1000);

        // Aquamarine
        RGB_LoadValue(0,   512,   512);
        __delay_ms(1000);

    }
     return;
}
```

Listing 6-5 Using Pulse Width Modulation with RGB LED

In this program, the RGB LED is connected to the microcontroller's RB0 (Red), RB3 (Green), and RB2 (Blue) pins. We configure the RGB LED pins as outputs, turn off analog functions for those pins, and configure Timer6 to be used for PWM generation. Additionally, we set up PWM parameters for three channels (PWM1, PWM2, and PWM3) and configure the Peripheral Pin Select (PPS) to map PWM outputs to the corresponding pins (RB0, RB1, and RB2).

In the "main()" function, we start by initializing the main configurations and then enter an infinite loop where we cycle through various colors. We use the "RGB_LoadValue" function to set the intensity of each color channel (Red, Green, Blue) using PWM. After setting a color, there is a delay of 1000 milliseconds (1 second) before transitioning to the next color.

The program demonstrates the flexibility of the RGB LED by showcasing different colors, such as red, green, blue, yellow, purple, and aquamarine, and the power of PWM.

Conclusion

This concludes this chapter, where we looked at a few more onboard modules of the PIC microcontroller. We looked at interrupts, which allow the microcontroller to instantly jump to a specific task; timers, which count regular clock pulses; counters, which count irregular pulses; and PWM, which is very useful in microcontroller applications such as lighting and motor control.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2024
A. Subero, *Programming PIC Microcontrollers with XC8*, Maker Innovations Series
https://doi.org/10.1007/979-8-8688-0467-0_7

7. Interfacing Actuators

Armstrong Subero[1]

(1) Moruga, Trinidad and Tobago

So far, we have covered input and output on the PIC microcontrollers as well as internal modules including the PWM module provided by the microcontroller. In this chapter, we apply the knowledge we learned thus far to control three common actuators in embedded systems: the DC motor, the stepper motor, and the servo motor. We incrementally examine these three actuators based on their ability to be controlled. We start with the DC motor, which has very coarse control of directionality (forward or backward) and is the easiest to control. Next, we look at the servo, which has a greater degree of ability to be controlled, allowing us to control movement to a predefined angle. Finally, the stepper motor has the finest level of granularity with control being down to a very specific degree size. Upon completing this chapter, you will be able to use actuators in robotic systems, appliances, and any other application you may have in mind.

Introducing Actuators

When we talk about actuators, what we are talking about are devices responsible for converting electrical, hydraulic, thermal, or pneumatic energy into mechanical motion we use to do work. You encounter actuators everywhere from vehicles and appliances to industrial machinery and advanced robots. Anyone who did any type of work in the industry may be able to tell you about hydraulic cylinders or pneumatic motors and persons who studied physics will be able to tell you about shape-memory actuators. Whatever their form or function, all actuators find use in controlling things like valves, automotive systems, manufacturing, medical devices, and any device that involves motion in some shape or form.

When we think of “actuation” in embedded systems, the DC motor comes to mind. The DC motor is known for its simplicity and is an excellent starting point in our exploration. Anytime you have simple motion in an embedded system, the DC motor is usually the actuator of choice you would begin your design with. Even modern general-purpose computing devices such as smartphones contain vibration motors (which are DC motors) for haptic feedback, providing vibration as opposed to rotational motion. However, there are other actuator types as we will explore in the next section.

Actuators in Embedded Systems

When introducing actuation into embedded systems, the three most common types of actuators are pneumatic, hydraulic, and electric. Of these three, electric-based actuation is the easiest and cheapest to design and prototype with. A wide spectrum of devices use electric motors, from electronic locks to toy cars and robots. Before we look at using an electric motor, let's discuss it a bit.

There are two types of electric motors. Some are powered by AC, and some are DC powered. For the applications, we will look at the DC variety, specifically the brushed DC motor.

DC motors work on the principle of current flowing through a magnetic field, thereby creating a force. Electric motors are used to convert electrical energy into mechanical energy. DC motors contain two input terminals, and applying a voltage across these terminals causes the motor shaft to spin. When a positive voltage is applied, the motor spins in one direction; when a negative voltage is applied, it spins in the other direction. To properly interface a DC motor to a microcontroller, we can use relays, MOSFETs, or transistors. The reason is simple:

If you connect a DC motor directly to the pin of your microcontroller, you will damage it!

Mind you, PIC microcontrollers are very difficult to destroy when compared to other microcontrollers. I have had PIC microcontrollers that were smoking and then worked for years after the problem was fixed, and still work! However, to ensure that you have a happy PIC microcontroller, do not connect the motor directly to an I/O pin.

You cannot connect a motor directly to a PIC microcontroller because an inductive load such as a motor requires a large amount of current. The I/O pins of a PIC microcontroller are incapable of supplying the current required by the motor. To interface small DC motors, such as small hobby motors or vibration motors commonly used for haptic feedback in embedded systems, I recommend using a simple transistor when prototyping. The reason is that when working with MOSFETs, extra precautions need to be taken to prevent static damage. So you can prototype with a transistor and use a MOSFET for your final design. Unless of course, the motor requires a large current in which case you can take precautions and use a MOSFET in your prototype as well.

The motors you will encounter when doing your designs typically have a diameter and length of a few inches. These motors run from about 3.3 v to 24 v and usually don't draw more than about 5−10 amps of current, though it is not uncommon to see motors that draw less than an amp being used in a variety of designs. Brushless DC motors typically draw more power, but when using them, we will usually use a special driver designed for them as well as a beefier power supply, and simple communication protocols will allow us to effectively control them.

When driving DC motors, something we must pay attention to is the stall current of the motor. The stall current of the motor is the maximum amount of current a motor draws when it is running but is prevented from rotating. When we connect our motor, it draws a certain amount of current to overcome the resistance and turn the load it's connected to, and when the motor is unable to turn either due to some external force or some mechanical constraint or failure, the amount of current it will try to draw will be much higher than usual.

This is dangerous because the motor is trying to produce enough torque to overcome the resistance it is experiencing and the excessive current draw it is facing can cause a lot of overheating and potential damage to the motor if it persists for an extended period. Thus, when selecting a driver for the motor, use one that is rated for the stall current of the motor and not the "no-load current" of the motor. Usually in the datasheet of the motor, you will find this information.

The process for turning a motor on and off is rather straightforward, and we can use the same processes we looked at for interfacing high-current inductive loads to PIC microcontrollers in our earlier chapter on input and output. Since motors can come from a variety of sources, any DC motor can work for this section. Ideally to follow along with the remaining DC motor section of this chapter, you will need two motors. The first is a 5 v hobby motor, and the other one I recommend is a 12 v motor with an encoder. The Pololu 131:1 37D metal gear motors are ideal. They have high torque and will help you gain a good understanding of how you can control DC motors that have a higher amperage than the few hundred milliamps or so you are accustomed to from hobby motors.

Simple On/Off Control

The first type of motor control we will look at is on/off control. In the realm of embedded systems motor control, simple on/off control with a DC motor is the most fundamental technique we can look at. The way we control a motor with on/off control is rather simple. We start and stop the rotation of the motor by supplying or cutting off power. Using our PIC microcontroller, we can use several methods. The simplest way is to use a transistor as a switch. BJT devices can be used for low-power DC motors, but when we want more power, we need to use MOSFETs. You can also use a relay to switch the motor.

The code for controlling a motor with on/off control is rather straightforward. It's the same program we use to vary the blinking of an LED. In this scenario, we set the specific pin connected to the transistor to be either high or low. Let's say we have a transistor connected to our pin. Setting the pin high allows current to flow through the transistor, powering the motor, and setting it low interrupts the current flow, effectively stopping the rotation of the motor. Let's look at Listing 7-1 to see the code for turning the motor on and off.

Figure 7-1 shows how you connect a DC motor to the microcontroller.

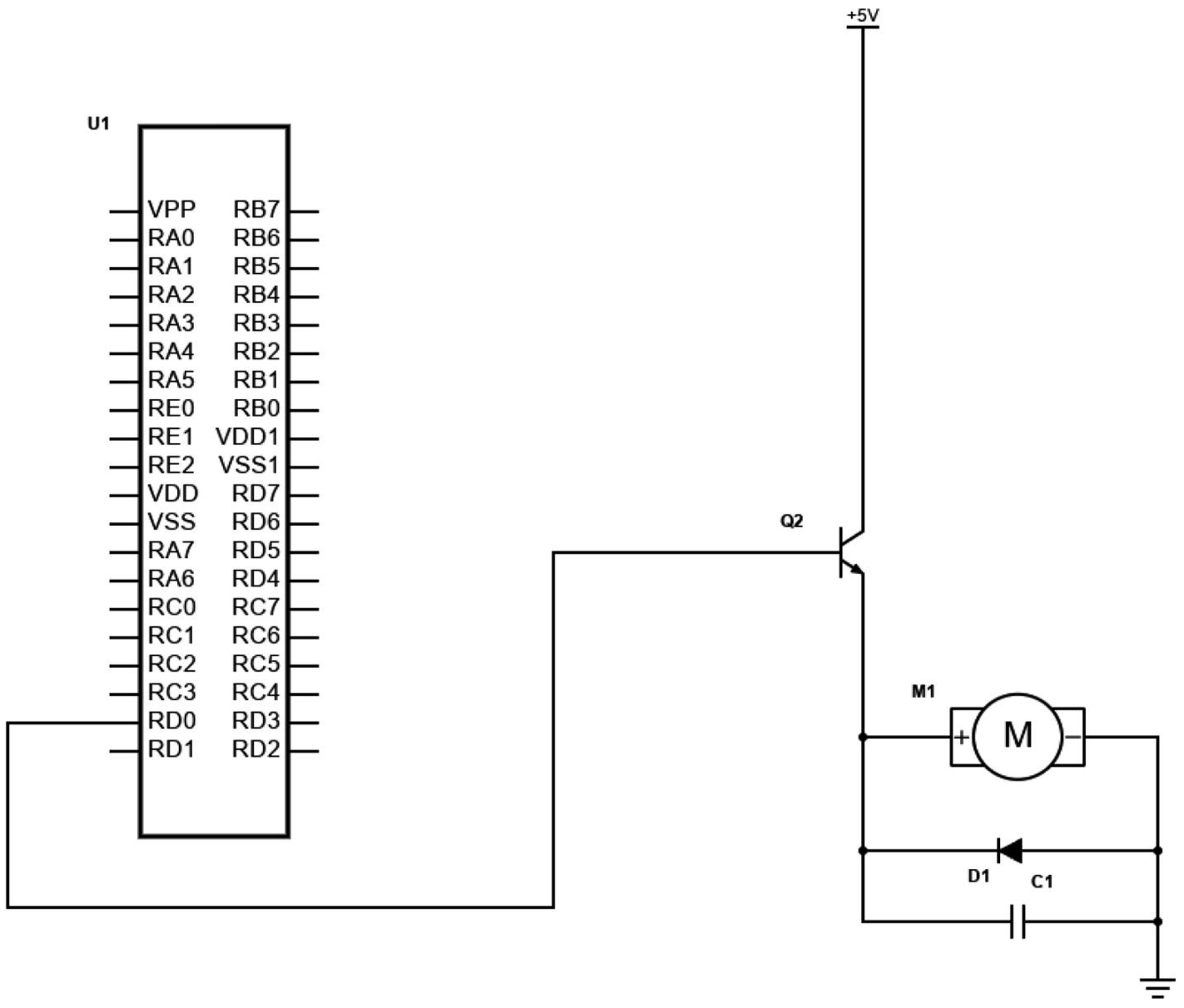


Figure 7-1 Connecting a DC motor to a PIC microcontroller

The value of the resistor, diode, and capacitors varies according to the size of the motor. For our purposes, the resistor can be 1k, the diode a standard 1N4001, and the capacitor a 0.1 uF ceramic type. The transistor can be a 2N2222A or a 2N3904; however, almost any NPN transistor can be substituted. Be sure to consult the datasheet for your motor. The code is rather simple, as shown in Listing 7-1.

```
/*
* File: Main.c
* Author: Armstrong Subero
* PIC: 16F1717 w/Int OSC @ 16MHz, 5v
* Program: 07_Motor
* Compiler: XC8 (v1.38, MPLAX X v3.40)
* Program Version: 1.0
*
*
* Program Description: This Program Allows PIC16F1717 to Turn
on a Motor
```

```
*
* Hardware Description: As per schematics
*
* Created November 4th, 2016, 1:00 PM
*/

/**********************************************************
*Includes and defines
**********************************************************/
#include "16F1717_Internal.h"

/*********************************************************
* Function: void initMain()
*
* Returns: Nothing
*
* Description: Contains initializations for main
*
* Usage: initMain()
**********************************************************/
void initMain(){
  // Run at 16 MHz
  internal_16();
  // Set PIN D0 as output
  TRISDbits.TRISD0 = 0;
}

/*********************************************************
* Function: Main
*
* Returns: Nothing
*
* Description: Program entry point
*****************************************************/
void main(void) {
    initMain();
   // Turn motor on for 5 seconds
    while(1){
        LATD0 = !LATD0;
        __delay_ms(5000);
    }
    return;
}
```

Listing 7-1 Driving a Motor with a PIC Microcontroller

This program should suffice for driving small motors for trivial applications. If we look at the code, the line responsible for turning the motor on is LATD0 = !LATD0. Just like in the LED program, this line drives the pin high for five seconds, which turns the transistor on. When the while loop runs again, the pin outputs low, which turns the transistor off for five seconds. You see even a simple application such as blinking an LED can be modified to perform other tasks in the real world. This is the power of microcontrollers; once you can perform simple actions, they can be easily modified or stacked together to perform more complex. The five-second on and off time is user determined, and you may modify it as you see fit. If you want to have the motor turn in the other direction, you could do this by modifying the circuit and simply switching the terminals of the motor. To do so, connect the positive terminal to the ground and connect the negative terminal to the emitter of the transistor.

Driving Motors with Power MOSFETs

Sometimes, however, you want a bit more power than what a bipolar transistor can provide; in such instances, you can opt to use a power MOSFET. We already talked about MOSFETs before, but I thought we could look at them in the context of driving motors. If we remember the way the MOSFET works, the charge on the gate will allow power to flow between the drain and source. Power MOSFETs are like ordinary MOSFETs, but they can handle a lot more power.

There are many power MOSFETs that we can select from and use. However, a popular device that can be used for driving 12 and 24 v motors is the IRF540 device, and that is the device we will use for our purposes. The IRF540 can handle up to 175 degrees Celsius operating temperature and can safely dissipate 50 watts of power with a maximum power dissipation of 150 watts and has a continuous drain current of 28 amps, though as the device gets hotter, this value drops to around 20 A. While this number sounds low, 28 A of current is quite a lot of current handling capability, and keep in mind that MOSFETs can be run in parallel. These devices are cheap, well known, and widely available, which makes them sort of a “jellybean” component. Sure, you can find other MOSFETs with better characteristics and current handling capability, but if its anything the recent chip shortage has taught us, it is that it’s better to design in a well-known part that is widely available than a part that is slightly better but can be susceptible to supply disruptions. Also, readers anywhere in the world should be able to get their hands on this component.

Contrary to what you may read in other articles (I have seen it browsing various online articles over the years), it is possible to drive the device with 5 v drive signals such as those from our PIC microcontroller; however, the total amount of current the MOSFET can switch will be reduced to around half (10 A or so), and the MOSFET will handle switching without heating up, especially if you are not switching at extremely high speeds. That 10 A switching current is a lot for most applications you will use in microcontroller-based systems, and at the current, you are reaching into the domain of power electronic systems; unless you are specifically designing devices for power applications, this part can cover the vast majority of your use cases.

There are logic-level power MOSFETs, the jellybean part being the IRLZ44N; however, it's limited to 55 V vs. the 100 V of the IRF540, though you do get higher current handling, 47 A, which drops closer to around 33 A as the device heats up. To drive the power MOSFET to drive a motor, we typically use a discrete transistor drive. The drive we will use consists of a discrete transistor, specifically the 2N2222A. We see the schematic for connecting our MOSFET to a microcontroller in Figure 7-2.

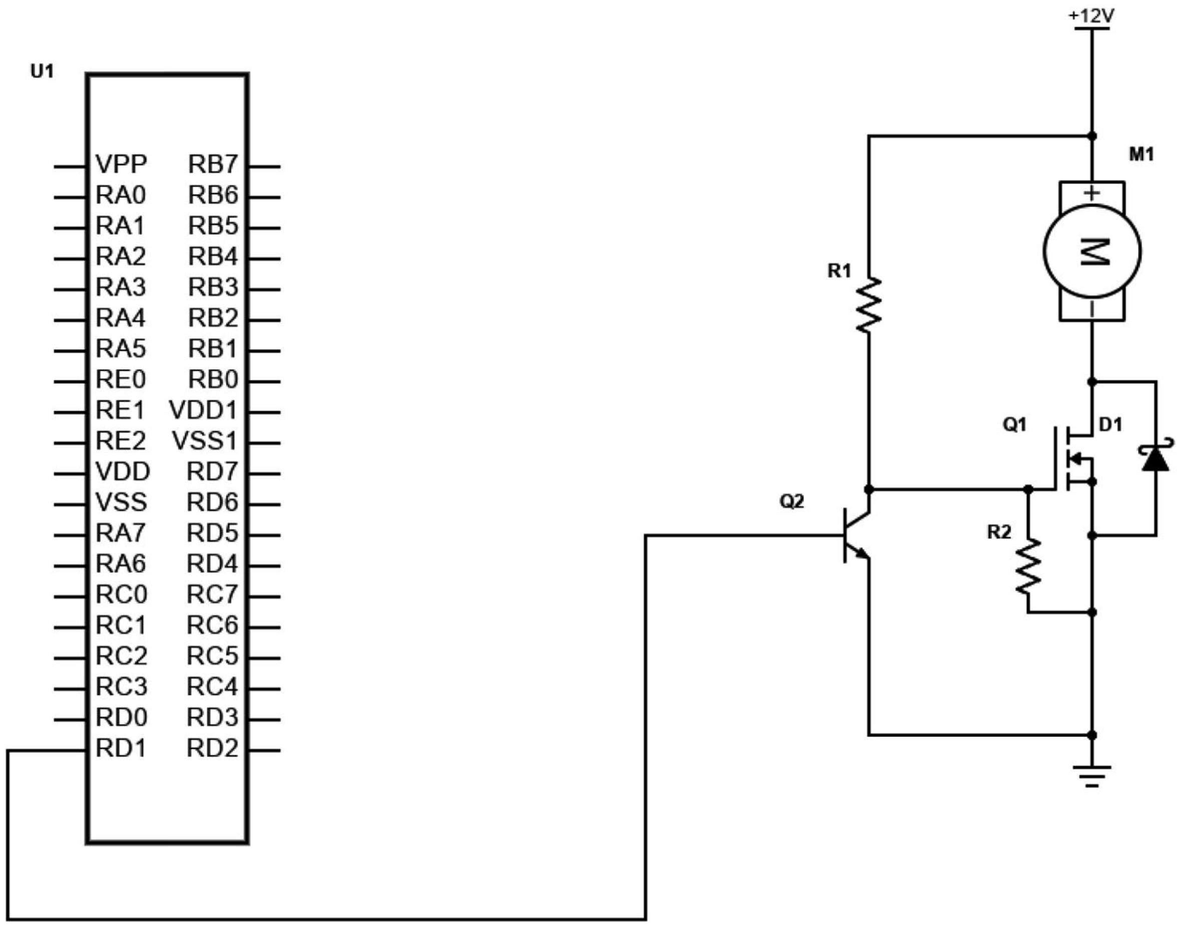


Figure 7-2 Connecting a power MOSFET to a microcontroller

In our circuit diagram, the transistor Q2 is used as the driver for the IRF540 MOSFET; the 2N2222A is recommended, though any NPN bipolar transistor with similar current handling capability can be used. R1 is a 1k pull-up resistor, and R2 is a 10k pull-down resistor that keeps the MOSFET in a defined state when it is not being driven. Q1 is out IRF540 MOSFET; if you browse the datasheet of your MOSFET, you will observe that there is a body diode; however, when driving inductive loads, it is highly recommended that you use an external diode. I recommend a Schottky diode because with it, you can reduce the reverse recovery effects associated with inductive loads. The MBR10100CT is a common Schottky diode that you can use for such an application. The motor we intend to run has a stall current of around 5 A so that diode is good enough since it has a peak forward surge current handling capability of 110 A, which should be enough for any inductive kick we can get from driving the motor.

We can write a small program that is suitable for driving the motor, which is given in Listing 7-2.

```
/*
* File: main.c
* Author: Armstrong Subero
* PIC: 16F1719 w/int OSC @ 32MHz, 5v
* Program: 31_Motor_On_Off
* Compiler: XC8 (v2.45, MPLAX X v6.15)
* Program Version: 1.2
*
* Program Description: This program allows for control of a
DC motor. The motor is turned on for 1 second and then off
for one second.
*
* Hardware Description: The 2N2222A transistor is used to
drive an IRF540 MOSFET that is then used to control a 131:1
gear motor rated for a 5A stall current. The base of the
2N2222A is connected to pin RD1
*
* Created November 4th, 2016, 1:00 PM
* Last Updated: January 01st, 2024, 6:57 PM
*/

/*********************************************************
*Includes and defines
********************************************************/
#include "PIC16F1719_Internal.h"

// Function to control the motor
/********************************************************
* Function: void initMain()
*
* Returns: Nothing
*
* Description: Function to control the motor
*
* Usage: controlMotor(1)
******************************************************/
void controlMotor(uint8_t state)
{
    // Assuming RD1 is connected to the base of the
transistor
    LATD1 = state;
}
```

```
/*******************************************************
* Function: void initMain()
*
* Returns: Nothing
*
* Description: Contains initializations for main
*
* Usage: initMain()
******************************************************/

void initMain()
{
    // Run at 16 MHz
    internal_16();

    ////////////////////
    // Configure Ports
    ///////////////////
    TRISDbits.TRISD1 = 0;
}

/*****************************************************
* Function: Main
*
* Returns: Nothing
*
* Description: Program entry point
****************************************************/

void main(void)
{
    initMain();

    while(1)
    {
        // Turn the motor on
        controlMotor(1);

        // Delay for 1 second
        __delay_ms(1000);

        // Turn the motor off
        controlMotor(0);

        // Delay for 1 second
        __delay_ms(1000);
```

```
      }
       return;
}
```

Listing 7-2 Program for Driving a Power MOSFET with a Microcontroller

In this program, we have our motor control function that turns the motor on and off; if you followed along in the last section, it's the same on/off control functionality, but the beefier driver circuit means we can now use a more powerful motor. Indeed, if you use the 131:1 geared motor from Pololu, you will find it nearly impossible to stop the rotation by holding it with a 12 v supply due to the amount of torque involved.

While on/off control can work for a variety of applications, there are a variety of problems with such an approach. The key one is the inrush current. When we start a motor at full speed, it experiences a surge of current we call inrush current, and the higher the voltage of the motor, the higher the inrush current can be. Since we are using a microcontroller, we can use intelligent software to limit the initial and operating speed of the motor. The way we do this is with Pulse Width Modulation (PWM), which we will look at in the next section.

PWM Speed Control

On/off control is useful in some applications; however, for the majority of applications where you will use a motor, you will want a way to control the speed. The simplest way to control motor speed is of course to use a resistor. If we use a resistor in series with a DC motor, the resistor will dissipate excess energy as heat leading to a very inefficient method of speed control. This is because when we apply Ohm's law, the resistor must restrict the flow of current to regulate the voltage. If we take V=IR, then as the current passes through the resistor, a voltage drop will be produced, and this voltage drop causes power to be dissipated as heat by the formula $P = I^2R$, giving a very inefficient linear method of speed control.

We need to find a better method of speed control. To do so, we will use Pulse Width Modulation (PWM), which will control the average voltage we send to the motor that regulates the speed. The microcontroller motor driver circuit generates a PWM signal, and the microcontroller controls the duty cycle of the PWM signal based on the desired speed or other control parameters.

Based on these PWM signals, we get something known as motor response, and the motor responds to the pulses as if it were receiving varying levels of voltage, although the actual supply voltage remains constant. For instance, at the full duty cycle, 100% of the motor receives the full supply voltage and operates at maximum speed, and as the duty cycle decreases, the resultant decreasing voltage slows down the motor.

PWM also allows the motor to run smoothly. The rapid switching of the voltage allows for smooth control of the motor speed as even though the motor is receiving a series of pulses, the inertia helps smooth out these changes, resulting in stable motion.

One of the only "gotcha" when using PWM-based control is the PWM frequency at which the motor is run. Typically, we run our motors within the 10 kHz to 30 kHz range. If we have a switching frequency that is too high, then we will have reduced efficiency of the control circuitry as there will generally be an increase in switching losses. A too-low frequency will produce a lot of audible noise; therefore, we tend to operate motors at least 20 kHz where practicable as the audio produced will be in the ultrasonic range, which will be silent to us. Keep in mind that some motors (like coreless DC motors) may have specific frequencies at which they operate. Generally, you would consult the datasheet for your device to determine the optimal frequency to limit facets like noise, efficiency, and motor performance.

We can use a similar circuit connection to allow for PWM-based motor control in Figure 7-3.

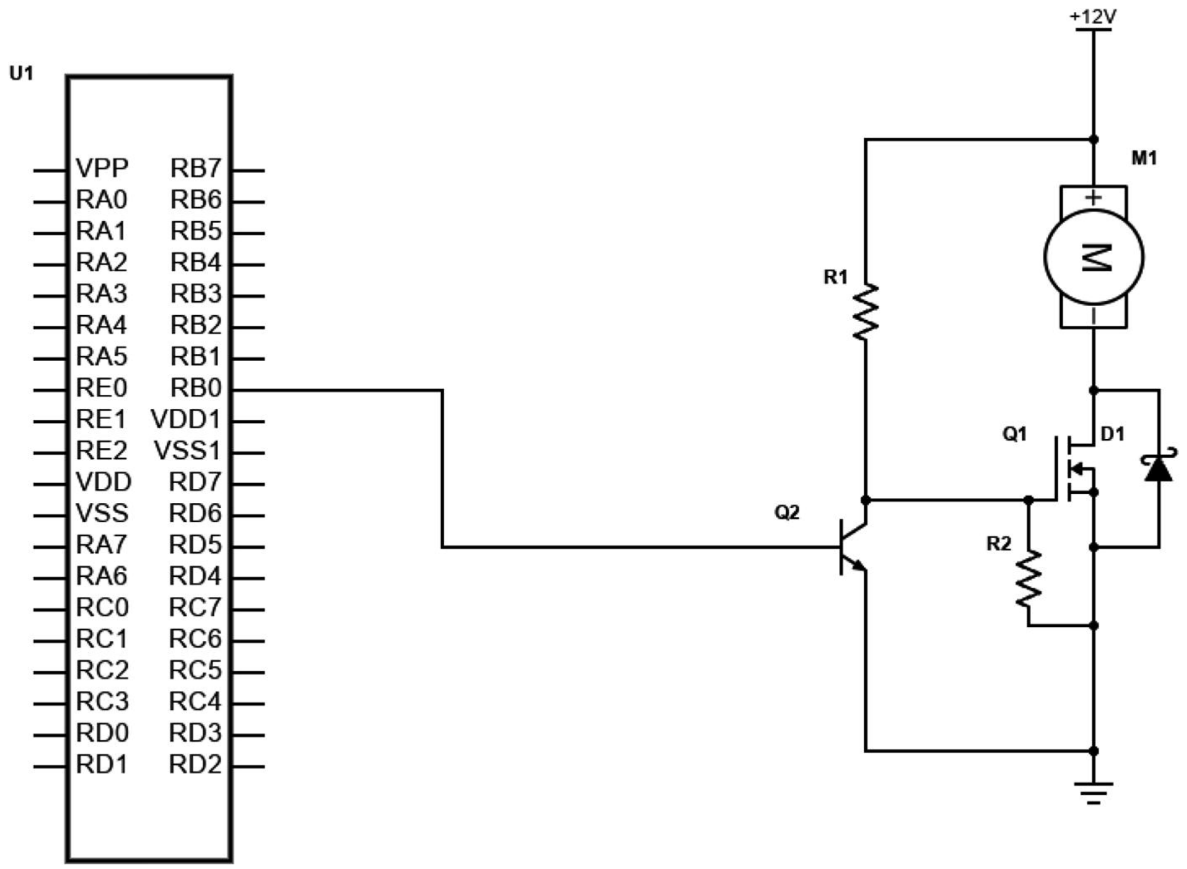


Figure 7-3 Motor speed control with PWM

The circuit is nearly identical to the one we used for simple on/off control; only this time we use the PWM-capable RB0 pin. We can then write a program to perform the actual Pulse Width Modulation given in Listing 7-3.

```
/*
* File: main.c
* Author: Armstrong Subero
* PIC: 16F1719 w/int OSC @ 32MHz, 5v
* Program: 32_Motor_PWM
* Compiler: XC8 (v2.45, MPLAX X v6.15)
* Program Version: 1.3
```

```
*
* Program Description: This program allows for control of a
DC motor using the PWM module. The motor speed is ramped up
then it ramped down.
*
* Hardware Description: The 2N2222A transistor is used to
drive an IRF540 MOSFET that is then used to control a 131:1
gear motor rated for a 5A stall current.
*
* Created November 4th, 2016, 1:00 PM
* Last Updated: January 11th, 2024, 7:31 PM
*/

/**********************************************************
*Includes and defines
*********************************************************/
#include "PIC16F1719_Internal.h"

/*********************************************************
* Function: void initMain()
*
* Returns: Nothing
*
* Description: Contains initializations for main
*
* Usage: initMain()
********************************************************/

void initMain(){
    // Run at 32 MHz
    internal_32();
    __delay_ms(100);

    ////////////////////
    // Configure Ports
    ///////////////////

    // Set PIN B0 as output
    TRISBbits.TRISB0 = 0;

    // Turn off analog
    ANSELB = 0;

    /////////////////////
    // Configure Timer6
    /////////////////////
    // Select PWM timer as Timer6 for CCP1
    CCPTMRSbits.C1TSEL = 0b10;

    // Enable timer Increments every 125 ns (32 MHz clock)
1000/(32/4)
    // Period = 256 x 0.125 us = 31.25 us

    //                          Crystal Frequency
    //    PWM Freq  = ---------------------------------------
--
    //                  (PRX + 1) * (TimerX Prescaler) * 4

    //    PWM Frequency = 32 000 000 / 256 * 1 * 4
    //    PWM Frequency = 31.250 kHz

    // Prescale = 1
    T6CONbits.T6CKPS = 0b00;

    // Enable Timer6
    T6CONbits.TMR6ON = 1;

    // Set timer period
    PR6 = 255;

    ///////////////////////////
    // Configure PWM
    //////////////////////////

    // Configure CCP1

    // LSB's of PWM duty cycle = 00
    CCP1CONbits.DC1B = 00;

    // Select PWM mode
    CCP1CONbits.CCP1M = 0b1100;

    ///////////////////////////////
    // Configure PPS
    //////////////////////////////

    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    PPSLOCKbits.PPSLOCKED = 0x00; // unlock PPS

    // Set RB0 to PWM1
    RB0PPSbits.RB0PPS = 0b01100;
    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    PPSLOCKbits.PPSLOCKED = 0x01; // lock PPS
}

/*******************************************************
* Function: void setPWMDutyCycle(uint8_t dutyCycle)
*
* Returns: Nothing
*
* Description: Sets the PWM duty Cycle
*
* Usage: setPWMDutyCycle(255)
******************************************************/

void setPWMDutyCycle(uint8_t dutyCycle)
{
    CCPR1L = dutyCycle; // Load the duty cycle value to CCP1
}

/******************************************************
* Function: void controlMotorPWM(uint8_t dutyCycle)
*
* Returns: Nothing
*
* Description: Controls the motor
*
* Usage: controlMotorPWM(dutyCycle)
*****************************************************/
void controlMotorPWM(uint8_t dutyCycle)
{
    // Set the PWM duty cycle
    setPWMDutyCycle(dutyCycle);
}

/*****************************************************
* Function: Main
*
* Returns: Nothing
*
* Description: Program entry point
*****************************************************/

void main(void)
{
    initMain();

    while(1)
    {
        // Increase duty cycle to ramp up motor speed
        for (uint8_t dutyCycle = 0; dutyCycle < 255;
dutyCycle++)
        {
            controlMotorPWM(dutyCycle);
            // Adjust delay as needed for smooth speed
changes
            __delay_ms(50);
        }

        // Decrease duty cycle to ramp down motor speed
        for (uint8_t dutyCycle = 255; dutyCycle > 0;
dutyCycle--)
        {
            controlMotorPWM(dutyCycle);
            // Adjust delay as needed for smooth speed
changes
            __delay_ms(50);
        }
    }
     return;
}
```

Listing 7-3 PWM Speed Control

If you looked at PWM control in the previous chapter, this program should seem familiar to you. In this program, once we set up and initialize the PWM, we create two functions. The first function, setPWMDutyCycle, is used to set the PWM duty cycle, and the controlMotorPWM function calls this function, which intuitively allows us to control the speed of the motor. This sort of abstraction makes our code a bit more portable so we can easily port it among microcontroller families.

Directional Control Small Motors

Directional control PWM is pivotal in motor applications where we need not only speed regulation but also controlling the motor's direction. The way that we usually employ directional control is with a bridge drive circuit. The simplest type of drive circuit is the half-bridge drive circuit. The half-bridge circuit contains two switching elements, which are in a configuration to control a load, in our case a motor. The half-bridge allows us to control our load in only one direction. A generic half-bridge circuit is given in Figure 7-4.

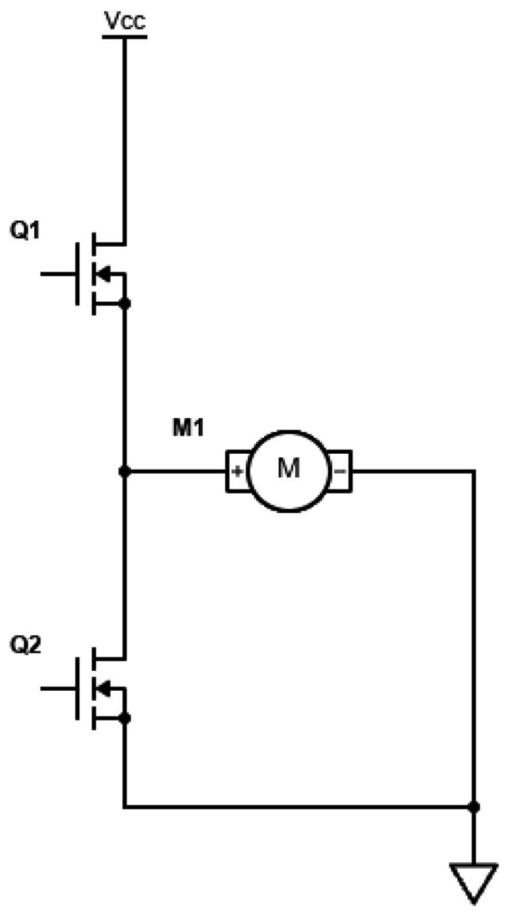


Figure 7-4 Generic half-bridge drive circuit

In the half-bridge drive, we usually use MOSFETs or transistors where one connects the load to the positive supply and the other connects the load to the negative supply. In the schematic in Figure 7-4, we need to use a driver at the base of Q1 and Q2 for a practical circuit. When Q1 is turned on, it allows the current to flow from the positive power supply to the load, and simultaneously Q2 remains off, creating an open circuit between the load and negative supply. So when Q1 is on and Q2 is off, current will flow through the motor in a specific direction, and when the states of Q1 and Q2 are reversed, the change directs the flow of current through the motor in the opposite direction, reversing the operation of the load. While the half-bridge is capable of driving the motor in both directions, it cannot do so simultaneously; thus, we need a way to allow for simultaneously allowing the motor to turn in both directions.

Usually, the way we control a motor in both directions we create what is known as a full-bridge drive. In the full-bridge drive configuration, the four switching elements are arranged to form a complete bridge topology. Full-bridge drives allow for bidirectional current flow by controlling the MOSFETs or transistors. You can find full-bridge drivers in a lot of power electronics circuits in various topologies. However, for our purposes, we are interested in the H-bridge for motor control applications. A generic H-bridge circuit is given in Figure 7-5.

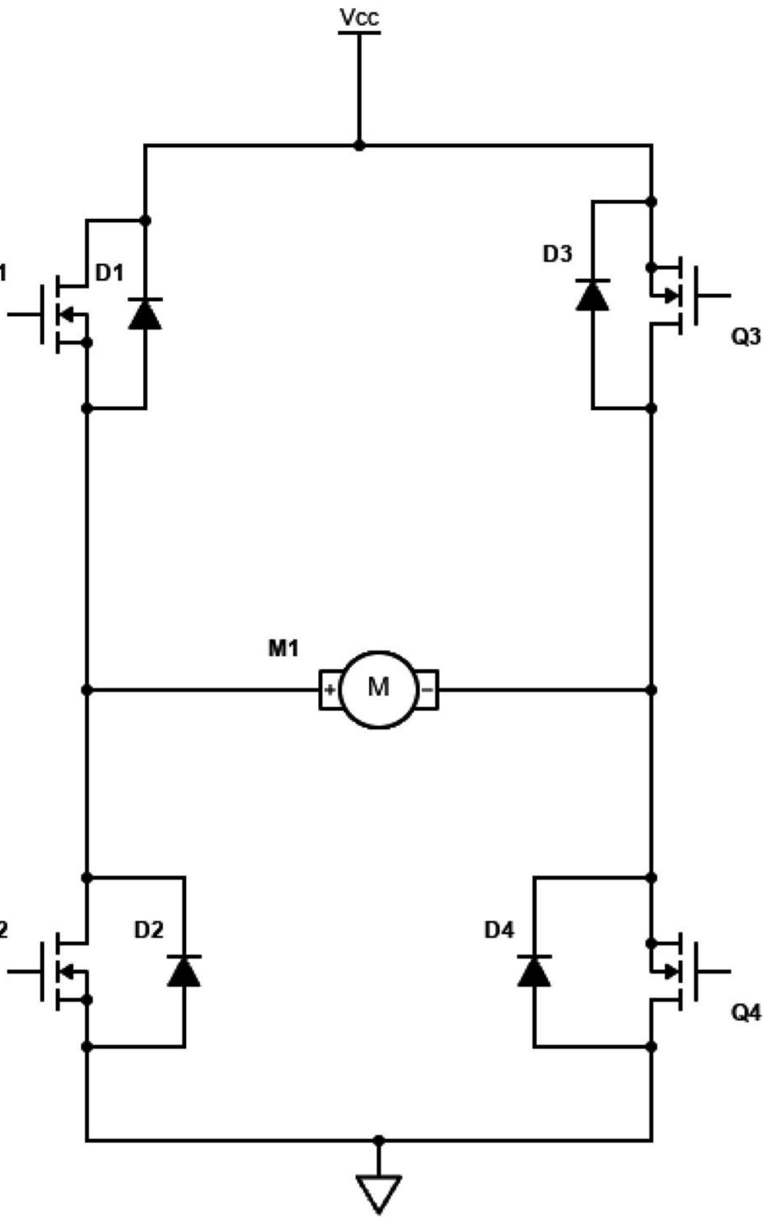


Figure 7-5 A generic H-bridge circuit

The H-bridge consists of four MOSFET or transistor switches, which are labeled as Q1, Q2, Q3, and Q4. These are arranged in pairs, and each pair controls the motor's connection to the power supply. Q1 and Q2 form one pair, and Q3 and Q4 form another pair. The way we control the motor's direction is that the H-bridge manipulates the switches in different configurations. To rotate the motor in one direction, Q1 and Q2 would be on, and Q3 and Q4 will be turned off. Current will flow from the power supply through Q1, the motor, and then through Q4 completing the circuit, which causes the motor to rotate in one direction. To cause the motor to rotate in the other direction, Q2 and Q3 are turned on while Q1 and Q4 are turned off. This reverses the flow of current through the motor, causing it to rotate in the opposite direction.

We can also provide braking functionality and freewheeling functionality. If we rapidly turn on both Q1 and Q3 or Q2 and Q4, we can get a short circuit across the motor, causing rapid deceleration due to the motor's back EMF (electromotive force). We can also do freewheeling when both pairs of switches are off and the motor terminals are left open, allowing the motor to freewheel spinning due to its inertia with minimal resistance. To use the H-bridge, we need a driver with control functionality, and we use a driver at base Q1, Q2, Q3, and Q4.

A better way though is by using an integrated circuit that has H-bridge functionality integrated into one package. The two common ICs that you will use are the L293D and SN754410NE ICs. The L293D is a quad half-H driver IC that can control the direction (bidirectional control) and provide drive currents for small to medium-sized DC motors or stepper motors. It features four half-H drivers, meaning it can control two motors bidirectionally (forward and reverse) and you can also drive a two-winding stepper motor. The L293D can handle currents up to 600 mA per channel and withstand peak currents up to 1.2 A per channel. It can also handle up to 36 V of voltages generally.

While the L293D is very popular, the SN754410NE is also a quad half-H driver designed for driving small to medium-sized DC motors and other inductive loads. It's almost identical to the L293D, making it more suitable for applications requiring slightly higher motor currents. It can handle up to 1 A per channel continuously and up to 2 A peak per channel. The code for using the SN754410NE H-bridge driver is given in Listing 7-4.

```
/*
* File: Main.c
* Author: Armstrong Subero
* PIC: 16F1717 w/Internal OSC @ 16MHz, 5v
* Program: I05_H_Bridge
* Compiler: XC8 (v1.38, MPLAX X v3.40)
* Program Version: 1.0
*
*
* Program Description: This demonstrates using an SN754410 H-
Bridge with a DC motor with a PIC microcontroller.
*
* Hardware Description: A generic brushed hobby DC motor is
connected to the SN754410 as per standard connections. The
PWM signals are emanating from RB0 and RB1 for forward and
reverse signals respectively.
*
*
* Created January 15th, 2017, 11:36 AM

/*************************************************************
*Includes and defines
************************************************************/
```

```
#include "16F1717_Internal.h"

/*******************************************************
* Function: void initMain()
*
* Returns: Nothing
*
* Description: Contains initializations for main
*
* Usage: initMain()
******************************************************/

void initMain(){
    // Run at 16 MHz
    internal_16();

    ////////////////////
    // Configure Ports
    ///////////////////

    // Set PIN B0 as output
    TRISBbits.TRISB0 = 0;

    // Set PIN B1 as output
    TRISBbits.TRISB1 = 0;

    // Turn off analog
    ANSELB = 0;

    /////////////////////
    // Configure Timer6
    /////////////////////

    // Select PWM timer as Timer6 for CCP1 and CCP2
    CCPTMRSbits.C1TSEL = 0b10;
    CCPTMRSbits.C2TSEL = 0b10;

    // Enable timer Increments every 250 ns (16MHz clock)
1000/(16/4)
    // Period = 256 x 0.25 us = 64 us

    //                          Crystal Frequency
    //    PWM Freq  = -------------------------------------
--
    //                  (PRX + 1) * (TimerX Prescaler) * 4
    //    PWM Frequency = 16 000 000 / 256 * 1 * 4
    //    PWM Frequency = 15.625 kHz

    // Prescale = 1
    T6CONbits.T6CKPS = 0b00;

    // Enable Timer6
    T6CONbits.TMR6ON = 1;

    // Set timer period
    PR6 = 255;

    //////////////////////////
    // Configure PWM
    /////////////////////////

    // Configure CCP1

    // LSB's of PWM duty cycle = 00
    CCP1CONbits.DC1B = 00;

    // Select PWM mode
    CCP1CONbits.CCP1M = 0b1100;

    // Configure CCP2

    // LSB's of PWM duty cycle = 00
    CCP2CONbits.DC2B = 00;

    // Select PWM mode
    CCP2CONbits.CCP2M = 0b1100;

    //////////////////////////////
    // Configure PPS
    /////////////////////////////

    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    PPSLOCKbits.PPSLOCKED = 0x00; // unlock PPS

    // Set RB0 to PWM1
    RB0PPSbits.RB0PPS = 0b01100;

    // Set RB1 to PWM2
    RB1PPSbits.RB1PPS = 0b01101;

    PPSLOCK = 0x55;
```

```
    PPSLOCK = 0xAA;
    PPSLOCKbits.PPSLOCKED = 0x01; // lock PPS

}

/**********************************************************
* Function: Main
*
* Returns: Nothing
*
* Description: Program entry point
*********************************************************/

void main(void) {
    initMain();

    while(1){
        // Run at 75% duty cycle @ 15.625 kHz for 5 sec

        // Forward
        CCPR1L = 192;
        CCPR2L = 0;

        __delay_ms(5000);

        CCPR1L = 0;
        CCPR2L = 0;

        __delay_ms(2000);

        // Reverse
        CCPR1L = 0;
        CCPR2L = 192;

        __delay_ms(5000);

        CCPR1L = 0;
        CCPR2L = 0;

        __delay_ms(2000);

    }
     return;
}
```

Listing 7-4 Using the SN754410 H-Bridge

In our setup, we rely on the SN754410NE full-bridge driver to handle the direction our motor moves. This driver works its magic by responding to the PWM signals sent out by the PIC MCU. These signals tell the driver which way the motor should spin. By teaming up the full-bridge driver with the MCU's PWM control, we get a smooth and accurate way to control the motor's direction, making our setup versatile for various tasks that need precise movement control. This simple circuit can find a variety of applications. Many simple robots, for example, can use this same circuit setup, utilizing multiple motors for control. Even industrial applications like an antenna movement control system or appliances use a similar setup to achieve their functionality. The schematic for interfacing the driver to our microcontroller is given in Figure 7-6.

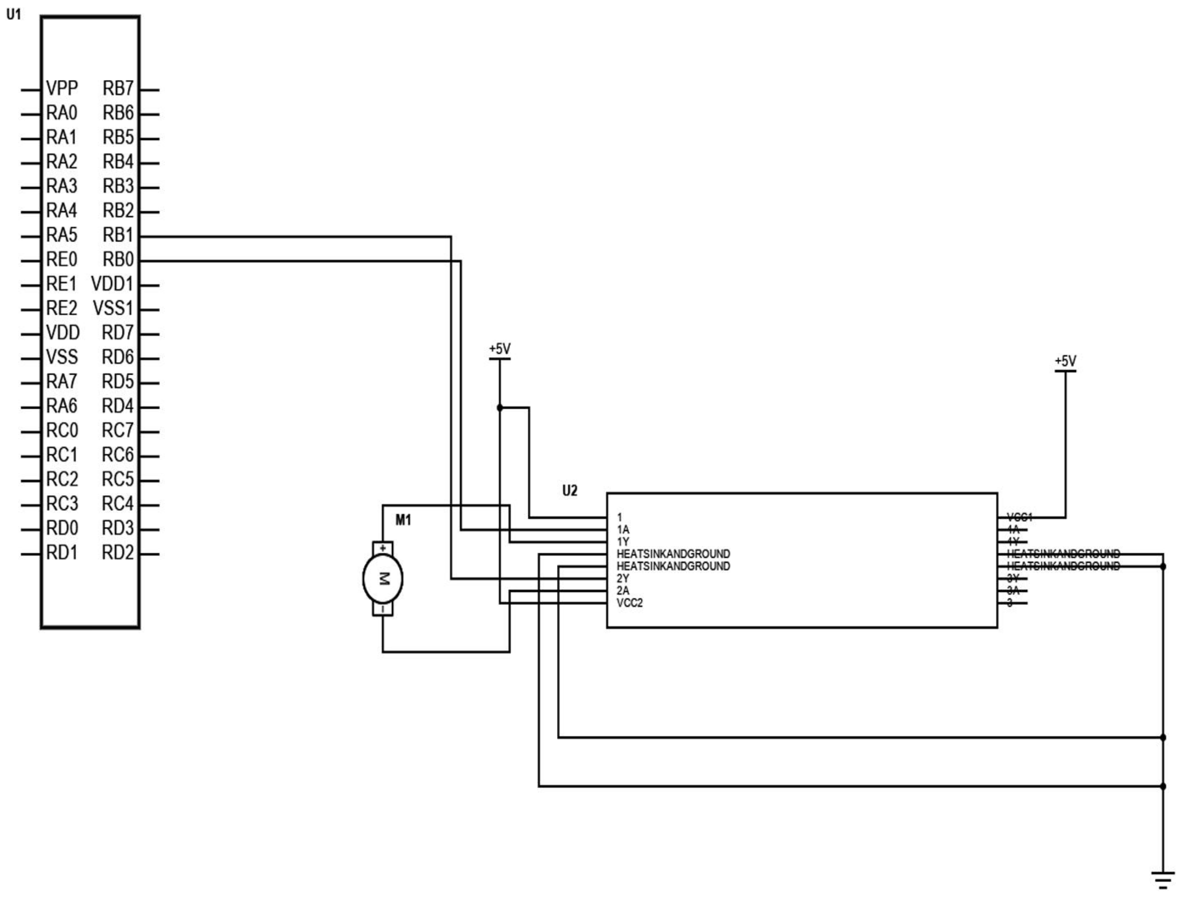


Figure 7-6 Connecting SN754410 to PIC MCU

We connect the motor to channel one of the ICs. Once the device is connected properly, you will observe the motor turning in one direction and then turning in another direction.

Directional Control Large Motors

If you require a bit more power for your motor control application, then there is a module that I recommend you use, which is the BTS7960 high-power H-bridge module. These modules are reliable as I have used them for many years, and they also can handle very high current stated to be up to 43 A and are great for driving 12 v and 24 v motors. This driver is shown in Figure 7-7.

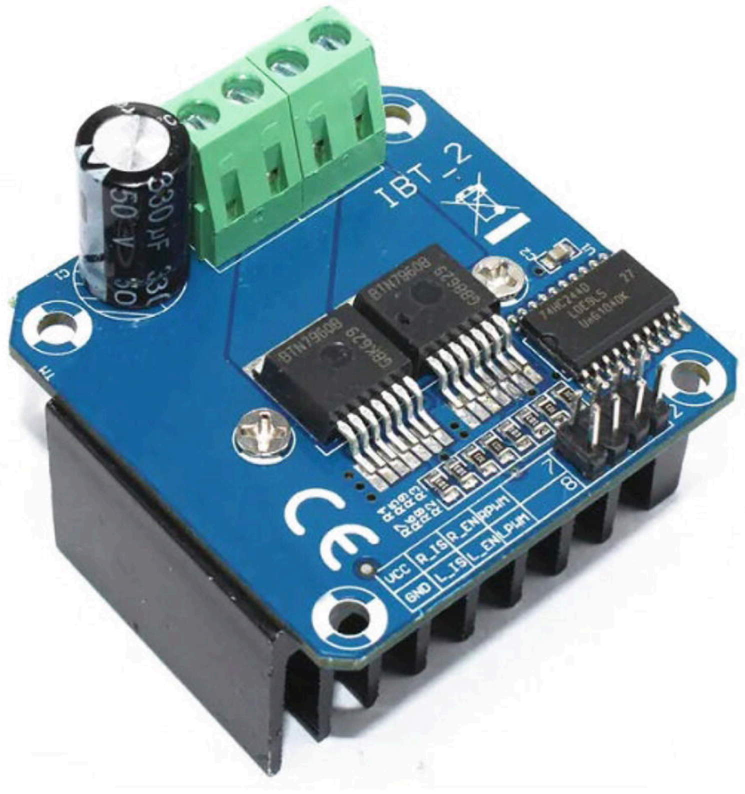


Figure 7-7 The BTS7960 motor driver module

If you prefer other options, you can check out Pololu as they have some great options for high-power motor drivers. We can connect the PIC MCU to the BTS7960 module as shown in Table 7-1.

Table 7-1 BTS7960 Module Pinouts

PIN No.	Function	Connection
1	RPWM	FORWARD PWM RB0
2	LPWM	REVERSE PWM RB1
3	R_EN	FORWARD DRIVE ENABLE RD0
4	L_EN	REVERSE DRIVE ENABLE RD1
5	R_IS	N/C
6	L_IS	N/C
7	VCC	5V

PIN No.	Function	Connection
8	GND	GND

Additionally, there are pins B- that connects to the negative power supply of the motor, which is ground, and B+ that connects to the positive power supply of the motor, and this can be anywhere from 6 to 27 volts. M+ and M- are connected to the terminals of the motor you want to control.

The program for controlling large motors using the BTS7960 driver with the PIC microcontroller is given in Listing 7-5.

```
/*
* File: main.c
* Author: Armstrong Subero
* PIC: 16F1719 w/int OSC @ 32MHz, 5v
* Program: 33_Motor_BTS7960
* Compiler: XC8 (v2.45, MPLAX X v6.15)
* Program Version: 1.0
*
* Program Description: This program allows a PIC
microcontroller to be interfaced to a BTS7960 motor driver
module
*
* Hardware Description: A BTS7960 motor driver is connected to
a PIC MCU as follows:
*
*                        RPWM - RB0
*                        LPWM - RB1
*                        R_EN - RD0
*                        L_EN - RD1
*                        R_IS - N/C
*                        L_IS - N/C
*                        VCC  - 5V
*                        GND  - GND
*
* The motor used to test is a 131:1 Pololu gear motor.
Recommended 4 1000 uF caps on power rails otherwise MCU may
fail to program
*
*
* Created February 26th, 2024, 11:34 PM
* Last Updated: February 26th, 2024, 11:34 PM
*/

/*************************************************************
*Includes and defines
```

```
*************************************************************/
#include "PIC16F1719_Internal.h"

/*************************************************************
* Function: void initMain()
*
* Returns: Nothing
*
* Description: Contains initializations for main
*
* Usage: initMain()
*************************************************************/

void initMain(){
    // Run at 32 MHz
    internal_32();

    ////////////////////
    // Configure Ports
    ///////////////////
    TRISDbits.TRISD1 = 0;

    // Set PIN B0 as output
    TRISBbits.TRISB0 = 0;

    // Set PIN B1 as output
    TRISBbits.TRISB1 = 0;

    // Turn off analog
    ANSELB = 0;

    // Control pins for BTS7960
    TRISDbits.TRISD0 = 0;
    TRISDbits.TRISD1 = 0;

    /////////////////////
    // Configure Timer6
    /////////////////////

    // Select PWM timer as Timer6 for CCP1 and CCP2
    CCPTMRSbits.C1TSEL = 0b10;
    CCPTMRSbits.C2TSEL = 0b10;

    // Enable timer Increments every 125 ns (32 MHz clock)
1000/(32/4)
    // Period = 256 x 0.125 us = 31.25 us
    //                          Crystal Frequency
    //    PWM Freq  = ----------------------------------------
-
    //                  (PRX + 1) * (TimerX Prescaler) * 4

    //    PWM Frequency = 32 000 000 / 256 * 1 * 4
    //    PWM Frequency = 31.250 kHz

    // Prescale = 1
    T6CONbits.T6CKPS = 0b00;

    // Enable Timer6
    T6CONbits.TMR6ON = 1;

    // Set timer period
    PR6 = 255;

    //////////////////////////
    // Configure PWM
    /////////////////////////

    // Configure CCP1

    // LSB's of PWM duty cycle = 00
    CCP1CONbits.DC1B = 00;

    // Select PWM mode
    CCP1CONbits.CCP1M = 0b1100;

    // Configure CCP2

    // LSB's of PWM duty cycle = 00
    CCP2CONbits.DC2B = 00;

    // Select PWM mode
    CCP2CONbits.CCP2M = 0b1100;

    //////////////////////////////
    // Configure PPS
    /////////////////////////////

    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    PPSLOCKbits.PPSLOCKED = 0x00; // unlock PPS

    // Set RB0 to PWM1
    RB0PPSbits.RB0PPS = 0b01100;
    // Set RB1 to PWM2
    RB1PPSbits.RB1PPS = 0b01101;

    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    PPSLOCKbits.PPSLOCKED = 0x01; // lock PPS
}

// enable reverse drive enable input
void leftEnable(void)
{
   LATDbits.LATD0 = 1;
}

// disable reverse drive enable input
void leftDisable(void)
{
    LATDbits.LATD0 = 0;
}

// enable forward drive enable input
void rightEnable(void)
{
    LATDbits.LATD1 = 1;
}

// disable forward drive enable input
void rightDisable(void)
{
    LATDbits.LATD1 = 0;
}

// set rightPWM duty cycle
void rightPWMDuty(uint16_t duty)
{
    CCPR1L = duty;
}

// set left PWM duty cycle
void leftPWMDuty(uint16_t duty)
{
    CCPR2L = duty;
}

/***************************************************************
* Function: void motorTurnLeft(uint16_t motorSpeed)
```

```
*
* Returns: Nothing
*
* Description: Turns the motor in the left direction
***********************************************************/
void motorTurnLeft(uint16_t motorSpeed)
{
   // enable both channels
   leftEnable();
   rightEnable();

   // turn off left PWM
   leftPWMDuty(0);

   // turn on right PWM
   rightPWMDuty(motorSpeed);
}

/*************************************************************
* Function: void motorTurnRight(uint16_t motorSpeed)
*
* Returns: Nothing
*
* Description: Turns the motor in the right direction
************************************************************/
void motorTurnRight(uint16_t motorSpeed)
{
   // enable both channels
   leftEnable();
   rightEnable();

   // turn off left PWM
   leftPWMDuty(motorSpeed);

   // turn on right PWM
   rightPWMDuty(0);

}

/*************************************************************
* Function: void motorStop(void)
*
* Returns: Nothing
*
* Description: Stops the motor rotation
*************************************************************/
```

```
void motorStop(void)
{
   // disable both channels
   leftDisable();
   rightDisable();

   // turn off left PWM
   leftPWMDuty(0);

   // turn off right PWM
   rightPWMDuty(0);
}

/*************************************************************
* Function: Main
*
* Returns: Nothing
*
* Description: Program entry point
*************************************************************/

void main(void) {
    initMain();

    while(1)
    {
        // turn the motor left
        motorTurnLeft(192);
        __delay_ms(3000);

        motorStop();
        __delay_ms(3000);

        // turn the motor right
        motorTurnRight(192);
        __delay_ms(3000);

        motorStop();
        __delay_ms(3000);
    }
     return;
}
```

Listing 7-5 Controlling Large DC Motors with the BTS7960 Driver

The program we have is very simple. The leftEnable() and rightEnable() functions activate the motor driver's left and right channels, respectively, allowing for movement in either direction, whereas the leftDisable() and rightDisable() functions deactivate these channels. The rightPWMDuty and leftPWMDuty functions are responsible for setting the PWM duty cycle, which controls the speed of the motor. A higher duty cycle increases the motor speed, and vice versa.

The motorTurnLeft and motorTurnRight functions provide a higher-level interface for controlling the motor's direction. To turn the motor left, the program enables the right channel and sets the left PWM duty cycle to zero, and for turning right, it does the opposite. These actions ensure that the motor rotates in the desired direction with the speed determined by the PWM duty cycle passed to these functions. The motorStop() function halts the motor by disabling both the left and right enable inputs and setting both PWM duty cycles to zero, ensuring an immediate stop.

In the main loop of the program, we have an infinite loop that allows the motor to turn left at a specified speed for three seconds, stop for three seconds, turn right at the same speed for three seconds, and then stop again for three seconds. This simple program can be used to test and control large DC motors such as those typically found in robotics and domestic applications.

Encoders

When controlling DC motors, sometimes you need a way to control not only the speed of the motor but you might also need to know the distance traveled by the motor or even get feedback on the speed of the motor. So far, we have been controlling motors without there being any feedback mechanism in place, but to introduce some feedback, we must find a method to do so.

The way we usually do this is with a quadrature encoder, which allows us to determine the number of rotations that the motor has taken. This is particularly useful when running DC motors that are connected to a wheel such as in a robot-based application. If you know the number of rotations that have taken place, you can also calculate things such as the velocity, acceleration, and even the current angle of the motor. Most motors have Hall Effect sensors within them to build the quadrature feature. The quadrature encoder usually has two channels. We can call these channels channel A and channel B. The 131:1 gear motor I used for developing these programs is one such motor with two channels. Each channel will generate digital signals with a 90-degree phase shift. When the motor rotates in a clockwise direction, the A channel will lead the B channel, and when the motor is rotated in a counterclockwise direction, the B channel will lead the A channel, which lets us determine what direction the motor is rotating.

Something we need to look at when working on encoders is the resolution of the encoder. We usually measure the resolution of the encoder in Counts Per Revolution or CPR. Each pulse of the encoder has two edges, a rising edge and a falling edge, which gives us a specific number of counts from the encoder output when the motor shaft completes one revolution.

The PIC microcontrollers provide interrupt-on-change capability (which we discussed in a previous chapter); we can use this to our advantage to read the quadrature encoder output.

To do this, we can set up two pins to read channel A and channel B of the encoder; we can then view these outputs verifying that we are reading the encoder channels. The program for using this interrupt-on-change mechanism to read the motor encoders is given in Listing 7-6.

```
/*
* File: main.c
* Author: Armstrong Subero
* PIC: 16F1719 w/int OSC @ 32MHz, 5v
* Program: 34_Motor_Encoder
* Compiler: XC8 (v2.45, MPLAX X v6.15)
* Program Version: 1.0
*
* Program Description: This program allows a PIC
microcontroller read the output of a quadrature encoder
*
* Hardware Description: A BTS7960 motor driver is connected to
a PIC MCU as follows:
*
*                        RPWM - RB0
*                        LPWM - RB1
*                        R_EN - RD0
*                        L_EN - RD1
*                        R_IS - N/C
*                        L_IS - N/C
*                        VCC  - 5V
*                        GND  - GND
*
* The motor used to test is a 131:1 Pololu gear motor.
Recommended 4 1000 uF caps on power rails otherwise MCU may
fail to program. The output channels of the quadrature encoder
A and B are connected to pins AN0 and AN1 respectively.
*
*
* Created February 27th, 2024, 10:20 PM
* Last Updated: February 27th, 2024, 10:20 PM
*/

/*************************************************************
*Includes and defines
************************************************************/
#include "PIC16F1719_Internal.h"
#include "EUSART.h"

// counters for the encoder
```

```
volatile int counterA = 0;
volatile int counterB = 0;

/************************************************************
* Function: void initMain()
*
* Returns: Nothing
*
* Description: Contains initializations for main
*
* Usage: initMain()
***********************************************************/

void initMain(){
    // Run at 32 MHz
    internal_32();
    __delay_ms(100);

    ////////////////////
    // Configure Ports
    ///////////////////
    TRISDbits.TRISD1 = 0;

    // Set PIN B0 as output
    TRISBbits.TRISB0 = 0;

    // Set PIN B1 as output
    TRISBbits.TRISB1 = 0;

    // Turn off analog
    ANSELB = 0;

    // Control pins for BTS7960
    TRISDbits.TRISD0 = 0;
    TRISDbits.TRISD1 = 0;

    /////////////////////
    // Configure Timer6
    /////////////////////

    // Select PWM timer as Timer6 for CCP1 and CCP2
    CCPTMRSbits.C1TSEL = 0b10;
    CCPTMRSbits.C2TSEL = 0b10;

    // Enable timer Increments every 125 ns (32 MHz clock)
1000/(32/4)
```

```
    // Period = 256 x 0.125 us = 31.25 us

    //                          Crystal Frequency
    //    PWM Freq  = ----------------------------------------
-
    //                  (PRX + 1) * (TimerX Prescaler) * 4

    //    PWM Frequency = 32 000 000 / 256 * 1 * 4
    //    PWM Frequency = 31.250 kHz

    // Prescale = 1
    T6CONbits.T6CKPS = 0b00;

    // Enable Timer6
    T6CONbits.TMR6ON = 1;

    // Set timer period
    PR6 = 255;

    //////////////////////////
    // Configure PWM
    /////////////////////////

    // Configure CCP1

    // LSB's of PWM duty cycle = 00
    CCP1CONbits.DC1B = 00;

    // Select PWM mode
    CCP1CONbits.CCP1M = 0b1100;

    // Configure CCP2

    // LSB's of PWM duty cycle = 00
    CCP2CONbits.DC2B = 00;

    // Select PWM mode
    CCP2CONbits.CCP2M = 0b1100;

    TRISBbits.TRISB2 = 0;
    ANSELBbits.ANSB2 = 0;

    /////////////////////////////
    // Configure Encoders
    /////////////////////////////
    TRISAbits.TRISA0 = 1;
    TRISAbits.TRISA1 = 1;
    ANSELAbits.ANSA0 = 0;
    ANSELAbits.ANSA1 = 0;

    // enable detection of falling edges on RA0
    IOCANbits.IOCAN0 = 1;
    IOCANbits.IOCAN1 = 1;

    // enable IOC interrupt
    INTCONbits.IOCIE = 1;

    // Enable global interrupt
    ei();

    //////////////////////////////
    // Configure PPS
    /////////////////////////////

    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    PPSLOCKbits.PPSLOCKED = 0x00; // unlock PPS

    // Set RB0 to PWM1
    RB0PPSbits.RB0PPS = 0b01100;

    // Set RB1 to PWM2
    RB1PPSbits.RB1PPS = 0b01101;

    RB2PPSbits.RB2PPS = 0x14;   //RB2->EUSART:TX;
    RXPPSbits.RXPPS = 0x0B;   //RB3->EUSART:RX;

    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    PPSLOCKbits.PPSLOCKED = 0x01; // lock PPS
}

// enable reverse drive enable input
void leftEnable(void)
{
   LATDbits.LATD0 = 1;
}

// disable reverse drive enable input
void leftDisable(void)
{
    LATDbits.LATD0 = 0;
}
```

```
// enable forward drive enable input
void rightEnable(void)
{
    LATDbits.LATD1 = 1;
}

// disable forward drive enable input
void rightDisable(void)
{
    LATDbits.LATD1 = 0;
}

// set rightPWM duty cycle
void rightPWMDuty(uint16_t duty)
{
    CCPR1L = duty;
}

// set left PWM duty cycle
void leftPWMDuty(uint16_t duty)
{
    CCPR2L = duty;
}

/************************************************************
* Function: void motorTurnLeft(uint16_t motorSpeed)
*
* Returns: Nothing
*
* Description: Turns the motor in the left direction
***********************************************************/
void motorTurnLeft(uint16_t motorSpeed)
{
   // enable both channels
   leftEnable();
   rightEnable();

   // turn off left PWM
   leftPWMDuty(0);

   // turn on right PWM
   rightPWMDuty(motorSpeed);
}

/**********************************************************
* Function: void motorTurnRight(uint16_t motorSpeed)
```

```
*
* Returns: Nothing
*
* Description: Turns the motor in the right direction
**********************************************************/
void motorTurnRight(uint16_t motorSpeed)
{
   // enable both channels
   leftEnable();
   rightEnable();

   // turn off left PWM
   leftPWMDuty(motorSpeed);

   // turn on right PWM
   rightPWMDuty(0);

}

/**********************************************************
* Function: void motorStop(void)
*
* Returns: Nothing
*
* Description: Stops the motor rotation
**********************************************************/
void motorStop(void)
{
   // disable both channels
   leftDisable();
   rightDisable();

   // turn off left PWM
   leftPWMDuty(0);

   // turn off right PWM
   rightPWMDuty(0);
}

/**********************************************************
* Function: Main
*
* Returns: Nothing
*
* Description: Program entry point
**********************************************************/
```

```
void main(void) {
    initMain();

    // Initialize EUSART module with 19200 baud
    EUSART_Initialize(19200);

    while(1)
    {
        __delay_ms(5000);

        motorTurnRight(192);
        __delay_ms(1000);

        motorStop();

        // Write encoder A output
        EUSART_Write_Text("EncoderA: ");
        EUSART_Write_Integer(counterA);
        EUSART_Write_Text("\t\t");

        // Write encoder B output
        EUSART_Write_Text("EncoderB: ");
        EUSART_Write_Integer(counterB);
        EUSART_Write_Text("\n");

        while(1);
    }
     return;
}

/************************************************************
* Function: __interrupt() isr(void)
*
* Returns: Nothing
*
* Description: Increments counters based on encoder output
***********************************************************/
void __interrupt() isr(void)
{
    if (INTCONbits.IOCIF)
    {
        if (IOCAFbits.IOCAF0)
        {
            IOCAFbits.IOCAF0 = 0;
            counterA++;
        }
        if (IOCAFbits.IOCAF1)
        {
            IOCAFbits.IOCAF1 = 0;
            counterB++;
        }
    }
}
```

Listing 7-6 Using Interrupt on Change to Read Motor Encoders

In our program, we set up the encoders by setting AN0 and AN1 as inputs; then we configure them for interrupt on change. The main loop of the program is straightforward. We wait five seconds; then we run the motor for one second; then we stop the motor, take the measured values from the encoder, and send them via UART.

The ISR we create is crucial for the dynamic response to encoder events. Upon detecting a falling edge on either of the encoder's output channels (A or B), signaled by the IOC flags, it increments the respective counter (counterA or counterB). This counting mechanism is vital for determining the position or speed of the motor based on the encoder's feedback. This program can be adapted to be used on any quadrature-type encoder system.

PID Control

PID (Proportional-Integral-Derivative) loops are a feedback mechanism widely used in industrial control systems. Robotics and automation systems in particular use this method to regulate processes and maintain setpoints. Think of an industrial heater that needs to maintain a precise temperature and keep it constant over a long period. Devices like 3D printers with a heated bed use PID controllers to maintain a constant bed temperature, showing the versatility of the PID controller.

The PID loop has three parts: the P or proportional component, the I or integral component, and the D or derivative component. The three facets of the PID loop work together to provide a very simple but robust control system. The P term responds in direct proportion to the present error, aiming to reduce the error quickly.

So let's say you are trying to reach a temperature of 30 degrees; this 30-degree temperature is our setpoint. The P term responds by immediately acting on the current difference between the actual temperature and the desired 30 degrees. It will apply a correction that is proportional to this instant error, making rapid adjustments to drive the temperature closer to the setpoint. While the P term can reduce the present error, it can lead to overshoot or oscillations around the target due to its responsiveness to the current error alone. This is where the I component steps in.

The integral term I continuously sums the error values over time and produces an output to correct accumulated errors, which we call the steady-state error. Essentially in the context of our example, if the temperature remains consistently above or below 30 degrees over time, the I term will gradually accumulate this sustained error and apply a corrective action; this accumulation is known as the sum of errors and is crucial as it allows small, prolonged deviations from the setpoint to be corrected, gradually nudging the system toward the desired temperature. The I term cannot be too strong as if it is too aggressive, it can cause a sluggish response that causes instability in the system.

The derivative term D will predict how the error will change over time and act to dampen sudden changes, reducing overshoot and oscillations. In our example of reaching and maintaining our temperature, the D term will anticipate and manage abrupt temperature changes. So let's imagine our heater is now turned on and is ramping up, rapidly approaching the 30-degree setpoint; then the D term, which is sensitive to this swift change, applies a damping effect, counteracting the system's momentum to prevent overshooting the setpoint. By calculating the rate of change, it anticipates the future trend of the temperature and applies a corrective action to smoothen the approach, ensuring a more precise and stable convergence toward the desired temperature. If this term is too aggressive, however, it can act as a noise amplifier and can cause instability in the loop.

So basically, the PID controller constantly compares the desired setpoint (which is the target value) with the current measure value. The P term will generate an output based on the current error, and a larger error will lead to a higher corrective action. The I term will accumulate the error over time and will apply corrective action to eliminate any offset (specifically persistent offsets) between the setpoint and the actual value, and the derivative term considers the rate of change of the error, anticipating rapid changes and counteracting them, to prevent overshoots and oscillations. These three terms work together to try to generate a control signal that adjusts the system to reduce the error toward zero.

For all its complexity, the generic PID loop looks something like this:

```
error = setpoint - measured_value
integral = integral + error * dt
derivative = (error - previous_error) / dt
output = Kp * error + Ki * integral + Kd * derivative
previous_error = error
```

In this snippet, the proportional term P is Kp * error, our integral term I is "integral = integral + error * dt", and the D term is calculated as "(error – previous_error) / dt" representing the change in error concerning time and is multiplied by the derivative gain "Kd" and added to the output for a complete PID loop. You will observe that a "dt" parameter is present in our algorithm. This "dt" (delta time) represents the time interval between successive iterations of the loop. This parameter is crucial for accurately calculating the integral and derivative components of the PID formula. The integral component requires dt to accumulate the error over time correctly, ensuring that the controller compensates for past errors at the right rate. Similarly, the derivative component uses it to determine the rate of change of the error, predicting future error trends based on how quickly the error is currently evolving.

So, for example, let's say we wanted to maintain the motor rotation at a certain speed. We can use the PID algorithm given previously and just change the main of our encoder program given in Listing 7-6 to get the program given in Listing 7-7.

```
void main(void) {
    initMain();

    // Initialize PID constants
    const float Kp = 2.0;
    const float Ki = 0.5;
    const float Kd = 1.0;

    // Initialize PID variables
    float setpoint = 100.0; // Desired position or speed
    float measured_value = 0.0;
    float error = 0.0;
    float integral = 0.0;
    float derivative = 0.0;
    float previous_error = 0.0;
    float output = 0.0;

    // Sample time
    float dt = 0.01; // Adjust based on your
timer  configuration

    __delay_ms(5000);

    while(1)
    {

        // Read the current position or speed from the
encoder
        measured_value = counterA;

        // Calculate the PID error term
        error = setpoint - measured_value;

        // Calculate the integral term
        integral += error * dt;

        // Calculate the derivative term
        derivative = (error - previous_error) / dt;

        // Calculate the PID output
        output = Kp * error + Ki * integral + Kd *
derivative;

        // Update the motor speed based on the PID output
        motorTurnRight(output);
        // Store the current error as the previous error for
the next iteration
        previous_error = error;

        // Delay for a bit before next iteration
        __delay_ms(10); // Adjust delay to match your dt if
necessary
    }
     return;
}
```

Listing 7-7 The PID Control Program

The program defines constants for the PID controller (Kp, Ki, Kd), which represent the proportional, integral, and derivative gains, respectively. These gains determine how aggressively the PID controller responds to errors in position or speed. It also initializes variables to hold the PID calculation components. This loop operates continuously, executing a series of steps to dynamically adjust the motor's operation based on feedback from an encoder. Initially, the loop begins by reading the current value from the encoder, which represents the motor's actual position or speed (the measured value). This reading is crucial as it serves as the basis for determining how far off the motor is from where it's intended to be, which is essential for PID control.

The loop then proceeds to calculate the error by subtracting this measured_value from the setpoint, which is the target value the motor is supposed to achieve. This error is a critical input for the PID calculations, representing the discrepancy between the motor's current state and the desired outcome. Following the error calculation, the loop updates the integral term by adding to it the product of the current error and the time elapsed (dt) since the last update. This accumulation of error over time helps the controller to counteract any ongoing discrepancies that are not corrected by the proportional term alone.

Next, the derivative term is calculated by determining the rate at which the error is changing, done by subtracting the previous error from the current error and dividing by dt. This term helps predict future error trends, allowing the controller to preemptively counteract them, thus improving the system's responsiveness and stability. With all three PID components calculated, the loop computes the output by combining these terms, each weighted by their respective coefficients (Kp for proportional, Ki for integral, and Kd for derivative). This output then dictates how the motor's control signal should be adjusted to minimize the error, effectively instructing the motor to speed up, slow down, or maintain its current speed to align with the setpoint.

Finally, before the loop iterates again, it updates the previous_error with the current error to prepare for the next cycle. This ensures that the derivative calculation in the subsequent iteration is based on the most recent error change. The loop also includes a brief delay (__delay_ms(10)) to regulate the execution speed, ensuring that the PID calculations occur at a consistent rate, aligned with the predetermined dt.

While this rudimentary PID control algorithm would require some refinement before it can be integrated into an end application, it does provide a starting point for if you need to use PID control for DC motor applications.

Servo Motor

Now that you have a fair understanding of the DC motor, let's look at the servo motor. There are many different types and sizes of servos. There are AC servos typically used for industrial applications and DC servos commonly used for smaller applications. Some servos are used exclusively for robots and are known as robotic servos. They have ridiculous torque for their size.

When we speak of torque, we are talking about the maximum force the servo can output. The servo typically has three wires. One wire is connected to power, the other to ground, and the last is known as the signal wire and is connected to the pin of the microcontroller. We will not go into the intricacies of the internals of how servo motors are constructed. For our purposes, all we need to know is that a servo contains control circuitry inside. Therefore, all we need to do is tell this control circuitry what position to move the servo motor to.

To tell the servo motor what position to go to, we send a pulse on the signal wire. According to the width of the pulse (essentially how long the pulse is high), the motor will turn to the desired location. A servo motor can either move to one of three angles: 0 degrees, 90 degrees, or 180 degrees. This type of servo is typically called a hobby servo as it is commonly used in hobbyists' projects, robotics, and various DIY applications.

Though you may think these are a different type of motors, they consist of a DC motor, gears, control circuitry, and a position feedback system. They operate in a closed-loop control system maintaining a desired position by receiving control signals. A pulse width modulated signal is used to set the desired position or speed of the motor. This pulse width signal is controlled by sending PWM signals with specific pulse widths. The servo expects the PWM signals with a pulse width between 1 ms and 2 ms, repeated every 20 ms, giving an overall frequency of around 50 Hz. While this will be sufficient to drive the majority of off-the-shelf hobby motors, you will need to consult your datasheet to be sure. Typically, a 1 ms pulse will let the servo be at one extreme, either extremely left or right; a 1.5 ms pulse will center the servo; and a 2 ms pulse will carry it to the other extreme. By controlling the pulse width in these ranges, we can precisely control the position of the servo.

Many of these motors also incorporate a potentiometer that provides feedback to the control circuitry, allowing the servo to adjust its position accurately. You connect the servo motor as shown in Figure 7-8.

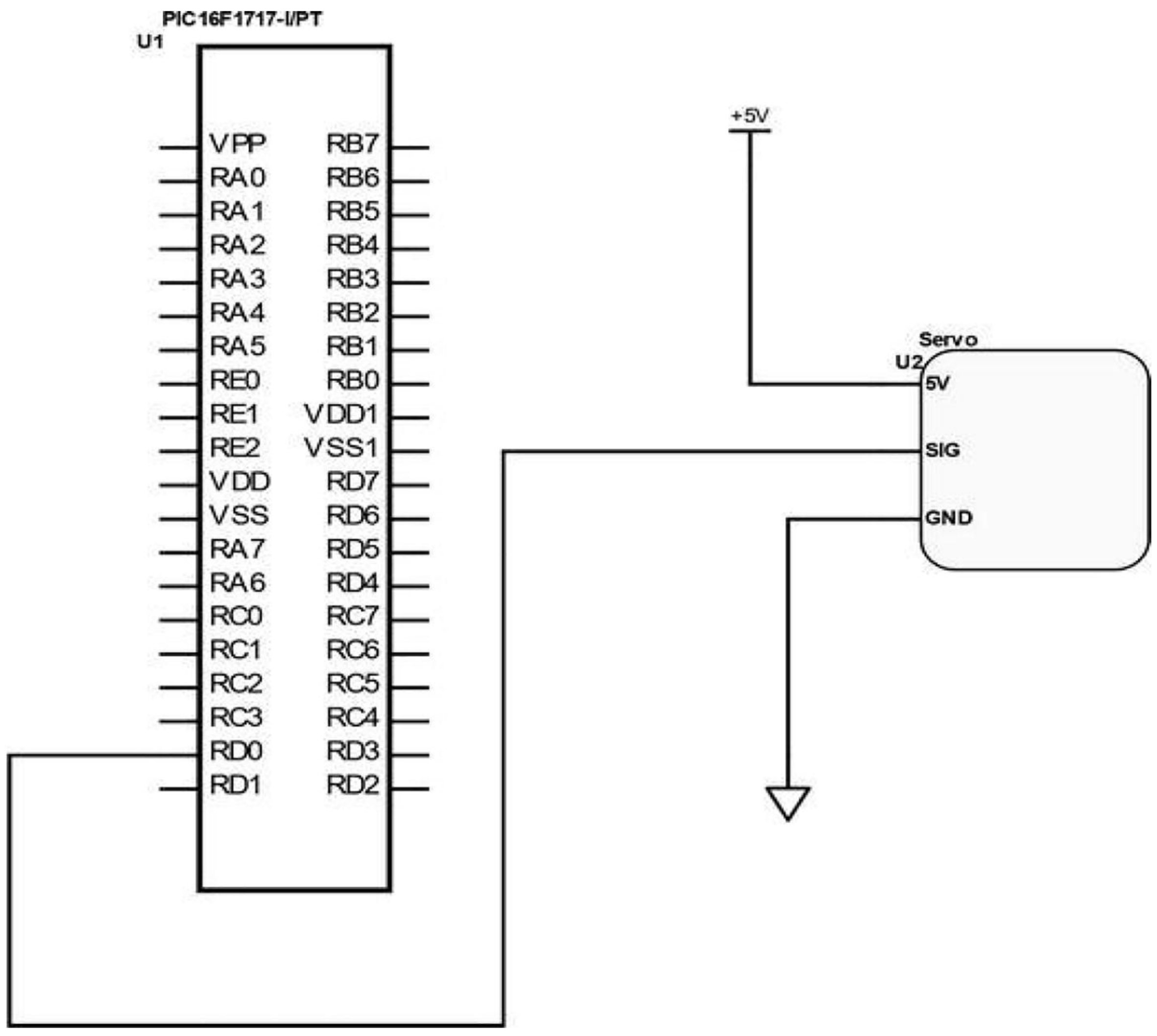


Figure 7-8 Connecting a servo motor to the PIC microcontroller

Create a file called Servo.h and use the code in Listing 7-8.

```
/*
* File: Servo.h
* Author: Armstrong Subero
* PIC: 16F1717 w/Internal OSC @ 16MHz, 5v
* Program: Header file to control a standard servo
* Compiler: XC8 (v1.38, MPLAX X v3.40)
* Program Version 1.0
*
* Program Description: This program header will allow you
control a standard servo.
*
*
* Created on January 14th, 2017, 9:15 PM
*/
```

```
/***********************************************************
*Includes and defines
***********************************************************/
#include "16F1717_Internal.h"

void servoRotate0(); //0 Degree
void servoRotate90(); //90 Degree
void servoRotate180(); //180 Degree
```

Listing 7-8 Servo Header File

Now you have to create a source file called Servo.c, which contains the code in Listing 7-9.

```
/*
* File: Servo.c
* Author: Armstrong Subero
* PIC: 16F1717 w/Int OSC @ 16MHz, 5v
* Program: Library file to configure PIC16F1717
* Compiler: XC8 (v1.38, MPLAX X v3.40)
* Program Version: 1.0
*
* Program Description: This Library allows you to control a
standard servo
*
* Created on January 14th, 2017, 10:41 PM
*/
#include "16F1717_Internal.h"
/***********************************************************
* Function: void servoRotate0()
*
* Returns: Nothing
*
* Description: Rotates servo to 0 degree position
*
**********************************************************/
void servoRotate0()
{
unsigned int i;
for(i=0;i<50;i++)
{
RD0 = 1;
__delay_us(700);
RD0 = 0;
__delay_us(19300);
}
}
```

```
/*******************************************************
* Function: void servoRotate90()
*
* Returns: Nothing
*
* Description: Rotates servo to 90 degree position
*
*******************************************************/
void servoRotate90() //90 Degree
{
unsigned int i;
for(i=0;i<50;i++)
{
RD0 = 1;
__delay_us(1700);
RD0 = 0;
__delay_us(18300);
}
}
/********************************************************
* Function: void servoRotate90()
*
* Returns: Nothing
*
* Description: Rotates servo to 180 degree position
*
********************************************************/
void servoRotate180() //180 Degree
{
unsigned int i;
for(i=0;i<50;i++)
{
RD0 = 1;
__delay_us(2600);
RD0 = 0;
__delay_us(17400);
}
```

Listing 7-9 Servo Source File

Listing 7-10 provides the main code.

```
/*
* File: Main.c
* Author: Armstrong Subero
* PIC: 16F1717 w/Internal OSC @ 16MHz, 5v
* Program: I02_Servo_Motor
* Compiler: XC8 (v1.38, MPLAX X v3.40)
* Program Version: 1.0
*
*
* Program Description: This demonstrates using a servo on a
pic microcontroller
*
* Hardware Description: A MG90s microservo is connected to
PIN RD0
*
* Created January 14th, 2017, 10:30 PM
*/
/************************************************************
*Includes and defines
**********************************************************/
#include "16F1717_Internal.h"
#include "Servo.h"
*********************************************************
* Function: void initMain()
*
* Returns: Nothing
*
* Description: Contains initializations for main
*
* Usage: initMain()
**********************************************************/
void initMain(){
   // Run at 16 MHz
   internal_16();

   ////////////////////
   // Configure Ports
   ///////////////////

   // Make D0 Output
   TRISDbits.TRISD0 = 0;

   // Turn off analog
   ANSELD = 0;
}

/***********************************************************
* Function: Main
*
* Returns: Nothing
```

```
*
* Description: Program entry point
*******************************************************************/
void main(void) {
initMain();
// Rotate servo 0 - 90 - 180
while(1){
    servoRotate0();
    servoRotate90();
    servoRotate180();
    return;
  }
}
```

Listing 7-10 Servo Source File

Once all goes well, the servo shaft will move along a 180-degree arc at 90-degree increments, starting at 0 degrees.

One caveat I should mention about actually using servo motors is that if you buy the cheap variety or even the high-quality plastic ones, they may not start at 0 degrees and end at 180 degrees.

The servo motors we have discussed so far have a limited range of motion, but there is one exception to this, which is the continuous rotation servo. It can rotate a full 360 degrees. To accomplish this, these motors usually use the PWM signal pulse width to determine the speed and direction of the motor. When we send a neutral position pulse width, the servo stops, increasing the pulse width in one direction increases the speed of rotation in that direction, and decreasing the pulse width from the neutral position causes rotation in the opposite direction, with the speed controlled by the pulse width. This property makes them ideal for rotating cameras or sensors that continuously need to scan or track. We can look at a program to control such a servo using Pulse Width Modulation in Listing 7-11.

```
/*
* File: main.c
* Author: Armstrong Subero
* PIC: 16F1719 w/int OSC @ 8MHz, 5v
* Program: 25_Servo
* Compiler: XC8 (v2.45, MPLAX X v6.15)
* Program Version: 1.2
*
* Program Description: This controls a servo to turn one
direction and then to turn the other direction using PWM
*
* Hardware Description: An continuous rotation servo is
connected to PINB0
*
* Created November 4th, 2016, 1:00 PM
* Last Updated: November 26th, 2023, 4:30 AM
*/

/*******************************************************
*Includes and defines
******************************************************/
#include "PIC16F1719_Internal.h"
*
/******************************************************
* Function: void initMain()
*
* Returns: Nothing
*
* Description: Contains initializations for main
*
* Usage: initMain()
*****************************************************/

void initMain(){
    // Run at 8 MHz
    internal_8();

    ////////////////////
    // Configure Ports
    ///////////////////

    // Set PIN B0 as servo output
    TRISBbits.TRISB0 = 0;

    // Turn off analog
    ANSELB = 0;

    /////////////////////
    // Configure Timer6
    /////////////////////

    // PR6 249;
    PR6 = 249;

    // TMR6 0;
    TMR6 = 0x00;

    // Clearing IF flag.
    PIR2bits.TMR6IF = 0;

    // T6CKPS 1:64; T6OUTPS 1:1; TMR6ON on;
    T6CON = 0x07;

    // Enable Timer6
    T6CONbits.TMR6ON = 1;

    //////////////////////////
    // Configure PWM
    /////////////////////////

    // Set the PWM1 to the options selected in the User
Interface

    // CCP1M PWM; DC1B 3;
    CCP1CON = 0x3C;

    // CCPR1L 124;
    CCPR1L = 0x7C;

    // CCPR1H 0;
    CCPR1H = 0x00;

    // Selecting Timer 6
    CCPTMRSbits.C1TSEL = 0x2;

    //////////////////////////////
    // Configure PPS
    /////////////////////////////

    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    PPSLOCKbits.PPSLOCKED = 0x00; // unlock PPS

    // Set RB0 to PWM1
    RB0PPSbits.RB0PPS = 0b01100;

    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    PPSLOCKbits.PPSLOCKED = 0x01; // lock PPS
}

/***********************************************************
* Function: Main
*
* Returns: Nothing
*
* Description: Program entry point
```

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

void PWM1_LoadDutyValue(uint16_t dutyValue)
{
   // Writing to 8 MSBs of pwm duty cycle in CCPRL register
    CCPR1L = ((dutyValue & 0x03FC)>>2);

   // Writing to 2 LSBs of pwm duty cycle in CCPCON register
    CCP1CON = (CCP1CON & 0xCF) | ((dutyValue & 0x0003)<<4);
}

void main(void) {
    initMain();

    uint16_t duty;

    while(1){
        PWM1_LoadDutyValue(375);

        __delay_ms(2000);

        PWM1_LoadDutyValue(5);

        __delay_ms(2000);
    }
     return;
}
```

Listing 7-11 Using a Continuous Rotation Servo

In this program, we use PWM to control the servo by adjusting the duty cycle. The servo will rotate in one direction for two seconds and then rotate in the opposite direction for two seconds, creating a continuous back-and-forth motion.

Stepper Motor

The stepper motor is the next topic of our actuator trio. Stepper motors are crucial to industrial applications and robotics. Stepper motors are used in many devices, including printers and CNC control, where very precise and controlled movement is needed.

The stepper motor has multiple coils, which, when organized together, are called phases. Each phase is turned on sequentially, causing the stepper motor to rotate. The distance between each step is known as the step angle. Two major windings exist for the coils of the stepper motor. One is known as the unipolar and the other as the bipolar. The basic difference between the two is that in the unipolar stepper motor, there is a coil with a center tap per phase, whereas a bipolar motor has one winding per phase. In this book, we focus on the unipolar variety.

There are three main ways to drive a stepper motor. There is the wave drive, the full-step drive, and the half-step drive.

In the wave drive mode, only a single phase is activated at a time. This results in the wave drive mode using less power than other modes but at the cost of loss of torque. The half-step drive mode alternates between two phases on and a single phase on. This increases angular resolution but lowers the torque. The full-step drive mode allows for two phases to be on at a time, thus allowing for maximum torque.

Thus, when designing with stepper motors, there is a trade-off between torque, power consumption, and angular resolution. The popular method for driving stepper motors though is to use the full-step drive mode; therefore, this is the method we will use in this book. To drive the stepper motor, we must use a driver to give the stepper motor the power it needs. For small stepper motors, you can use H-bridges or Darlington transistor arrays. We will use the Darlington array since from my experience they are simpler to use.

The IC we will use that contains these packages is the ULN2003A. The ULN2003A contains seven Darlington pairs and will be adequate for our purposes. The ULN2003A includes fly-back diodes and can work up to 50 V with a single output providing 500 mA. These specifications make it a popular choice for driving small stepper motors. The motor we will use in this example is a four-phase unipolar stepper motor rated at 5 v. The motor also has a holding torque of 110 g-cm and a step angle of 7.5 degrees and requires a current of 500 mA. Motors with similar specifications can be purchased from a variety of online suppliers; just ensure you buy one where the datasheet is easily accessible; otherwise, you may not know which wire does what. The circuit in Figure 7-9 indicates how you connect a stepper motor to the microcontroller.

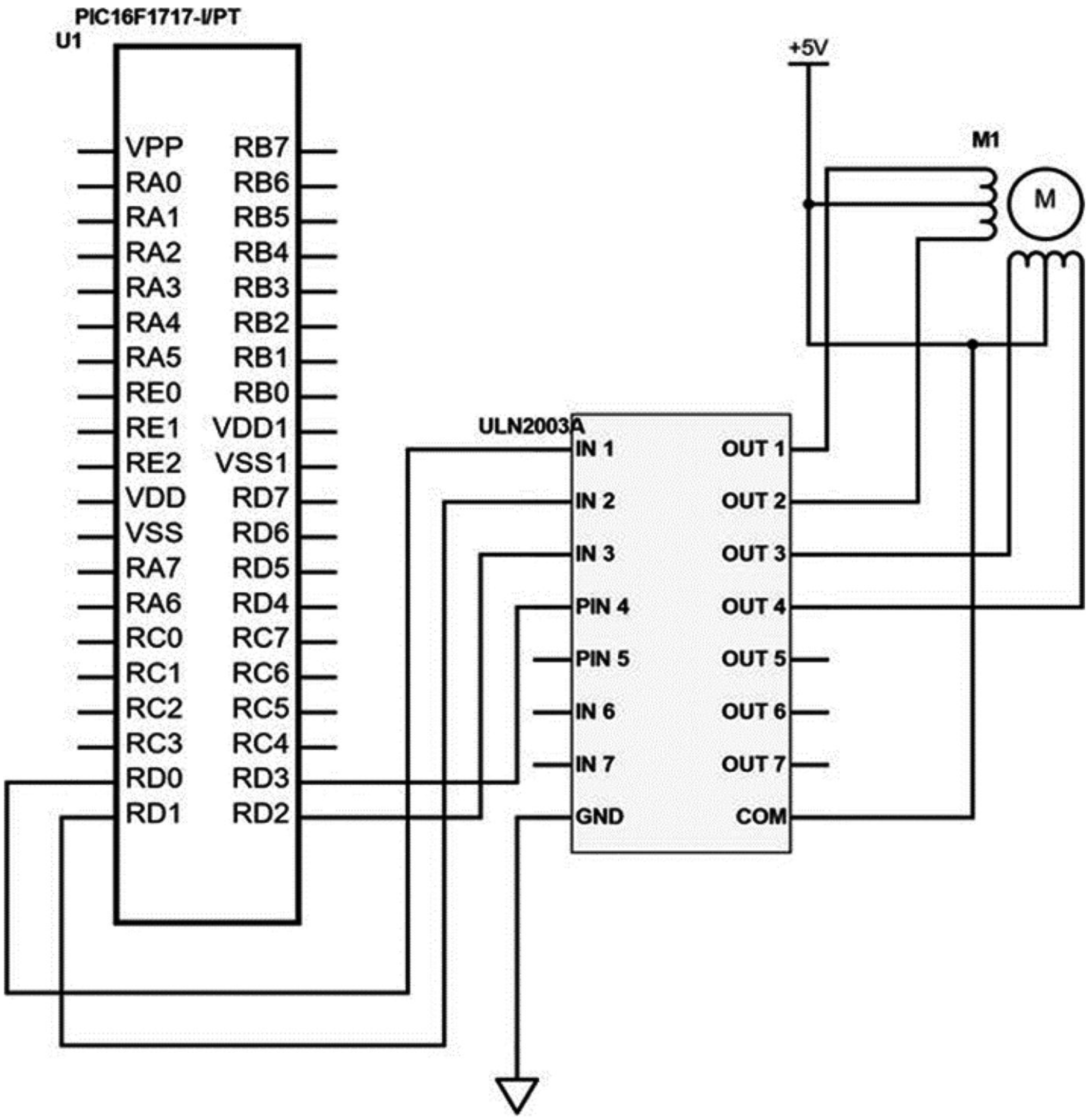


Figure 7-9 Connecting a stepper motor to a microcontroller

The code in Listing 7-12 is rather straightforward and drives the stepper in full-step mode.

```
/*
* File: Main.c
* Author: Armstrong Subero
* PIC: 16F1717 w/Internal OSC @ 16MHz, 5v
* Program: I03_Servo_Motor
* Compiler: XC8 (v1.38, MPLAX X v3.40)
* Program Version: 1.0
*
*
```

```
* Program Description: This demonstrates using a unipolar
stepper motor with a pic microcontroller.
*
* Hardware Description: A Sinotech 25BY4801 4 phase unipolar
stepper motor is connected to PORT D of the microcontroller
with a ULN2003A Darlington transistor array used as a driver
for the stepper motor.
*
* Connections are as follows:
*
* The ULN2003A is connected to the microcontroller as
* shown below:
*
* IN1 --> RD0
* IN2 --> RD1
* IN3 --> RD2
* IN4 --> RD3
*
* The Stepper motor is connected to the ULN2003A as
* follows:
*
* White = OUT1
* Black = OUT2
* Red = OUT3
* Yellow = OUT4
*
* Created January 15th, 2017, 12:05 AM
*/
/*************************************************************
*Includes and defines
*************************************************************/
#include "16F1717_Internal.h"

/************************************************************
* Function: void initMain()
*
* Returns: Nothing
*
* Description: Contains initializations for main
*
* Usage: initMain()
***********************************************************/
void initMain(){
  // Run at 16 MHz
  internal_16();
  ////////////////////
  // Configure Ports
  ///////////////////

  // Make D0 Output
  TRISD = 0;

  // Turn off analog
  ANSELD = 0;
}
/**********************************************************
* Function: Main
*
* Returns: Nothing
*
* Description: Program entry point
**********************************************************/
void main(void) {
  initMain();

  while(1){
   LATD = 0b00001001; // Step 1
   __delay_ms(500);
     LATD = 0b00000101; // Step 2
     __delay_ms(500);

     LATD = 0b00000110; // Step 3
     __delay_ms(500);

     LATD = 0b00001010; // Step 4
     __delay_ms(500);
   }
   return;
}
```

Listing 7-12 Driving a Stepper Motor in Full-Step Mode

If you look in the main while loop in Listing 7-12, you'll notice that the lower four bits of PORTD keep changing. In case you are wondering what is going on here, here is how it works. Remember we said that in full-drive step mode, the motor turns on two phases at a time? Well, if you revisit the schematic in Figure 7-9, you will notice that each phase of the stepper motor is connected to a pin of the microcontroller. Now I mentioned that the coils are center-tapped in the unipolar variety stepper motor, and if you look at your schematic, this is shown in Figure 7-9. Therefore, two phases need to be turned on at a time, since in actuality current only flows in half the winding of the stepper motor at a time. Thus, if you look at the code, you will notice that two output pins are high at a time. This ensures that by energizing half windings of the two coils within the stepper motor alternately, you get the full 360-degree rotation of the motor.

If you are still having trouble understanding how this works, there are excellent resources on the Web that go in depth into the intricacies of the operation of stepper motors. In this book, however, you can safely ignore all the details of the operation. As long as you verify with your datasheet that you have connected the stepper motor as shown in Figure 7-9, this code will work with unipolar stepper motors of any size.

Do not worry if your stepper motor runs hot. Stepper motors use a lot of current, and they use the same amount of current when they are stationary and running, so do not panic; this is completely normal.

Conclusion

In this chapter, we looked at using actuators with the PIC microcontroller. The first actuator we looked at is the DC motor. We looked at controlling the DC motor using on/off, PWM, and PID control and also controlling large DC motors. We also looked at using standard and continuous rotation DC motors and, finally, using stepper motors. The information in this chapter provides a solid foundation for you to use DC actuators.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2024
A. Subero, *Programming PIC Microcontrollers with XC8*, Maker Innovations Series
https://doi.org/10.1007/979-8-8688-0467-0_8

8. USART, SPI, I2C, and Communication Protocols

Armstrong Subero[1]

(1) Moruga, Trinidad and Tobago

In this chapter, we look at using serial communication protocols. In the realm of embedded systems and microcontroller applications, the ability to communicate effectively with external devices is fundamental. Serial communication is the lifeblood of embedded systems, allowing microcontrollers to interact with many external devices and peripherals. The most ubiquitous of these are USART, SPI, and I2C, which I will be explaining in this chapter. We will embark on a journey to demystify these protocols, breaking down their functionalities, configurations, and applications. The applications we cover using sensors, GPS, GSM, and a host of other things once we have learned how to use these protocols will give you hands-on experience in using these devices in your applications. So grab a bottle of water and sit down. This will be a long one. By the end of this chapter, you will get a solid grasp of serial communications setting the stage for exciting projects and innovations in your embedded systems journey. Let’s get started!

Understanding Serial Communication

When you think about communication in embedded systems, and if you were to come up with a protocol for communication, the one that seems most obvious and intuitive is parallel communication. Parallel communication is a method whereby multiple bits of data are transmitted simultaneously over several wires. It is a parallel method of data transfer where each dedicated wire transmits a bit of information. Since each data word is sent at the same time, the number of parallel lines you need to use would depend on the word size of the data. For example, if you are doing 8-bit data transmission, then you need eight parallel lines to transmit all eight bits simultaneously. In addition to the data lines, you will also have several control lines for things like clocking, addressing, and control, and these signals synchronize and manage the flow of data across the parallel channels.

All these data and control lines create a requirement for multiple wires or conductors and higher data widths such as 64 bits; it becomes challenging to design and implement systems using parallel communication. In practice, the only feasible applications I see for parallel interfaces are in the use of parallel SRAM and DDR interfaces as well as for displays when you want a high refresh rate or need to provide features such as video playback. Many PIC microcontrollers include a Parallel Master Port (PMP) for parallel communication.

While parallel communication does have its place and will continue to do so for the foreseeable future, for most applications, the communication protocol that you would use is serial communication.

With serial communication, we sequentially transmit data bit by bit over a single communication channel. With serial communication, we send our data over two wires or in some instances a single wire. We usually also need a clock line to ensure both the sender and receiver have accurate data transmission. What serial data lines take away in physical complexity they add in software complexity. The added overhead of serial communication results in it having a lower data transfer width compared with parallel communication. With modern systems, however, and the speed they run at for most applications, serial communication suffices. The most common serial communication protocols are UART, SPI, and I2C. Each of these has its own rules and applications that we will explore in the following sections.

USART

The Universal Synchronous Asynchronous Receiver Transmitter (USART) is my favorite communication protocol. The reason it is my favorite is because it is the simplest to use. USART is a preferred communication protocol in embedded systems owing to its simplicity and versatility.

An embedded systems designer can understand every detail of the USART protocol. While this is true for other protocols, once you understand how USART works, understanding other serial protocols will be much easier.

The first step to using the USART module is the configuration. To use USART in a microcontroller, you begin by configuring its settings. This includes setting things like the baud rate, the parity number of data bits, stop bits, and other parameters according to the communication requirements.

On the transmission end, we send the data. To send USART data, we prepare the data to be transmitted in the microcontroller’s buffer. The data can be anything from sensor readings to instructions for other devices. You then load the prepared data into the USART transmit buffer register. The MCU will then initiate transmission by enabling the USART’s transmitter. The transmission then starts with the USART sending a start bit (logical low) to indicate the beginning of the data frame. The data bits are then sent one by one following the configured sets we set in the configuration sending data LSB to MSB order.

Sometimes you may see USART being written as just “UART.” For our applications, they do pretty much the same thing. USART is just an enhanced UART protocol, as the missing “S” (synchronous) requires clocking to be synchronous and adds a little complexity to your design. Therefore, we will use the USART module asynchronously in compliance with KISS.

The USART module onboard the PIC microcontroller can be used synchronously or asynchronously. When used asynchronously, all the communication takes place without a clock. This saves an I/O pin as in this mode, only the transmit and receiver lines are required. The asynchronous mode is the type of communication we will use.

Synchronous mode allows the module to be used with a clock and is not as widely used; thus, we will not discuss it in this book.

An important consideration for USART is the baud rate. The baud rate of the USART specifies the rate at which the USART transfers data. A baud rate of 2400 means that the USART can transfer a maximum of 2400 bits per second.

Another facet of USART you can look at is flow control. Flow control allows us to play a crucial role in managing the flow of data between devices. USART supports both hardware and software flow control mechanisms to prevent issues like data overrun and ensure seamless communication. While hardware flow control exists, USART usually uses software flow control to prevent issues like data overrun and ensure seamless communication. Usually, this involves two special characters using special characters. XON for starting transmission and XOFF for stopping transmission embedded within the data stream are the typical characters used in USART.

Error detection and correction along with framing and stop bits are integral components of serial communication protocols, ensuring the reliability and accuracy of data transmission. In the context of serial communication, errors can arise due to noise, interference, and other factors. Error detection mechanisms such as parity checking and checksums are employed to identify discrepancies between transmitted and received data. If an error is detected, corrective measures can be taken to retransmit the data or apply the error correction algorithms.

The framing and stop bits are crucial to asynchronous serial communications; each data byte is framed by a start bit followed by the actual data bits, an optional parity bit for error checking, and one or more stop bits. The start bit is the beginning of the data byte, and the stop bit indicates the end. The framing ensures that the receiver can accurately interpret the transmitted data by synchronizing its internal clock with the incoming bitstream. Together, error detection and correction mechanisms and framing with stop bits contribute to the robustness and integrity of serial communications.

Throughout the book, the main reason we use USART for debugging is because USART is simple to set up and all modern popular operating systems support COM ports. COM ports stand for Communication Ports, and they are used for serial communication between a computer and an external device. Besides industrial computers, you rarely see COM ports today. Instead in most applications, the way you use USART is with virtual COM ports (VCPs). Since USB is ubiquitous today, UART is rather easy to use and implement. Virtual COM ports allow USB to appear and function like a traditional hardware COM port. The way we usually use COM ports is with USB-to-serial adapter chips. These adapter chips convert USB to serial and vice versa.

Using a USB to serial IC, virtual COM ports can be configured through the OS to match all the features of regular COM ports. So we can do things like choose a port number, adjust the baud rates, and configure all the parameters you are accustomed to like in physical COM ports. Some common USB-to-serial chips include the CP2102, CP2104, and PL2303HX devices. These are available from a wide variety of retailers and can be used for all the examples that require UART in this book.

Here is the program you can use to use the UART module. To use the EUSART module, we need three files: a header file, a source file, and the main file. We will begin by looking at the header file in Listing 8-1.

```
/*
 * File: EUSART.h
 * Author: Armstrong Subero
 * PIC: 16F1719 w/Int OSC @ 32MHz, 5v
 * Program: Header file to setup PIC16F1717
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 * Program Version 1.1
 *
 * Program Description: This header sets up the EUSART module
 * Created on November 7th, 2016, 7:00 PM
 * Updated on November 24th 2023, 1:02 AM
 */

/*************************************************************
 * Function Prototype
 *************************************************************/

char EUSART_Initialize(const long int baudrate);
uint8_t EUSART_Read(void);
void EUSART_Write(uint8_t txData);
void EUSART_Write_Text(char *text);
void EUSART_Read_Text(char *Output, unsigned int length);
void EUSART_Write_Integer(int value_to_send);
```

Listing 8-1 EUSART Header File

The header file contains all the function prototypes we are using, and we do the actual implementation in the source file, which we will see in Listing 8-2.

```
/*
 * File: EUSART.c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/Int OSC @ 16MHz, 5v
 * Program: Library file containing functions for the EUSART
module
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 * Program Version: 1.2
 *                *Added additional comments
 *                *Simplified UART calculations
 *
 * Program Description: This Library allows you to use the
EUSART module of the PIC16F1719 in Asynchronous Mode
 *
 * Created on November 7th, 2016, 7:10 PM
 * Updated on November 24th 2023, 1:00 AM
 */

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

```
 *Includes and defines
 *********************************************************/

#include "PIC16F1719_Internal.h"
#include "EUSART.h"

/**********************************************************
 * Function: char EUSART_Initialize (const long int baudrate)
 *
 * Returns: Nothing
 *
 * Description: Initializes the EUSART module
 *
 * Usage: EUSART_Initialize()
 *********************************************************/

char EUSART_Initialize(const long int baudrate)
{
    unsigned int x;
    x = (_XTAL_FREQ - baudrate*64)/(baudrate*64);

    SPBRGL = x;
    BRGH = 0;
    BRG16 = 0;
    SYNC = 0;
    SPEN = 1;
    CREN = 1;
    TXEN = 1;

    return 0;
}

/**********************************************************
 * Function: char EUSART_Read (void)
 *
 * Returns: Nothing
 *
 * Description: Reads the EUSART module
 *
 * Usage: EUSART_Read()
 *********************************************************/

uint8_t EUSART_Read(void)
{

   RC1STAbits.SREN = 1;
    while(!PIR1bits.RCIF)
    {
    }

    if(1 == RC1STAbits.OERR)
    {
        // EUSART error - restart

        RC1STAbits.SPEN = 0;
        RC1STAbits.SPEN = 1;
    }

    return RC1REG;
}

/***********************************************************
 * Function: void EUSART_Read_Text(char *Output, unsigned int
length)
 *
 * Returns: Nothing
 *
 * Description: Writes to the EUSART module
 *
 * Usage:
 **********************************************************/
void EUSART_Read_Text(char *Output, unsigned int length)
{
    int i;
    for(int i=0;i<length;i++)
        Output[i] = EUSART_Read();
}

/*************************************************************
 * Function: char EUSART_Write (uint8_t txData)
 *
 * Returns: Nothing
 *
 * Description: Writes to the EUSART module
 *
 * Usage: EUSART_Write(x)
 ***********************************************************/

void EUSART_Write(uint8_t txData)
{
    while(0 == PIR1bits.TXIF)
    {
    }
    TX1REG = txData;    // Write the data byte to the USART.
}

/**********************************************************
 * Function: char EUSART_Write_Text (char *text)
 *
 * Returns: Nothing
 *
 * Description: Writes text the EUSART module
 *
 * Usage: EUSART_Write_Text("Some String")
 *********************************************************/

void EUSART_Write_Text(char *text)
{
  int i;
  for(i=0;text[i]!='\0';i++)
      EUSART_Write(text[i]);
}

/*********************************************************
 * Function: void itoa(char **buf, int d)
 *
 * Returns: Nothing
 *
 * Description: Converts integer to ASCII values
 *
 * Usage:     char buffer[20]; // Buffer to hold the ASCII
representation of the integer
 *            char *ptr = buffer; // Pointer to buffer
 *
 *            itoa(&ptr, value_to_send); // Convert integer to
ASCII
 ********************************************************/
void itoa(char **buf, int d)
{
    int div = 1;
    if (d < 0)
    {
        *((*buf)++) = '-';
        d = -d;
    }

    int temp = d;
    while (temp / 10)
    {
        temp /= 10;
        div *= 10;
    }

    while (div != 0)
    {
        int num = d / div;
        d %= div;
        div /= 10;
        *((*buf)++) = num + '0';
    }
    *(*buf) = '\0'; // Terminate the string
}

/***********************************************************
 * Function: char EUSART_Write_Integer (int value_to_send)
 *
 * Returns: Nothing
 *
 * Description: Writes text the EUSART module
 *
 * Usage: EUSART_Write_Text(integer)
 **********************************************************/
void EUSART_Write_Integer(int value_to_send)
{
    char buffer[20]; // Buffer to hold the ASCII
representation of the integer
    char *ptr = buffer; // Pointer to buffer

    itoa(&ptr, value_to_send); // Convert integer to ASCII

    // Send each character (digit) of the ASCII representation
over EUSART
    char *temp_ptr = buffer; // Temporary pointer to iterate
through the buffer
    while (*temp_ptr != '\0') {
        EUSART_Write(*temp_ptr++);
    }
}
```

Listing 8-2 EUSART Source File

Within the main source file, the EUSART_Initialize function initializes the module with a specified baud rate, configuring registers for communication settings like baud rate, transmitter, receiver, and other parameters. The EUSART_Read function retrieves a byte from the EUSART module, handling potential overrun errors. EUSART_Read_Text reads a specified number of bytes from the module and stores them in a character array. The EUSART_Write and EUSART_Write_Text functions send a byte or a string of characters, respectively, over the EUSART. Additionally, the library includes an itoa function for converting integers to ASCII and an EUSART_Write_Integer function to send integer values. We can use this in a main source file, which we see in Listing 8-3.

```
/*
 * File: main.c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/int OSC @ 32MHz, 5v
 * Program: 18_UART
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 * Program Version: 1.2
 *                   * Updated from PIC16F1717 to PIC16F1719
 *                   * Updated clock speed to 32 MHz
 *                   * Added PLL stabilization
 *
 * Program Description: This program allows the PIC16F1719 to
communicate via the EUSART module
 *
 * Hardware Description: PIN RB2 of a the PIC16F1719 MCU is
connected to a PL2303XX USB to UART converter cable
 *
 * Created November 7th, 2016, 7:05 PM
 * Last Updated: November 24th, 2023, 12:54 AM
 */

/*******************************************************
 *Includes and defines
 *******************************************************/

#include "PIC16F1719_Internal.h"
#include "EUSART.h"

/*******************************************************
 * Function: void initMain()
 *
 * Returns: Nothing
 *
 * Description: Contains initializations for the main
 *
 * Usage: initMain()
 ******************************************************/

void initMain(){

    // Run at 32 MHz
    internal_32();
    // Allow PLL startup time ~2 ms
    __delay_ms(10);

    // Set PIN D1 as output
    TRISDbits.TRISD1 = 0;

    // Turn off LED
    LATDbits.LATD1 = 0;

    // Setup PORTD
    TRISD = 0;
    ANSELD = 0;

    TRISBbits.TRISB2 = 0;
    ANSELBbits.ANSB2 = 0;

    /////////////////////
    // Setup EUSART
    ////////////////////
    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    PPSLOCKbits.PPSLOCKED = 0x00; // unlock PPS

    RB2PPSbits.RB2PPS = 0x14;   //RB2->EUSART:TX;
    RXPPSbits.RXPPS = 0x0B;   //RB3->EUSART:RX;

    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    PPSLOCKbits.PPSLOCKED = 0x01; // lock PPS
}

/************************************************************
 * Function: Main
 *
 * Returns: Nothing
 *
 * Description: Program entry point
 ***********************************************************/

void main(void) {
    initMain();

    // Initialize EUSART module with 19200 baud
    EUSART_Initialize(19200);

    while(1)
    {
        // Send text
        EUSART_Write_Text("Hello World\n");

        __delay_ms(1000);
    }

    return;
}
```

Listing 8-3 Main File

The schematic is rather straightforward as we see in Figure 8-1.

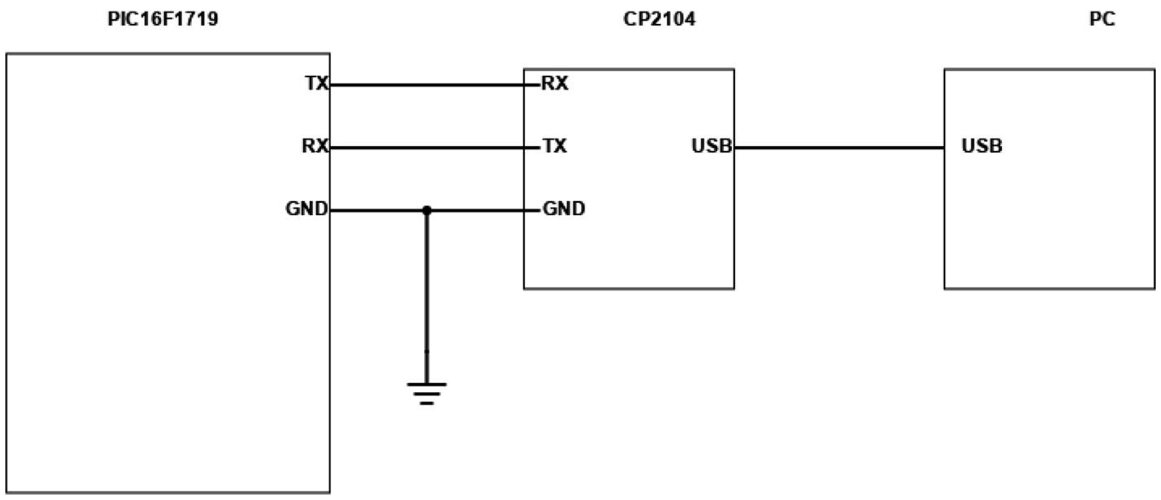


Figure 8-1 PIC16F1719 USART connection

The most important thing to remember when connecting a USART peripheral is that the TX must be connected to the RX line and the RX line must be connected to the TX line. They must also share a common ground connection.

Using GPS (Global Positioning System)

Now that we have an understanding of UART, we can apply it to interfacing to different modules. The first one we will look at is the use of a GPS receiver. GPS receivers use satellites to determine the position of the GPS receiver that receives a signal from these satellites. It would not be possible to cover GPS in its entirety in this book as the scope is rather large. We also don't need to understand it in detail as a combination of hardware and software features will allow us to interface very simply with it.

The way that GPS works is that the satellites we discuss form a constellation of at least 24 satellites in medium Earth orbit. These satellites emit a continuous signal containing information about their position and the precise time that the signals were transmitted. A GPS receiver equipped with channels, a minimum of four, will intercept these satellite signals.

A process called trilateration takes place, and the GPS receiver calculates the distance from each satellite based on the time it took for the signals to reach it. These distances are then combined, and the receiver will determine the exact three-dimensional position on Earth. The accuracy of the GPS relies on factors such as the number of satellites that are visible to the receiver and the signal quality. Getting into the nitty-gritty of how this operates will involve getting into orbital dynamics and signal processing, which is beyond the scope of this book. The GPS module that we will use is the U-BLOX NEO-6M GPS module, though since we will be using the NMEA (National Marine Electronics Association) protocol, once you have a GPS module that procures NMEA signals, you should be able to easily follow along with this section.

NMEA Commands

As I mentioned in the previous section, most GPS receivers allow us to communicate with it by the use of NMEA commands. NMEA stands for National Marine Electronics Association and refers to the body that produced the specification for the protocol that allows the GPS receiver to give the time, position, and velocity data that the user can parse. The GPS receiver receives commands and outputs data in a particular format that we can interpret.

The first two letters are always GP followed by three other letters, which tell us what the sentence is about. For example, the sentence we are interested in is the GLL, which stands for Geographic Latitude and Longitude. The GLL-type sentence has the following format:

```
$GPGLL,1123.01,N,1234.00,W,000111,A,*65
```

The $GPGLL tells us the type of NMEA sentence it is. The fragment 1123.01, N tells us a position of latitude that is 11 degrees and 23.01 minutes North. Similarly, 1234.00, W indicates a position of longitude that is 12 degrees and 34.00 minutes West.

To extract the information about the position of the receiver, we need to create a buffer to store the information as it comes in. Then we need to eliminate any invalid data and use the C language strstr function, which is provided by XC8, to determine if the string we are looking for is present in the buffer. Let's look at the program for interfacing a GPS module as we will see in Listing 8-4.

```
/*
 * File: Main. c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/Int OSC @ 16MHz, 5v
 * Program: 47_GPS
 * Compiler: XC8 (v1.38, MPLAX X v3.40)
 * Program Version: 1.0
 *
 *
 * Program Description: This Program Allows PIC16F1719 to
communicate via the EUSART module to a NEO-6M GPS module and
display Latitude and Longitude coordinates on an LCD.
 *
 * Hardware Description: A NEO-6M GPS module is connected to
the PIC16F1719 as follows:
 *
 *                        PPS -> NC
 *                        RXD -> TX
 *                        TXD -> RX
 *                        GND -> GND
 *                        VCC -> VCC
 *
 *
 * Created April 18th, 2017, 12:51 PM
 * Updated March 25th, 2024,  1:30 PM
 */

/**************************************************************
 *Includes and defines
 *************************************************************/

#include "PIC16F1719_Internal.h"
#include "EUSART.h"
#include "LCD.h"
#include <string.h>

// Variables
volatile char c;
volatile char d;

char* data;

static char uartBuffer[300];
int i;

char* terminator;
char conversionString[8];

double lat = 0.0;
double lon = 0.0;

double *longitude = &lon;
double *latitude  = &lat;

// Function prototype
void read_gps();

/************************************************************
 * Function: void initMain()
 *
 * Returns: Nothing
 *
 * Description: Contains initializations for main
 *
 * Usage: initMain()
 **********************************************************/

void initMain(){
    // Run at 16 MHz
    internal_16();

    // Setup PINS
    TRISBbits.TRISB3 = 1;
    ANSELBbits.ANSB3 = 0;

    TRISBbits.TRISB2 = 0;
    ANSELBbits.ANSB2 = 0;

    TRISD = 0;
    ANSELD = 0;
    PORTD = 0;

    /////////////////////
    // Setup EUSART
    ////////////////////
    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    PPSLOCKbits.PPSLOCKED = 0x00; // unlock PPS

    RB2PPSbits.RB2PPS = 0x14;   //RB2->EUSART:TX;
    RXPPSbits.RXPPS = 0x0B;   //RB3->EUSART:RX;

    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    PPSLOCKbits.PPSLOCKED = 0x01; // lock PPS

    INTCONbits.GIE = 1;
    INTCONbits.PEIE = 1;

    // set up UART 1 receive interrupt
    PIE1bits.RCIE = 1;
}

/**********************************************************
 * Function: Main
 *
 * Returns: Nothing
 *
 * Description: Program entry point
 *********************************************************/

void main(void) {
    initMain();
    Lcd_Init();
    Lcd_Clear();

    // Initialize EUSART module with 9600 baud
    EUSART_Initialize(9600);

    // give the module time to stabilize
    __delay_ms(100);

    while(1){

         Lcd_Set_Cursor(1,1);

         read_gps();

         // Write Latitude
         Lcd_Write_Float(*latitude);

         Lcd_Set_Cursor(2,1);

         // Write Longitude
         Lcd_Write_Float(*longitude);

         __delay_ms(2000);
         Lcd_Clear();
    }

    return;

}

/*********************************************************
 * Function: void read_gps()
 *
 * Returns: Pointers to lat and lon
 *
 * Description: Function to read the GPS module
 *
 * Usage: read_gps()
 *********************************************************/
```

```
void read_gps(){

    // Read characters from UART into buffer
    for(i=0; i<sizeof(uartBuffer)-1; i++)
    {
        d = EUSART_Read();
        uartBuffer[i] = d;
    }

    // Last character is null terminator
    uartBuffer[sizeof(uartBuffer)-1] = '\0';

    // Look for needle in haystack to find string for GPGLL
    data = strstr(uartBuffer, "$GPGLL");

    // if null exit
    if(data == NULL)
    {
        return;
    }

    // Find terminator
    terminator = strstr(data,",");

    // if null exit
    if(terminator == NULL)
    {
        return;
    }

    // If the first byte of the latitude field is ',', there
is no info
    // so exit

    if(data[7] == ',')
    {
        return;
    }

    ///////////////////////////////////
    // Search buffer data for Latitude
    // and Longitude values
    //////////////////////////////////

    data = terminator+1;

    terminator = strstr(data,",");
```

```
if(terminator == NULL)
{
    return;
}

memset(conversionString,0,sizeof(conversionString));
memcpy(conversionString, data, 2);
*latitude = atof(conversionString);

data += 2;
*terminator = '\0';
*latitude += (atof(data)/60);

data = terminator+1;
terminator = strstr(data,",");
if(terminator == NULL)
{
    return;
}
if(*data == 'S')
{
    *latitude *= -1;
}

data = terminator+1;
terminator = strstr(data,",");
if(terminator == NULL)
{
    return;
}
memset(conversionString,0,sizeof(conversionString));
memcpy(conversionString, data, 3);
*longitude = atof(conversionString);

data += 3;
*terminator = '\0';
*longitude += (atof(data)/60);

data = terminator+1;
terminator = strstr(data,",");
if(terminator == NULL)
{
    return;
}
if(*data == 'W')
{
        *longitude *= -1;
    }

}
```

Listing 8-4 Interfacing with a GPS Module

The initialization section configures the microcontroller’s pins, sets up communication via the EUSART module, and establishes the necessary PPS mappings. The main function continuously reads GPS data, parses it using the read_gps function, and displays the Latitude and Longitude on the LCD. The read_gps function reads characters from the EUSART module into a buffer, searches for the “$GPGLL” string indicating the start of a GPS data segment, and processes the data to extract Latitude and Longitude values. These values are then converted and displayed on the LCD in a continuous loop, providing real-time geographical coordinates.

Software USART

The PIC16F1719 microcontroller has one UART module. To add additional UART capability to the microcontroller, we can add a software UART. The integration of a software USART, or bit-banged USART, allows for additional serial communication channels to be created through software by manually controlling the timing and state of the GPIO pins used for transmission and reception. This approach can be particularly useful in scenarios where hardware resources are limited but the application demands multiple serial communication interfaces. However, implementing the software USART requires careful consideration of timing, processor load, and the specific requirements of the communication protocol to ensure reliable data transmission and reception.

Software USART implementations are inherently more CPU-intensive than their hardware counterparts. Every bit transmitted or received through the software USART requires the CPU to precisely manage the timing of the pin states, leaving less processing power for other tasks. This is a critical consideration in real-time applications or those with stringent timing requirements. Moreover, as the baud rate increases, the margin for timing error decreases, making higher-speed communications more susceptible to errors. For applications that demand higher reliability and speed, developers might explore other options, such as using SPI/I2C to USART bridge chips, which can provide additional hardware USART interfaces without the need for a complete redesign or a more capable microcontroller.

When deciding on the best approach to expand USART capabilities, it’s essential to weigh the trade-offs between cost, complexity, and performance. Software USART might be the most cost-effective and straightforward solution for low-speed, noncritical data communication tasks. However, for applications that require high-speed communications and greater reliability, or have stringent real-time constraints, investing in hardware with additional USART modules or utilizing bridge chips could be a more viable approach. This decision must consider the project’s specific needs, existing system architecture, and future scalability to ensure a balance between resource utilization, system complexity, and overall performance.

GSM Module

We will look at an application of the bit-banged USART using a GSM module. We will use the SIM800L module as it is widely available and is simple to use. The SIM800L module is a versatile and compact GSM/GPRS module that enables mobile communication in a wide range of applications. It operates on frequencies 850/900/1800/1900 MHz and can be used in global GSM networks. With its small size and low cost, it's particularly suitable for mobile devices, IoT (Internet of Things) projects, and DIY (do-it-yourself) applications where space and budget are limited. The SIM800L allows for SMS, call, and data transmission, making it a comprehensive solution for wireless communication needs. It supports GPRS multi-slot class 12/10 and mobile station class B, alongside a range of protocols for seamless integration into various projects.

In addition to its communication capabilities, the SIM800L module includes several features that make it highly adaptable for complex projects. It has onboard support for TCP/IP protocols, enabling Internet connectivity for embedded applications. This feature is particularly useful for IoT devices that need to send data to the cloud or interact with web services. Furthermore, the module provides interfaces for serial communication, which allows it to easily connect with microcontrollers and other electronic components in a system. Its low power consumption makes it ideal for battery-operated applications, and it can be powered up through a range of 3.4 V to 4.4 V, which offers flexibility in terms of power management for portable devices. The SIM800L module, with its blend of features, offers a powerful tool for developers looking to add GSM/GPRS functionality to their projects. The SIM800L module is given in Figure 8-2.

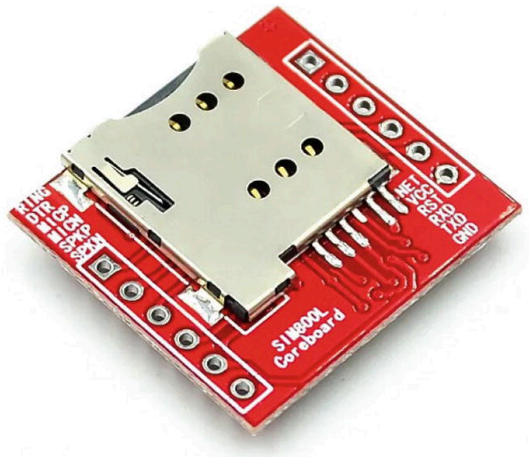


Figure 8-2 The SIM800L module

AT Commands

In a previous section, we looked at data structured in NMEA format, which is the type of data commonly given by GSM modules. In this section, we look at using AT (attention) commands. AT commands are commonly used to control modems. AT commands follow a specific syntax where each command begins with "AT" to get the attention of the device, followed by a series of characters that define the specific action to be performed. For example, "AT+CMGS" is used to send an SMS message. The simplicity and universality of AT commands across different devices make them an invaluable tool in the development of IoT applications, embedded systems, and telecommunications engineering; additionally, it’s very simple to use with PIC microcontrollers.

The use of AT commands extends beyond simple device control to include complex configurations for networking and IoT applications. Developers can use these commands to connect devices to the Internet via cellular networks, configure parameters for data transmission, and manage connections to ensure efficient communication between devices. The versatility of AT commands allows for their application in a variety of scenarios, ranging from sending data packets over a network to configuring IoT devices to operate in low-power modes. This broad applicability means that understanding and mastering AT commands is crucial for anyone working with cellular modules.

Despite their widespread use and utility, working with AT commands comes with its challenges. The correct sequence and syntax must be followed in sequence to avoid misconfigurations, and the response of the device must be correctly interpreted to ensure the intended operation is performed successfully. Moreover, the specific set of AT commands and their implementation can vary slightly between manufacturers and devices, so it’s best to consult the datasheet for the device you are using.

Here are some commonly used AT commands for controlling the SIM800L module:

- AT – Check to see if the module is working correctly. If it is, it will return OK.
- AT+CREG? – Get information about network registration. If the modules return +CREG: 0,1, then everything is fine.
- AT+IPR=9600 – Change the baud rate. For example, changing the 9600 to 2400 in the example would set the baud rate to 2400.

The other AT commands are used for things like sending and receiving messages, calls, and accessing the GPS features, among others. In this example, we will focus solely on sending text messages, so the commands for sending texts are shown in the code in Listing 8-5.

```
/*
 * File: Main.c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/Int OSC @ 16MHz, 5v
 * Program: 46_UBLOX_GSM
 * Compiler: XC8 (v1.38, MPLAX X v3.40)
 * Program Version: 1.0
 *
 *
 * Program Description: This Program Allows PIC16F1719 to
communicate with a SIM800L GSM module and send an SMS message
using a software UART.
 *
 *
 * Hardware Description: A SIM800L GSM module is connected as
follows:
 *
 *                        5v      -> 5V
 *                        GND     -> GND
 *                        VDD     -> NC
 *                        SIM_TXD -> RX
 *                        SIM_RXD -> TX
 *                        GND     -> GND
 *
 *
 * Created April 18th, 2017, 1:11 PM
 * Updated March 25th, 2024, 11:26 AM
 */

/*******************************************************************
 *Includes and defines
 ******************************************************************/
#include "PIC16F1719_Internal.h"

// Setup for soft UART
#define Baudrate 2400 //bps
#define OneBitDelay (1000000/Baudrate)
#define DataBitCount 8 // no parity, no flow control
#define UART_RX LATEbits.LATE0// UART RX pin
#define UART_TX LATEbits.LATE1 // UART TX pin
#define UART_RX_DIR TRISE0// UART RX pin direction register
#define UART_TX_DIR TRISE1 // UART TX pin direction register

//Function Declarations
void InitSoftUART(void);
unsigned char UART_Receive(void);
void UART_Transmit(const char);
void SUART_Write_Text(char *text);
void SUART_Write_Char(char a);
void SUART_Read_Text(char *Output, unsigned int length);

/*******************************************************************
 * Function: void initMain()
 *
 * Returns: Nothing
 *
 * Description: Contains initializations for main
 *
 * Usage: initMain()
 ******************************************************************/
```

```
void initMain(){
    // Run at 16 MHz
    internal_16();

    // Setup pins
    TRISD = 0x00;
    ANSELD = 0x00;
    PORTD = 0x00;

    TRISE = 0;
    ANSELE = 0;
}

/*************************************************************
 * Function: Main
 *
 * Returns: Nothing
 *
 * Description: Program entry point
 ************************************************************/

void main(void) {
    initMain();

    __delay_ms(1000);

     InitSoftUART(); // Initialize Soft UART
                    __delay_ms(1000);

    while(1){

        // Send commands to module
        SUART_Write_Text("AT+CMGF=1\r\n");
        __delay_ms(1000);

        SUART_Write_Text("AT+CMGS=\"1113951\"\r\n");
        __delay_ms(1000);

        // Message to send
        SUART_Write_Text("Test");
        __delay_ms(1000);

        UART_Transmit((char)26);
        __delay_ms(1000);
}

    return;
```

```
}

/*************************************************************
 * Function: void InitSoftUART(void)
 *
 * Returns: Nothing
 *
 * Description: Initialize the software UART module
 ***********************************************************/
void InitSoftUART(void)
{
 UART_TX = 1; // TX pin is high in idle state

 UART_RX_DIR = 1; // Input
 UART_TX_DIR = 0; // Output
}

/*************************************************************
 * Function: unsigned char UART_Receive(void)
 *
 * Returns: Status
 *
 * Description: Recieve a character via software USART
 ***********************************************************/
unsigned char UART_Receive(void)
{
 unsigned char DataValue = 0;

 //wait for start bit
 while(UART_RX==1);

 __delay_us(OneBitDelay);
 __delay_us(OneBitDelay/2); // Take sample value in the mid of
bit duration

 for ( unsigned char i = 0; i < DataBitCount; i++ )
 {
  if ( UART_RX == 1 ) //if received bit is high
  {
   DataValue += (1<<i);
  }

  __delay_us(OneBitDelay);
 }

 // Check for stop bit
 if ( UART_RX == 1 ) //Stop bit should be high
 {
  __delay_us(OneBitDelay/2);
  return DataValue;
 }
 else //some error occurred !
 {
  __delay_us(OneBitDelay/2);
  return 0x000;
 }
}

/*********************************************************************
 * Function: void UART_Transmit(const char DataValue)
 *
 * Returns: Status
 *
 * Description: Transmit a character via software USART
 *********************************************************************/
void UART_Transmit(const char DataValue)
{
 /* Basic Logic
    TX pin is usually high. A high to low bit is the starting
bit and a low to high bit is the ending bit. No parity bit. No
flow control. BitCount is the number of bits to transmit. Data
is transmitted LSB first.
 */

 // Send Start Bit
 UART_TX = 0;
 __delay_us(OneBitDelay);

 for ( unsigned char i = 0; i < DataBitCount; i++ )
 {
  //Set Data pin according to the DataValue
  if(((DataValue>>i)&0x1) == 0x1 ) //if Bit is high
  {
   UART_TX = 1;
  }
  else //if Bit is low
  {
   UART_TX = 0;
  }

     __delay_us(OneBitDelay);
 }
 //Send Stop Bit
 UART_TX = 1;
 __delay_us(OneBitDelay);
}

/**************************************************************
 * Function: void SUART_Write_Text(char *text)
 *
 * Returns: Nothing
 *
 * Description: Writes Text via the Software UART module
 *************************************************************/
void SUART_Write_Text(char *text)
{
  int i;

  for(i=0;text[i]!='\0';i++)
      UART_Transmit(text[i]);
}

/*************************************************************
 * Function: void SUART_Read_Text(char *Output, unsigned int
length)
 *
 * Returns: Nothing
 *
 * Description: Reads Text via the Software UART module
 ************************************************************/
void SUART_Read_Text(char *Output, unsigned int length)
{
    int i;
    for(int i=0;i<length;i++)
        Output[i] = UART_Receive();
}

/*************************************************************
 * Function: void SUART_Write_Char(char a)
 *
 * Returns: Nothing
 *
 * Description: Writes Text via the Software UART module
 ***********************************************************/
void SUART_Write_Char(char a)
{
    UART_Transmit(a - 0x128);
}
```

Listing 8-5 Using the GSM Module

We use the soft UART to send AT commands to the SIM800L module. These commands are used to configure the module for sending an SMS message. The sequence of AT commands includes setting the SMS text mode (AT+CMGF=1), specifying the recipient's phone number (AT+CMGS=\“number\”), and the message content, followed by sending a Ctrl+Z character (ASCII 26) to indicate the end of the message input. Delays are inserted between commands to ensure proper execution and response time from the GSM module.

Using SPI (Serial Peripheral Interface)

Serial Peripheral Interface (SPI) is another type of serial communication protocol commonly used in embedded systems and present on the PIC microcontroller. SPI is a very important protocol and is widely implemented on a variety of sensors. Unlike USART, where very few applications require a clock, in SPI, a clock is present in all applications because SPI uses synchronous data transfer.

The device in the SPI communication that generates the clock is known as the master, and the other is known as the slave. SPI always only has one master device, although there can be many slaves. SPI has several lines: Serial Clock (SCK), Master Out Slave In (MOSI), Master In Slave Out (MISO), and Slave Select (SS). If there is only one slave device connected to the SPI bus, then this line may be left low as SPI is active low.

One of the major disadvantages of SPI is that it uses a lot of I/O lines. Although SCK, MOSI, and MISO remain the same regardless of the number of slave devices on the bus, an additional SS line must be used for each slave device that is connected to the bus. The advantage of SPI is that it can transfer millions of bytes of data per second and is useful when interacting with devices such as SD cards, which would require data transfer at very high speeds.

In learning to use the SPI peripheral available on the PIC microcontroller, we will use an MCP4131 digital potentiometer with the PIC microcontroller. To use the MCP4131 is connected to the PIC16F1719 as shown in Figure 8-3.

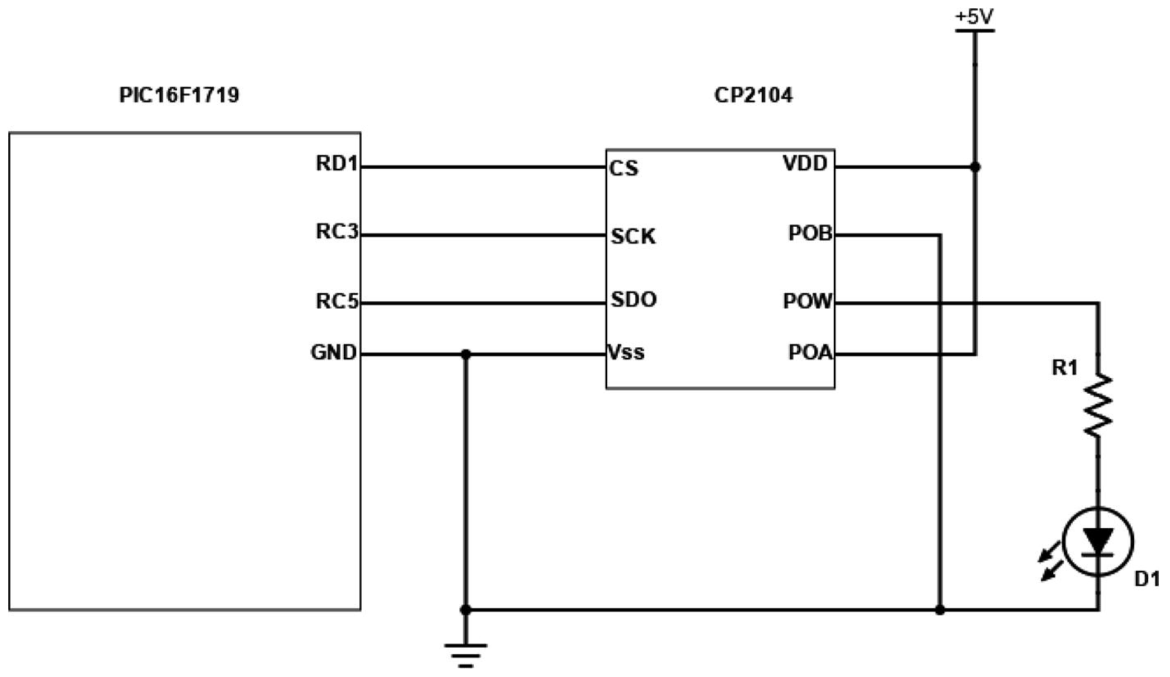


Figure 8-3 MCP4131 SPI connection

The code for the SPI was generated using the Microchip Code Configurator. There were some modifications done to the generated SPI code. The code for the header file is given in Listing 8-6.

```
/*
 * File: SPI.h
 * Author: Armstrong Subero
 * PIC: 16F1719 w/X OSC @ 32MHz, 5v
 * Program: Header file for SPI module
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 *
 * Program Version 1.1
 *                 *Updated Comments
 *
 * Program Description: This program header provides function
prototypes for SPI module on PIC16F1719
 *
 *
 *
 * Created on November 7th,  2016, 5:45 PM
 * Updated on November 24th, 2023, 4:13 AM
 */

/**************************************************************
 *Includes and defines
 *************************************************************/
```

```
#include "PIC16F1719_Internal.h"
#define DUMMY_DATA 0x0

void SPI_Initialize(void);
uint8_t SPI_Exchange8bit(uint8_t data);
uint8_t SPI_Exchange8bitBuffer(uint8_t *dataIn, uint8_t
bufLen, uint8_t *dataOut);
bool SPI_IsBufferFull(void);
bool SPI_HasWriteCollisionOccured(void);
void SPI_ClearWriteCollisionStatus(void);
```

Listing 8-6 The SPI Header File

In our header file, we have a macro that defines a constant, “DUMMY_DATA”, set to 0x0, which can be used as a placeholder or default value. We also have function prototypes for SPI initialization and communication including functions for initializing the module and transferring data. We also have the source file for the SPI module given in Listing 8-7.

```
/**
  Section: Included Files
*/

#include <xc.h>
#include "SPI.h"

/**
  Section: Macro Declarations
*/

#define SPI_RX_IN_PROGRESS 0x0

/**
  Section: Module APIs
*/

/**
  Section: Macro Declarations
*/

#define DUMMY_DATA 0x0

/**
  Section: SPI Module APIs
*/

/**
  @Summary
```

```
    Initializes the SPI

  @Description
    This routine initializes the SPI.
    This routine must be called before any other MSSP routine
is called.
    This routine should only be called once during system
initialization.

  @Preconditions
    None

  @Param
    None

  @Returns
    None

  @Comment

  @Example
    <code>
    uint8_t     myWriteBuffer[MY_BUFFER_SIZE];
    uint8_t     myReadBuffer[MY_BUFFER_SIZE];
    uint8_t     writeData;
    uint8_t     readData;
    uint8_t     total;

    SPI_Initialize();

    total = 0;
    do
    {
        total = SPI_Exchange8bitBuffer(&myWriteBuffer[total],
MY_BUFFER_SIZE - total, &myWriteBuffer[total]);

        // Do something else...

    } while(total < MY_BUFFER_SIZE);

    readData = SPI_Exchange8bit(writeData);
    </code>
 */

void SPI_Initialize(void)
{
   // Set the SPI module to the options selected in the User
Interface

    // BF RCinprocess_TXcomplete; UA dontupdate; SMP Sample At
Middle; P stopbit_notdetected; S startbit_notdetected; R_nW
write_noTX; CKE Active to Idle; D_nA lastbyte_address;
    SSP1STAT = 0x40;

    // SSPEN enabled; WCOL no_collision; SSPOV no_overflow;
CKP Idle:Low, Active:High; SSPM FOSC/64;
    SSP1CON1 = 0x22;

    // SSP1ADD 0;
    SSP1ADD = 0x00;
}

/**
  @Summary
    Exchanges a data byte over SPI

  @Description
    This routine exchanges a data byte over SPI bus.
    This is a blocking routine.

  @Preconditions
    The SPI_Initialize() routine should be called
    prior to use this routine.

  @Param
    data - data byte to be transmitted over SPI bus

  @Returns
    The received byte over SPI bus

  @Example
    <code>
    uint8_t     writeData;
    uint8_t     readData;
    uint8_t     readDummy;

    SPI_Initialize();

    // for transmission over SPI bus
    readDummy = SPI_Exchange8bit(writeData);

    // for reception over SPI bus
    readData = SPI_Exchange8bit(DUMMY_DATA);
    </code>
 */

uint8_t SPI_Exchange8bit(uint8_t data)
{
    // Clear the Write Collision flag, to allow writing
    SSP1CON1bits.WCOL = 0;

    SSPBUF = data;

    while(SSP1STATbits.BF == SPI_RX_IN_PROGRESS)
    {
    }

    return (SSPBUF);
}

/**
  @Summary
    Exchanges buffer of data over SPI

  @Description
    This routine exchanges buffer of data (of size one byte)
over SPI bus.
    This is a blocking routine.

  @Preconditions
    The SPI_Initialize() routine should be called
    prior to use this routine.

  @Param
    dataIn  - Buffer of data to be transmitted over SPI.
    bufLen  - Number of bytes to be exchanged.
    dataOut - Buffer of data to be received over SPI.

  @Returns
    Number of bytes exchanged over SPI.

  @Example
    <code>
    uint8_t     myWriteBuffer[MY_BUFFER_SIZE];
    uint8_t     myReadBuffer[MY_BUFFER_SIZE];
    uint8_t     total;

    SPI_Initialize();

    total = 0;
    do
    {
        total = SPI_Exchange8bitBuffer(&myWriteBuffer[total],
MY_BUFFER_SIZE - total, &myWriteBuffer[total]);

        // Do something else...

    } while(total < MY_BUFFER_SIZE);
    </code>
 */

uint8_t SPI_Exchange8bitBuffer(uint8_t *dataIn, uint8_t
bufLen, uint8_t *dataOut)
{
    uint8_t bytesWritten = 0;

    if(bufLen != 0)
    {
        if(dataIn != NULL)
        {
            while(bytesWritten < bufLen)
            {
                if(dataOut == NULL)
                {
                    SPI_Exchange8bit(dataIn[bytesWritten]);
                }
                else
                {
                    dataOut[bytesWritten] =
SPI_Exchange8bit(dataIn[bytesWritten]);
                }

                bytesWritten++;
            }
        }
        else
        {
            if(dataOut != NULL)
            {
                while(bytesWritten < bufLen )
                {
                    dataOut[bytesWritten] =
SPI_Exchange8bit(DUMMY_DATA);

                    bytesWritten++;
                }
            }
        }
    }

    return bytesWritten;
}

bool SPI_IsBufferFull(void)
{
    return (SSP1STATbits.BF);
}

/**
  @Summary
    Clears the status of write collision.

  @Description
    This routine clears the status of write collision.

  @Preconditions
    The SPI_Initialize() routine must have been called prior
to use this routine.

  @Param
    None

  @Returns
    None

  @Example
    if(SPI_HasWriteCollisionOccured())
    {
        SPI_ClearWriteCollisionStatus();
    }
*/

bool SPI_HasWriteCollisionOccured(void)
{
    return (SSP1CON1bits.WCOL);
}

void SPI_ClearWriteCollisionStatus(void)
{
    SSP1CON1bits.WCOL = 0;
}
/**
 End of File
```

```
*/
```

Listing 8-7 SPI Source File

In our source file, the SPI_Initialize function configures the SPI module's operational parameters, such as clock polarity, phase, and speed, according to predefined settings. It sets up the SPI control registers (SSP1STAT and SSP1CON1) and the SPI baud rate (SSP1ADD) for the communication. This function lays the groundwork for SPI communication by ensuring the SPI module is correctly configured with the desired operational characteristics before any data exchange occurs. This function must be called every time we use SPI on the PIC microcontroller.

Following initialization, the code provides two primary data exchange functions: SPI_Exchange8bit for sending and receiving single bytes of data and SPI_Exchange8bitBuffer for handling buffers of data. The SPI_Exchange8bit function transmits a single byte to the SPI bus and waits for a response, effectively enabling synchronous send-receive operations. This function checks for write collisions and waits for the receive buffer to be full, indicating that data has been received and is ready to be read. The SPI_Exchange8bitBuffer function extends this capability to arrays of data, allowing for efficient transmission and reception of multiple bytes. This function iterates through the input buffer, sending each byte via SPI_Exchange8bit and optionally storing received bytes in an output buffer, thereby facilitating bulk data transfers over SPI.

Additionally, our source file has utility functions, such as SPI_IsBufferFull, SPI_HasWriteCollisionOccured, and SPI_ClearWriteCollisionStatus, which offer mechanisms to check the status of the SPI buffer and manage write collisions, enhancing error handling and ensuring reliable SPI communication. Now that we have our source file for controlling the SPI module, we can look at the main program file in Listing 8-8.

```
/*
 * File: main.c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/int OSC @ 32MHz, 5v
 * Program: 20_SPI
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 * Program Version: 1.2
 *                   * Updated from PIC16F1717 to PIC16F1719
 *                   * Updated clock speed to 32 MHz
 *                   * Added PLL stabilization
 *
 * Program Description: This program allows the PIC16F1719 to
demonstrate use of the SPI bus
 *
 * Hardware Description: A MCP4131 Digipot is connected to the
SPI bus as follows:-
 *                        Vss --> Vss
 *                        Vdd --> Vdd
 *                         SS  --> RD1
 *                         SCK --> RC3
 *                         SDI --> RC5
 *                         POB --> GND
 *                         POW --> LED via 1k resistor
 *                         POA --> Vdd
 *
 * Created November 7th, 2016, 7:05 PM
 * Last Updated: November 24th, 2023, 3:36 AM
 */

/*************************************************************
 *Includes and defines
 ************************************************************/
#include "PIC16F1719_Internal.h"
#include "SPI.h"

void digiPot_write(int i);

/*************************************************************
 * Function: void initMain()
 *
 * Returns: Nothing
 *
 * Description: Contains initializations for main
 *
 * Usage: initMain()
 *************************************************************/

void initMain(){
    // Run at 32 MHz
    internal_32();

    // Allow PLL startup time ~2 ms
    __delay_ms(10);

    ////////////////////////
    // Setup SPI
    ///////////////////////

    // Set PIN D1 as SS
    TRISDbits.TRISD1 = 0;
    ANSELCbits.ANSC3 = 0;
    ANSELCbits.ANSC4 = 0;
    ANSELCbits.ANSC5 = 0;

    TRISCbits.TRISC3 = 0;
    TRISCbits.TRISC4 = 1;
    TRISCbits.TRISC5 = 0;

    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    PPSLOCKbits.PPSLOCKED = 0x00; // unlock PPS

    SSPDATPPSbits.SSPDATPPS = 0x14;   //RC4->MSSP:SDI;
    RC3PPSbits.RC3PPS = 0x10;   //RC3->MSSP:SCK;
    RC5PPSbits.RC5PPS = 0x11;   //RC5->MSSP:SDO;

    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    PPSLOCKbits.PPSLOCKED = 0x01; // lock PPS

    // Initialize SPI
    SPI_Initialize();

}

/**************************************************************
 * Function: Main
 *
 * Returns: Nothing
 *
 * Description: Program entry point
 **************************************************************/

void main(void) {
    initMain();

    // Digipot variable
    int i;

    while(1){

      // Seven bit resolution
      for (i = 0; i <= 128; i++){
          // Write Value
          digiPot_write(i);
          __delay_ms(100);
      }
    }

    return;

}
```

```
/**************************************************************
 * Function: digipot_write
 *
 * Returns: Nothing
 *
 * Description: Writes a particular value to a MCP4131 digital
potentiometer
 *
 * Usage: digiPot_write(x);
 *************************************************************/

void digiPot_write(int i){
    // Set SS Low
    LATDbits.LATD1 = 0;

    // Slave address
    SPI_Exchange8bit(0x00);

    // Data
    SPI_Exchange8bit(i);

    // Set SS High
    LATDbits.LATD1 = 1;
}
```

Listing 8-8 SPI Program Main File

This program is designed for the PIC16F1719 microcontroller and demonstrates the use of the Serial Peripheral Interface (SPI) bus. The specific application involves controlling an MCP4131 digital potentiometer through SPI communication. The main program itself is very simple. Inside the loop, the program iterates through values from 0 to 128 and writes each value to the MCP4131 digital potentiometer using the "digiPot_write" function. A delay of 100 milliseconds is introduced between each write.

We have a function "digiPot_write" to write values to the digital potentiometer. This function is responsible for writing a particular value to the MCP4131 digital potentiometer via SPI communication. It sets the Slave Select (SS) line low, exchanges the slave address (0x00), exchanges the data value (0 to 128), and then sets the SS line high.

Using the I2C (Inter-Integrated Circuit) Protocol

We now look at the final protocol presented in this chapter: the I2C (Inter-Integrated Circuit) protocol. There are a lot of tutorials that go in depth into the I2C protocol, and thus, I will not attempt to give a detailed explanation of it. There are some things that you must first know to use this protocol effectively.

The I2C protocol is widely used. When compared to SPI, I2C uses fewer (only two) lines, but communication occurs at a slower rate and it is a very complex protocol. The example we look at is reading and writing an EEPROM. This is an important application because the PIC16F1717 does not have an onboard EEPROM.

I2C is unique when compared to the other protocols we have discussed so far in that only one line is used for data flow. On the I2C bus, the device known as the master is used to communicate with another device, known as the slave. The master can communicate with the slave because each device on the slave has its address.

The lines used for communication with the slave are the serial clock line (SCL) and serial data line (SDA). For I2C to work, the lines must be connected to VCC using pull-up resistors. Some calculations can be used to determine the value of these resistors by working out the capacity of the lines. In practice, however, I have found that either 4.7k or 1k resistors do the job quite adequately.

These pull-up resistors are required because I2C devices pull the signal line low but cannot drive it high, and the pull-up resistors are there to restore the signal line to high.

The speed most I2C devices use to communicate is either 100 kHz or 400 kHz. The protocol transmits information via frames. There are two types of frames available: an address frame, which informs the bus which slave devices will receive the message, followed by data frames containing the actual 8-bit data. Every frame has a 9th bit, called the acknowledge (ACK) or not acknowledge (NACK) bit, which is used to indicate whether the slave device reads the transmission.

Every I2C communication from the master starts with the master pulling the SDA line low and leaving the SCL line high. This is known as the start condition. Similarly, there is a stop condition, where there is a low-to-high transition on SDA after a low-to-high transition on SCL with SCL being high. The header file for I2C is given in Listing 8-9.

```
/*
 * File: I2C.h
 * Author: Armstrong Subero
 * PIC: 16F1717 w/X OSC @ 16MHz, 5v
 * Program: Header file to setup PIC16F1717 I2C
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 * Program Version 1.3
 *                * Updated comments
 *                *Separated file into Header and C source
file
 *                *Used non-mcc code
 *
 *
 * Program Description: This program header will allows setup
of I2C
 *
 * Created on September 12th, 2016, 7:00 PM
 * Updated on September 24th, 2023, 3:42 AM
 */

/*************************************************************
 *Includes and defines
 ************************************************************/

#include "PIC16F1719_Internal.h"

void I2C_Init(void);
void Send_I2C_Data(unsigned int databyte);
unsigned int Read_I2C_Data(void);
void Send_I2C_ControlByte(unsigned int BlockAddress,unsigned
int RW_bit);
void Send_I2C_StartBit(void);
void Send_I2C_StopBit(void);
void Send_I2C_ACK(void);
void Send_I2C_NAK(void);
```

Listing 8-9 I2C Header File

The function prototypes in the header serve as a foundational toolkit for implementing I2C communication, allowing us to perform essential operations such as initializing the I2C module, sending and receiving data bytes, and managing control signals within the I2C communication process. We will implement these functions in the source file given in Listing 8-10.

```
/*
 * File: I2C.c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/Int OSC @ 32MHz, 5v
 * Program: Library file to configure PIC16F1719 I2C module
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 * Program Version: 1.1
 *                *Updated for 32 MHz operation
 *                *Added additional comments
 *
 * Program Description: This Library allows you to control
PIC16F1719 I2C in master mode
 *
 * Created on November 12th, 2016, 7:05 AM
 * Updated on November 25th, 2023, 3:39 AM
 */

/*************************************************************
 *Includes and defines
 ************************************************************/
#include "I2C.h"

void I2C_Init(void){

    //*****************************************
// Setup MSSP as I2C Master mode, 100 kHz
```

```
    SSPCONbits.SSPM=0x08;       // I2C Master mode, clock =
Fosc/(4 * (SSPADD+1))
    SSPCONbits.SSPEN=1;         // enable MSSP port

    //**************************************************
    // The SSPADD register value  is used to determine the
clock rate for I2C communication.
    //
    // Equation for I2C clock rate:   Fclock = Fosc/[(SSPADD
+1)*4]
    //
    // For this example we want the the standard 100Khz I2C
clock rate and our
    // internal Fosc is 32Mhz so we get:  100000 =
32000000/[(SSPADD+1)*4]
    // or solving for SSPADD = [(32000000/100000)-4]/4
    // and we get SSPADD = 79 (39 for 16 MHz)

    SSPADD = 79;                // set Baud rate clock divider
    //**************************************************

    __delay_ms(10); // let everything settle.
}

//**************************************************
// Send one byte to SEE
//**************************************************
void Send_I2C_Data(unsigned int databyte)
{
    PIR1bits.SSP1IF=0;          // clear SSP interrupt bit
    SSPBUF = databyte;              // send databyte
    while(!PIR1bits.SSP1IF);    // Wait for interrupt flag to
go high indicating transmission is complete
}

//*************************************************************
// Read one byte from SEE
//*************************************************************
unsigned int Read_I2C_Data(void)
{
    PIR1bits.SSP1IF=0;          // clear SSP interrupt bit
    SSPCON2bits.RCEN=1;         // set the receive enable bit
to initiate a read of 8 bits from the serial eeprom
    while(!PIR1bits.SSP1IF);    // Wait for interrupt flag to
go high indicating transmission is complete
```

```
    return (SSPBUF);            // Data from eeprom is now in
the SSPBUF so return that value
}

//**************************************************************
// Send control byte to SEE (this includes 4 bits of device
code, block select bits and the R/W bit)
//**************************************************************
// Notes:
// 1) The device code for serial eeproms is defined as '1010'
which we are using in this example
// 2) RW_bit can only be a one or zero
// 3) Block address is only used for SEE devices larger than
4K, however on
// some other devices these bits may become the hardware
address bits that allow you
// to put multiple devices of the same type on the same
bus.  Read the datasheet
// on your particular serial eeprom device to be sure.
//**************************************************************
void Send_I2C_ControlByte(unsigned int BlockAddress,unsigned
int RW_bit)
{
    PIR1bits.SSP1IF=0;          // clear SSP interrupt bit

    // Assemble the control byte from device code, block
address bits and R/W bit
    // so it looks like this: CCCCBBBR
    // where 'CCCC' is the device control code
    // 'BBB' is the block address
    // and 'R' is the Read/Write bit

    SSPBUF = (((0b1010 << 4) | (BlockAddress <<1)) +
RW_bit);  // send the control byte

    while(!PIR1bits.SSP1IF);    // Wait for interrupt flag to
go high indicating transmission is complete
}

//**************************************************************
// Send start bit to SEE
//**************************************************************
void Send_I2C_StartBit(void)
{
    PIR1bits.SSP1IF=0;          // clear SSP interrupt bit
    SSPCON2bits.SEN=1;          // send start bit
    while(!PIR1bits.SSP1IF);    // Wait for the SSPIF bit to
go back high before we load the data buffer
}

//************************************************************
// Send stop bit to SEE
//************************************************************
void Send_I2C_StopBit(void)
{
    PIR1bits.SSP1IF=0;          // clear SSP interrupt bit
    SSPCON2bits.PEN=1;          // send stop bit
    while(!PIR1bits.SSP1IF);    // Wait for interrupt flag to
go high indicating transmission is complete
}

//***********************************************************
// Send ACK bit to SEE
//***********************************************************
void Send_I2C_ACK(void)
{
   PIR1bits.SSP1IF=0;          // clear SSP interrupt bit
   SSPCON2bits.ACKDT=0;        // clear the Acknowledge Data
Bit - this means we are sending an Acknowledge or 'ACK'
   SSPCON2bits.ACKEN=1;        // set the ACK enable bit to
initiate transmission of the ACK bit to the serial eeprom
   while(!PIR1bits.SSP1IF);    // Wait for interrupt flag to
go high indicating transmission is complete
}

//************************************************************
// Send NAK bit to SEE
//************************************************************
void Send_I2C_NAK(void)
{
    PIR1bits.SSP1IF=0;           // clear SSP interrupt bit
    SSPCON2bits.ACKDT=1;        // set the Acknowledge Data
Bit- this means we are sending a No-Ack or 'NAK'
    SSPCON2bits.ACKEN=1;        // set the ACK enable bit to
initiate transmission of the ACK bit to the serial eeprom
    while(!PIR1bits.SSP1IF);    // Wait for interrupt flag to
go high indicating transmission is complete
}
```

Listing 8-10 I2C Source File

The I2C_Init function is designed to configure the I2C module embedded within the PIC16F1719, setting up parameters such as clock speed, bus characteristics, and any necessary hardware configurations to ensure the microcontroller can communicate over an I2C bus. Following initialization, the functions Send_I2C_Data and Read_I2C_Data enable the transmission and reception of data bytes to and from I2C devices, facilitating two-way communication. These operations are critical for interacting with a wide range of peripheral devices, such as sensors, EEPROMs, or other microcontrollers, allowing for data exchange and device control over the I2C bus.

Additionally, the code snippet includes functions for managing the I2C communication protocol's control signals. Send_I2C_ControlByte is used to transmit a control byte to specify the target device address and the operation (read or write) to be performed; Send_I2C_StartBit and Send_I2C_StopBit are responsible for initiating and terminating an I2C communication session, respectively; and Send_I2C_ACK and Send_I2C_NAK manage acknowledgments between the microcontroller and peripheral devices, facilitating error checking and flow control within the communication process. These control functions are essential for establishing a reliable and effective communication session between the PIC16F1719 and other devices on the I2C bus, enabling complex interactions such as sequential read/write operations and device addressing within a multi-device I2C environment. We can use these to read and write a serial EEPROM device. The schematic for this is given in Figure 8-4.

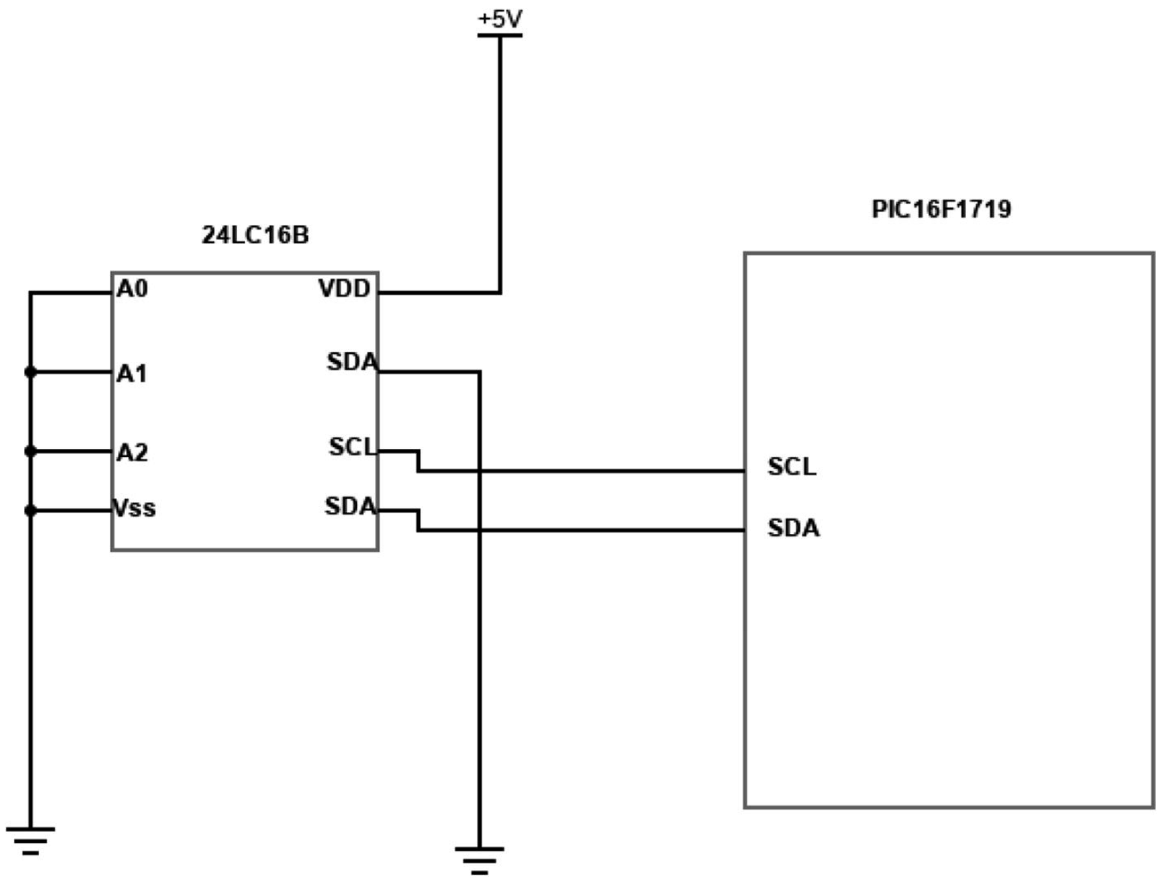


Figure 8-4 Connecting PIC16F1719 to EEPROM

In case you are not familiar with EEPROMs, they serve as nonvolatile storage solutions, meaning they can retain data even without power, which makes them ideal for storing settings, calibration data, small amounts of data logging, device configuration, and firmware updates. PC enthusiasts may be familiar with EEPROMs used for specific BIOS or firmware settings. They are also used in TV remotes, presets for radios in automotive applications, and things like operational parameters for electronic control units (ECUs). With the onset of IoT and security being at the forefront of embedded design, EEPROMs have use cases in storing unique identifiers and encryption keys, for example, so learning to use them is very important. The main program source code for interfacing the EEPROM to the PIC microcontroller is given in Listing 8-11.

```
/*
 * File: main.c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/int OSC @ 32MHz, 5v
 * Program: 19_I2C
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 * Program Version: 1.2
 *                  * Updated from PIC16F1717 to PIC16F1719
 *                  * Updated clock speed to 32 MHz
 *                  * Added PLL stabilization
 *
 * Program Description: This program allows the PIC16F1719 to
demonstrate use of the I2C bus
 *
 * Hardware Description: PIN RB2 of a the PIC16F1719 MCU is
connected to a PL2303XX USB to UART converter cable and a
24LC16B EEPROM is connected to the I2C bus
 *                       -- RC4 SDA
 *                       -- RC5 SCL
 *                       10k pull up resistors are used on the
I2C lines
 *
 * Created November 7th, 2016, 7:05 PM
 * Last Updated: November 24th, 2023, 3:36 AM
 */

/*************************************************************
 *Includes and defines
 ************************************************************/

#include "PIC16F1719_Internal.h"
#include "EUSART.h"
#include "I2C.h"

//////////////////////////////////////
// Function to Write EEPROM
```

```
////////////////////////////////////
void EEPROM_Write(unsigned char block_address, unsigned char
word_address, unsigned char eeprom_data) {
    EUSART_Write_Text("Writing EEPROM\n");

    Send_I2C_StartBit();
    Send_I2C_ControlByte(block_address, 0);
    Send_I2C_Data(word_address);
    Send_I2C_Data(eeprom_data);
    Send_I2C_StopBit();

    // Add a delay after write (adjust the delay time as per
EEPROM specifications)
    __delay_ms(200); // Increase the delay time to 20
milliseconds or more

    EUSART_Write_Text("Write completed\n");
}

//////////////////////////////////////////
// Function to Read EEPROM
/////////////////////////////////////////
unsigned char EEPROM_Read(unsigned char block_address,
unsigned char word_address) {
    EUSART_Write_Text("Reading EEPROM\n");

    Send_I2C_StartBit();
    Send_I2C_ControlByte(block_address, 0);
    Send_I2C_Data(word_address);

    Send_I2C_StartBit();
    Send_I2C_ControlByte(block_address, 1);
    unsigned char incoming_data = Read_I2C_Data();
    Send_I2C_NAK();
    Send_I2C_StopBit();

    __delay_ms(1000);

    return incoming_data;
}

 /************************************************************
 * Function: void initMain()
 *
 * Returns: Nothing
 *
 * Description: Contains initializations for main
 *
 * Usage: initMain()
 ***********************************************************/

void initMain(){

    // Run at 32 MHz
    internal_32();

    // Allow PLL startup time ~2 ms
    __delay_ms(10);

    // Set PIN D1 as output
    TRISDbits.TRISD1 = 0;

    // Turn off LED
    LATDbits.LATD1 = 0;

    // Setup PORTD
    TRISD = 0;
    ANSELD = 0;

    TRISBbits.TRISB2 = 0;
    ANSELBbits.ANSB2 = 0;

    // Setup pins for I2C
    ANSELCbits.ANSC4 = 0;
    ANSELCbits.ANSC5 = 0;

    TRISCbits.TRISC4 = 1;
    TRISCbits.TRISC5 = 1;

    /////////////////////
    // Setup EUSART
    ////////////////////
    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    PPSLOCKbits.PPSLOCKED = 0x00; // unlock PPS

    RB2PPSbits.RB2PPS = 0x14;   //RB2->EUSART:TX;
    RXPPSbits.RXPPS = 0x0B;   //RB3->EUSART:RX;

    RC4PPSbits.RC4PPS = 0x0011;   //RC4->MSSP:SDA;
    SSPDATPPSbits.SSPDATPPS = 0x0014;   //RC4->MSSP:SDA;
    SSPCLKPPSbits.SSPCLKPPS = 0x0015;   //RC5->MSSP:SCL;
    RC5PPSbits.RC5PPS = 0x0010;   //RC5->MSSP:SCL;
    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    PPSLOCKbits.PPSLOCKED = 0x01; // lock PPS
}

 /************************************************************
 * Function: Main
 *
 * Returns: Nothing
 *
 * Description: Program entry point
 ************************************************************/

void main(void) {
    initMain();

    // Initialize EUSART module with 19200 baud
    EUSART_Initialize(19200);

    // Initialize the I2C Module
    I2C_Init();

    __delay_ms(100);

    // Replace with your EEPROM's address
    unsigned char block_address = 0x55;

    // Replace with the word address to read/write
    unsigned char word_address = 0;

    // Replace with the data to be written
    unsigned char eeprom_data = 8;

    int x = 0;

    while (1) {
        EEPROM_Write(block_address, word_address,eeprom_data);
        unsigned char read_data =
EEPROM_Read(block_address,        word_address);

        EUSART_Write_Text("Word Address: ");
        EUSART_Write_Integer(word_address);

        EUSART_Write_Text(", Written Data: ");
        EUSART_Write_Integer(eeprom_data);

        EUSART_Write_Text(", Read Data: ");
        EUSART_Write_Integer(read_data);
        EUSART_Write_Text("\n");

        // Increment the data for the next write
        eeprom_data+=2;

        // Increment the word address
        word_address++;

        // Delay between write-read cycles
        __delay_ms(5000);
    }
    return;
}
```

Listing 8-11 Interfacing PIC16F1719 to 24LC16B EEPROM

This program is designed for the PIC16F1719 microcontroller and demonstrates the use of the Inter-Integrated Circuit (I2C) bus. The specific application involves reading and writing data to a 24LC16B EEPROM memory chip. The EEPROM write function EEPROM_Write is responsible for writing data to the EEPROM. It utilizes the I2C communication protocol to send the block address, word address, and data to the EEPROM. We also have a function to do reads, the EEPROM read function EEPROM_Read, which reads data from the EEPROM. It uses I2C to send the block address and word address for reading, receives the data from the EEPROM, and then sends a NAK (Not Acknowledged) signal before stopping the communication.

Within the main program, we write data to the EEPROM using EEPROM_Write, which reads data from the EEPROM using EEPROM_Read and displays the results through EUSART communication. There are delays between each write-read cycle.

I3C (Improved Inter-Integrated Circuit) Protocol

Though I2C is very powerful and is widely used in the industry, recently a new standard has been created that is meant to fix some of the limitations of both I2C and SPI and add some improvements to these serial protocols. I2C in particular is known to suffer from bus collision and a 7-bit fixed address as well as limited data rates. I3C is an amalgamated approach to serial communications. I3C is backward compatible with I2C so we can reuse existing I2C devices with the I3C bus. I3C can reach speeds as high as 12.5 MHz and will gradually be integrated into new devices. I2C devices will be available for the foreseeable future; however, should you have the desire to work with the I3C protocol, there are devices such as the PIC18F06Q20 and PIC18F16Q20 PIC microcontrollers that have the hardware necessary to implement the protocol.

Conclusion

In this chapter, we looked at USART, SPI, and I2C, which are the fundamental communication protocols of microcontroller-based systems. We also looked at GPS, GSM, LCDs, and a host of other things. Once you understand these communication protocols, you can easily interface your microcontroller to a host of sensors. At this point, you can do quite a lot; however, keep reading because the next few chapters will take your skills to another level.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2024
A. Subero, *Programming PIC Microcontrollers with XC8*, Maker Innovations Series
https://doi.org/10.1007/979-8-8688-0467-0_9

9. Interfacing Displays

Armstrong Subero[1]

(1) Moruga, Trinidad and Tobago

In the realm of embedded systems, the ability to communicate with the outside world through visual feedback is fundamental. Displays, ranging from simple LEDs to complex graphical screens, serve as essential interfaces between machines and humans, offering a window into the operations of microcontrollers. In our modern society, it seems like displays are everywhere. Everything from coffee makers to stereos and watches now have intelligent displays on them. Modern smartphones also have a large display, and when designing devices, except for the most basic devices, your users will expect some sort of visual feedback. This chapter delves into the intricacies of interfacing displays with PIC microcontrollers, and we will look at using various display technologies including liquid crystal displays (LCDs), organic light-emitting diode (OLED) displays, and touch displays.

Displays

Displays serve as the primary interface for human-machine interaction in our embedded systems, offering a visual representation of data, graphics, and user inputs across a wide range of applications. “Human-machine interaction” and “human-computer interaction” (HCI) are so important that it has emerged as an entirely new discipline we can study.

Another field of study, user interface (UI) design is also based on how our applications look and feel on displays. From the simplest devices like digital watches to sophisticated systems like smartphones and automotive dashboards, we need displays to not only provide feedback to users but also to provide user interaction. The evolution of display technology over the years has significantly enhanced the functionality, efficiency, and aesthetics of electronic devices, making them more intuitive and user-friendly. The nice thing about the evolution of display technology is that we designers in the embedded space get the trickle-down of this technology. As technology advances, the types of displays that are available for us to use in our projects also change.

The simplest type of display available to us is an LED display. These are widely used for both small-scale applications, such as indicator lights and digital clocks, and large-scale applications, such as billboards and stadium screens. We already covered LED displays when we looked at seven-segment displays. So for this chapter, we will focus on other types of display technologies.

Liquid crystal displays (LCDs) are among the most commonly used types of displays, valued for their energy efficiency and the ability to produce sharp images with high resolution. Once you start using displays with microcontrollers, the first type of display you will use is an LCD. LCDs operate by manipulating light using liquid crystals and polarizers, allowing them to display text, images, and videos with minimal power consumption. This makes them particularly suitable for battery-operated devices, such as mobile phones, handheld gaming consoles, and laptops. Moreover, advancements in LCD technology, such as the development of Thin Film Transistor (TFT) LCDs, have further improved color accuracy, response times, and viewing angles, enhancing the overall visual experience. LCDs can be very simple such as the ones we will cover in this chapter, or they can be very large.

Organic light-emitting diode (OLED) displays represent a more recent innovation in display technology, known for their superior color contrast, deeper blacks, and wider viewing angles compared to LCDs. Unlike LCDs, OLED displays do not require a backlight, as they comprise organic materials that emit light when an electric current is applied. In embedded systems design, this is a blessing for us as creating a product and integrating a backlight with it can be cumbersome and bulky. Not using a backlight allows OLED panels to be thinner, more flexible, and capable of producing true blacks by turning off individual pixels, leading to significantly higher contrast ratios and energy savings in scenarios where dark scenes dominate the display content. This makes them great for the slim modern gadgets we have.

One type of LCD you will encounter a lot is chip-on-glass (COG) LCDs. These are used when you want to save power. You can run a microcontroller driving a COG LCD with some supercapacitors for weeks. Since these are so prevalent in modern gadgets and instruments, I thought I would take some time to talk about them. You can see what the typical COG display looks like in Figure 9-1.

Chip-on-glass (COG) technology represents a sophisticated advancement in display engineering, where the display's microcontroller or driver chip is directly mounted onto the glass substrate of the display itself. This integration technique offers several advantages over traditional display assembly methods; specifically, it makes products very compact. COG displays are commonly found in applications where space and power efficiency are paramount, such as in mobile devices, wearable technology, and various small-screen applications. The multimeter is one you are sure to have interacted with.

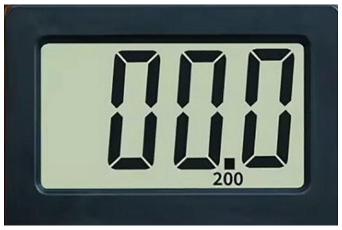


Figure 9-1 Typical COG display

The primary benefit of COG technology lies in its space-saving design. By eliminating the need for separate circuit boards for the display's driver components, COG displays can be made significantly thinner and lighter. This is particularly beneficial in modern electronic devices, where the trend is toward sleeker, more compact designs without compromising on performance or display quality. Additionally, the direct bonding of the chip to the glass substrate leads to improved signal integrity, which can result in better image quality and faster response times compared to traditional methods where the chip is mounted on a separate PCB.

Another advantage of COG displays is the potential for reduced manufacturing costs and increased reliability. The process of attaching the driver chip directly onto the display glass simplifies the assembly process, reducing the number of components and interconnections. This not only cuts down on production costs but also minimizes the points of failure, enhancing the overall durability of the device. With fewer connectors and flexible cables, the likelihood of issues such as loose connections or wear and tear over time is significantly reduced.

COG technology also offers advantages in terms of power efficiency. The shorter signal paths between the driver chip and the display pixels can lead to lower power consumption, which is critically important for battery-operated devices.

Despite its numerous advantages, the implementation of COG technology does require high precision and control during the manufacturing process, as the direct attachment of the chip to the glass substrate can be more complex than traditional mounting techniques. For this reason, as embedded designers, we typically use these when we are making a high-volume product.

Another prevalent technology you will come across, particularly when you want to save power, is E-Ink displays. E-Ink, short for Electronic Ink, is a revolutionary display technology that closely mimics the appearance of ink on paper, making it significantly different from traditional backlit displays. At its core, E-Ink displays utilize tiny microcapsules that contain positively charged white particles and negatively charged black particles suspended in a clear fluid. When an electric field is applied, these particles move within the microcapsules, allowing the display to show images or text in black and white. This technology is renowned for its extremely low power consumption, as it only requires power during the refresh phase when the display content changes, not while maintaining an image or text. Due to these characteristics, E-Ink displays are ideal for applications where reading comfort and battery life are critical, such as e-readers, smart labels, and outdoor signage.

In embedded systems, E-Ink displays offer unique advantages due to their low power requirements and excellent visibility under direct sunlight, unlike traditional LCD or OLED screens that can be challenging to read in bright conditions. These features make E-Ink particularly suitable for portable, battery-powered devices that require long battery life and readability in various lighting conditions. For instance, embedded applications like wearable devices, remote sensors, and portable medical devices benefit significantly from E-Ink's energy efficiency and clear, paper-like display. Furthermore, the bistable nature of E-Ink, meaning the display can hold an image without power, allows for the development of devices that can retain information on the screen even when turned off or when the battery is depleted, adding to their versatility in embedded systems design.

Comparing E-Ink to chip-on-glass (COG) technology reveals some fundamental differences in their applications and advantages. While COG focuses on integrating the display driver directly onto the glass substrate of the display for compactness and improved signal integrity, E-Ink emphasizes ultra-low power consumption and paper-like readability. COG is often employed in devices where space saving and direct control over display elements are paramount, such as in small electronic devices with LCD or OLED screens. On the other hand, E-Ink is chosen for its unique visual qualities and efficiency, particularly in applications where extended battery life and readability in direct sunlight are crucial. Although both technologies offer distinct benefits for embedded systems, the choice between them largely depends on the specific requirements of the application, such as the need for color, refresh rate, power efficiency, and form factor.

Character Display

The most common display you will encounter is the HD44780 character LCD. The Hitachi HD44780 is known as the industry standard character LCD. The reason is simple: the HD44780 is very easy to use. The LCD is used to display characters to users. The most commonly used display type is the 2x16 variety, which displays up to 16 characters on two lines. This display is shown in Figure 9-2.

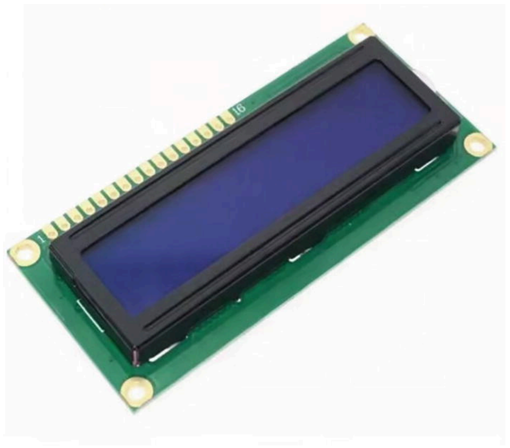


Figure 9-2 The HD44780 LCD

The LCD is essential for any embedded toolbox and makes an excellent prototyping display. The LCD commonly has 14 pins; however, LCDs that have a backlight have 16 pins. I recommend the version with the backlight. Download the datasheet for your particular display to determine which version display you have.

The HD44780 and its compatible LCDs stand as a cornerstone in the realm of alphanumeric liquid crystal displays, widely acclaimed for their ease of use, versatility, and straightforward interfacing with microcontrollers. At the heart of these displays is the HD44780 controller, a driver that manages the operations of the LCD at a low level, thereby simplifying the process of displaying text and characters. This controller operates in both 8-bit and 4-bit modes, allowing it to communicate with a wide range of microcontrollers with varying data bus widths. It manages the display's internal operations, such as character generation, display refresh, and cursor movements, through a set of predefined instruction sets. Users can send commands to the HD44780 to perform various functions like initializing the display, clearing the screen, controlling the display's on/off state, and setting the cursor position, making it highly adaptable to different display requirements.

The operation of HD44780-compatible LCDs involves sending data and commands from the microcontroller to the LCD through a parallel interface, typically consisting of data lines (D0−D7 for 8-bit, D4−D7 for 4-bit), a register select (RS) line, an enable (E) line, and a read/write (R/W) line. The RS line determines whether the sent data is treated as a command (RS=0) or as displayable data/character (RS=1). The enable line triggers the LCD to latch in the data presented on its data lines. In the 4-bit mode, data is sent in two halves, significantly reducing the number of I/O pins required from the microcontroller, making it ideal for projects with limited pin availability. The HD44780's built-in character generator ROM provides a selection of character patterns, enabling the display of a wide range of ASCII characters and symbols. This simplicity in operation, coupled with the ability to directly control pixels for custom character generation, makes HD44780-compatible LCDs highly favored for a multitude of applications, from simple DIY projects to complex industrial systems.

Let's look at the code for using the HD44780 LCD. The first thing we need to do is to create a header file in Listing 9-1.

```
/*
 * File: LCD.h
 * Author: Armstrong Subero
 * PIC: 16F1717 w/X OSC @ 16MHz, 5v
 * Program: Header file to
 * Compiler: XC8 (v1.38, MPLAX X v3.40)
 * Program Version 1.1
 *                *Added additional comments
 *
 * Program Description: This program header provides routines for
controlling a STD HITACHI HD44780 and compatible LCD's
 *
 * Hardware Description:
 *
 * RS ---> RD2
 * R/W ---> GND
 * EN ---> RD3
 * D4 ---> RD4
 * D5 ---> RD5
 * D6 ---> RD6
 * D7 ---> RD7
 *
 *
 * Created on November 7th, 2016, 11:56 PM
 * Updated on March 3rd, 2024, 10:00PM
 */

/*************************************************************
 *Includes and defines
 *************************************************************/
// STD XC8 include
#include <xc.h>

#define RS RD2  //Register Select (Character or Instruction)
#define EN RD3  //LCD Clock Enable PIN, Falling Edge Triggered

// 4 bit operation
#define D4 RD4  //Bit 4
#define D5 RD5  //Bit 5
#define D6 RD6  //Bit 6
#define D7 RD7  //Bit 7

// function prototypes
void Lcd_Port(char a);
void Lcd_Cmd(char a);
void Lcd_Clear();
void Lcd_Set_Cursor(char a, char b);
void Lcd_Init();
void Lcd_Write_Char(char a);
void Lcd_Write_String(const char *a);
void Lcd_Shift_Right();
void Lcd_Shift_Left();
void Lcd_Write_Integer(int v);
void Lcd_Write_Float(float f);
```

Listing 9-1 LCD Header File

This header file has function prototypes for controlling the LCD including things like initialization, setting the cursor, and writing strings and characters to the display. We implement these function prototypes in the LCD source file in Listing 9-2.

```
/*
 * File: LCD.c
 * Author: Armstrong Subero
 * PIC: 16F1717 w/Int OSC @ 16MHz, 5v
 * Program: Library file to configure PIC16F1717
 * Compiler: XC8 (v1.38, MPLAX X v3.40)
 * Program Version: 1.1
 *                *Added additional comments
 *
 * Program Description: This Library allows you to interface HD44780 and
Compatible LCD’s
 *
 * Created on November 7th, 2016, 11:55 AM
 * Updated on March 3rd, 2024, 10:20 PM
 */

#include "LCD.h"
#include "PIC16F1719_Internal.h"
#include <stdlib.h>
#include <string.h>

/*************************************************************
 * Function: void Lcd_Port (char a)
 *
 * Returns: Nothing
 *
 * Description: LCD Setup Routines
 ***********************************************************/

void Lcd_Port(char a)
{
   if(a & 1)
      D4 = 1;
   else
      D4 = 0;

   if(a & 2)
      D5 = 1;
   else
      D5 = 0;

   if(a & 4)
      D6 = 1;
   else
      D6 = 0;

   if(a & 8)
      D7 = 1;
   else
      D7 = 0;
}

/***********************************************************
 * Function: void Lcd_Cmd (char a)
 *
 * Returns: Nothing
 *
 * Description: Sets LCD command
 ***********************************************************/

void Lcd_Cmd(char a)
{
   RS = 0;             // => RS = 0
   Lcd_Port(a);
   EN  = 1;             // => E = 1
        __delay_ms(1);
        EN  = 0;             // => E = 0
}

/***********************************************************
 * Function: void Lcd_Clear()
 *
 * Returns: Nothing
 *
 * Description: Clears the LCD
 ***********************************************************/
```

```
void Lcd_Clear()
{
   Lcd_Cmd(0);
   Lcd_Cmd(1);
}

/**************************************************************
 * Function: void Lcd_Set_Cursor(char a, char b)
 *
 * Returns: Nothing
 *
 * Description: Sets the LCD cursor position
 **************************************************************/

void Lcd_Set_Cursor(char a, char b)
{
   char temp,z,y;
   if(a == 1)
   {
     temp = 0x80 + b - 1;
      z = temp>>4;
      y = temp & 0x0F;
      Lcd_Cmd(z);
      Lcd_Cmd(y);
   }
   else if(a == 2)
   {
      temp = 0xC0 + b - 1;
      z = temp>>4;
      y = temp & 0x0F;
      Lcd_Cmd(z);
      Lcd_Cmd(y);
   }
}

/**************************************************************
 * Function: void Lcd_Init()
 *
 * Returns: Nothing
 *
 * Description: Initialises the LCD
 **************************************************************/

void Lcd_Init()
{
  Lcd_Port(0x00);
   __delay_ms(10);
  Lcd_Cmd(0x03);
   __delay_ms(3);
  Lcd_Cmd(0x03);
   __delay_ms(10);
  Lcd_Cmd(0x03);
  /////////////////////////////////////////////////////
  Lcd_Cmd(0x02);
  Lcd_Cmd(0x02);
  Lcd_Cmd(0x08);
  Lcd_Cmd(0x00);
  Lcd_Cmd(0x0C);
  Lcd_Cmd(0x00);
  Lcd_Cmd(0x06);
}

/*************************************************************
 * Function: void Lcd_Write_Char (char a)
 *
 * Returns: Nothing
 *
 * Description: Writes a character to the LCD
 *************************************************************/

void Lcd_Write_Char(char a)
{
   char temp,y;
   temp = a&0x0F;
   y = a&0xF0;
   RS = 1;             // => RS = 1
   Lcd_Port(y>>4);     //Data transfer
   EN = 1;
   __delay_us(20);
   EN = 0;
   Lcd_Port(temp);
   EN = 1;
   __delay_us(20);
   EN = 0;
}

/*************************************************************
 * Function: void Lcd_Write_String (const char *a)
 *
 * Returns: Nothing
 *
 * Description: Writes a string to the LCD
 *************************************************************/

void Lcd_Write_String(const char *a)
{
   int i;
   for(i=0;a[i]!='\0';i++)
      Lcd_Write_Char(a[i]);
}

/*************************************************************
 * Function: void Lcd_Shift_Right()
 *
 * Returns: Nothing
 *
 * Description: Shifts text on the LCD right
 **************************************************************/

void Lcd_Shift_Right()
{
   Lcd_Cmd(0x01);
   Lcd_Cmd(0x0C);
}

/**************************************************************
 * Function: void Lcd_Shift_Left()
 *
 * Returns: Nothing
 *
 * Description: Shifts text on the LCD left
 **************************************************************/

void Lcd_Shift_Left()
{
   Lcd_Cmd(0x01);
   Lcd_Cmd(0x08);
}

/**************************************************************
 * Function: void Lcd_Write_Integer(int v)
 *
 * Returns: Nothing
 *
 * Description: Converts a string to an integer
 **************************************************************/

// A simple itoa function for base 10 conversion
char* c_itoa(int value, char* result, int base) {
    // Check that the base is valid
    if (base < 2 || base > 36) {
        *result = '\0';
        return result;
    }

    char* ptr = result, *ptr1 = result, tmp_char;
    int tmp_value;

    // Handle negative values if necessary
    if (value < 0 && base == 10) {
        *ptr++ = '-';
    }

    ptr1 = ptr;

    do {
        tmp_value = value;
        value /= base;
        *ptr++ =
"zyxwvutsrqponmlkjihgfedcba9876543210123456789abcdefghijklmnopqrstuvwxyz"
[35 + (tmp_value - value * base)];
    } while ( value );

    // Apply terminating null character
    *ptr-- = '\0';
    while(ptr1 < ptr) {
        tmp_char = *ptr;
        *ptr--= *ptr1;
        *ptr1++ = tmp_char;
    }
    return result;
}

void Lcd_Write_Integer(int v)
{
    unsigned char buf[8];

    Lcd_Write_String(c_itoa(buf, v, 10));
}

/*************************************************************
 * Function: void Lcd_Write_Float(float f)
 *
 * Returns: Nothing
 *
 * Description: Converts a string to a float
 *************************************************************/

char * c_ftoa(float f, int * status)
{
    static char buf[17];
    char * cp = buf;
    unsigned long l, rem;

    if(f < 0) {
        *cp++ = '-';
        f = -f;
    }
    l = (unsigned long)f;
    f -= (float)l;
    rem = (unsigned long)(f * 1e6);
    sprintf(cp, "%lu.%6.6lu", l, rem);
    return buf;
}

void Lcd_Write_Float(float f)
{
    char* buf11;
    int status;

    buf11 = c_ftoa(f, &status);

    Lcd_Write_String(buf11);
}
```

Listing 9-2 The LCD Source File

At the heart of the library are functions designed for sending commands and data to the LCD. The Lcd_Port function is responsible for setting the state of the data pins (D4 to D7) based on the input character's bit pattern, enabling the microcontroller to communicate with the LCD at a 4-bit level. The Lcd_Cmd function sends commands to the LCD to perform various operations, such as clearing the display or setting the cursor position, by first setting the RS pin to indicate a command mode, then applying the command, and finally toggling the EN pin to process the command. The Lcd_Init function sequences through a series of commands to properly initialize the LCD, preparing it for data display.

Additional functions enhance the library's usability for specific display tasks. Lcd_Write_Char and Lcd_Write_String allow the user to display individual characters or strings on the LCD, respectively. For dynamic content, Lcd_Write_Integer and Lcd_Write_Float provide the capability to convert integers and floating-point numbers to their string representations and display them. These conversions are handled by custom c_itoa and c_ftoa functions, which convert integer and float values into strings. The library also includes features for manipulating the display's content visually, such as Lcd_Shift_Right and Lcd_Shift_Left, which scroll the text across the LCD in Listing 9-3.

```
/*
 * File: Main.c
 * Author: Armstrong Subero
 * PIC: 16F1717 w/Int OSC @ 16MHz, 5v
 * Program: 15_HD44780_LCD
 * Compiler: XC8 (v1.38, MPLAX X v3.40)
 * Program Version: 1.0
 *
 *
 * Program Description: This Program Allows PIC16F1717 to interface to
HD44780 and compatible LCD's
 *
 *
 * Hardware Description: An HD44780 compatible LCD is connected to PORTD
of the microcontroller as follows:
 *
 *                        RS ---> RD2
 *                        R/W ---> GND
 *                        EN ---> RD3
 *                        D4 ---> RD4
 *                        D5 ---> RD5
 *                        D6 ---> RD6
 *                        D7 ---> RD7
 *
 *
 * Created November 7th, 2016, 11:05 AM
 */

/************************************************************
 *Includes and defines
 ************************************************************/

#include "16F1717_Internal.h"
#include "LCD.h"

/************************************************************
 * Function: void initMain()
 *
 * Returns: Nothing
 *
 * Description: Contains initializations for main
 *
 * Usage: initMain()
 *************************************************************/

void initMain(){
    // Run at 16 MHz
    internal_16();

    TRISD = 0x00;
    ANSELD = 0x00;
    PORTD = 0x00;
}

/*************************************************************
 * Function: Main
 *
 * Returns: Nothing
 *
 * Description: Program entry point
 *************************************************************/

void main(void) {
    initMain();
    Lcd_Init();
    __delay_ms(1000);

    int a;
    int c;
    float b;

    while(1){
        Lcd_Clear();
        Lcd_Set_Cursor(1,1);
        __delay_ms(100);

        // Write String
        Lcd_Write_String("PIC16F1717");
        __delay_ms(10);

        // Shift it left
        for(a=0;a<15;a++)
        {
            __delay_ms(500);
            Lcd_Shift_Left();
        }

        // Shift it right
        for(a=0;a<15;a++)
        {
            __delay_ms(500);
```

```
                Lcd_Shift_Right();
            }

            Lcd_Clear();
            Lcd_Set_Cursor(1,1);

            // Write Integer
            for (c = 0; c < 100; c++){
                Lcd_Write_Integer(c);
                __delay_ms(300);
                Lcd_Clear();
                __delay_ms(15);

            }

            // Write Float
            for (b = 0.0; b <= 5; b+= 0.5)
            {
                Lcd_Write_Float(b);
                __delay_ms(300);
                Lcd_Clear();
                __delay_ms(15);
            }

            __delay_ms(1000);
}

    return;
}
```

Listing 9-3 LCD Source File

OLED Display

Organic light-emitting diodes (OLEDs) represent a cutting-edge display technology that offers superior contrast, color saturation, and energy efficiency compared to traditional liquid crystal displays (LCDs). Unlike LCDs, which require a backlight to illuminate pixels, OLED displays are made up of organic compounds that emit light when an electric current is passed through them. This self-illuminating property allows OLEDs to produce true blacks and vibrant colors, as each pixel can be individually turned on or off. The flexibility of the organic layers also enables the production of flexible, bendable, and even rollable displays, opening up new possibilities for innovative device designs. OLEDs are widely used in various applications, including smartphones, televisions, wearable devices, and digital signage, thanks to their thin profiles, deep color representation, and excellent viewing angles.

OLED technology not only stands out for its visual excellence but also for its efficiency. Since OLED pixels emit their light, no energy is wasted on lighting up unneeded parts of the screen, especially in scenes with a lot of dark areas. This leads to better battery life in portable devices, making OLEDs especially attractive for mobile phones and wearable technology. Furthermore, the response time of OLEDs is much faster than that of traditional LCDs, resulting in smoother animations and transitions with virtually no motion blur. These attributes make OLED displays a premium choice for a wide range of applications, from high-end consumer electronics to professional-grade monitors.

The SSD1306 is a popular OLED display driver IC that is widely used in the DIY electronics community and by professionals alike. It is designed to drive 128×64 pixel OLED displays via I2C (Inter-Integrated Circuit), a serial communication protocol that uses only two wires for communication, making it ideal for microcontroller-based projects where pin resources are limited. The SSD1306 is depicted in Figure 9-3.

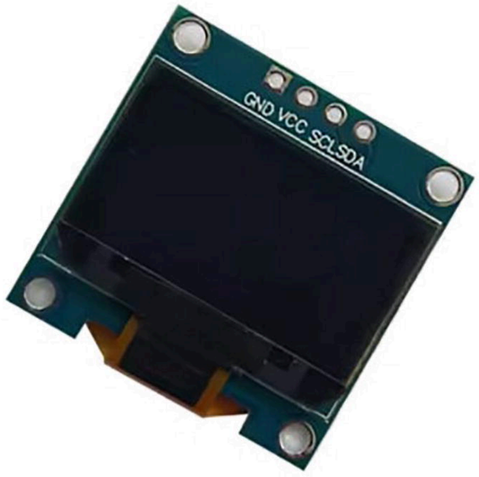


Figure 9-3 The SSD1306 OLED

The SSD1306 module integrates a controller that manages the OLED panel, providing a simple interface for drawing text, graphics, and animations. It supports multiple addressing modes, allowing for efficient updating of the display content. The module's ability to work over I2C means that it can be easily connected to a wide range of microcontrollers and development boards, making it highly versatile for building interactive projects and user interfaces.

Operating the SSD1306 I2C display module involves sending commands and data over the I2C bus to control the display's internal settings, such as contrast and display orientation, as well as to define the pixel data for display. The microcontroller communicates with the SSD1306 using the I2C protocol, where it acts as the master sending commands to the SSD1306, which acts as the slave device. This setup allows for the dynamic display of text, graphics, and animations with low power consumption. Libraries and software support for the SSD1306 are widely available across various programming platforms, including Arduino and Raspberry Pi, providing functions for easy drawing and text rendering. This extensive support, combined with the module's low power consumption and high contrast display, makes the SSD1306 OLED display a preferred choice for portable, battery-powered applications and projects requiring high-quality visual output. The header file for the OLED is given in Listing 9-4.

```
/*
 * File: oled.h
 * Author: Armstrong Subero
 * PIC: 16F1717 w/X OSC @ 16MHz, 5v
 * Program: Header file to setup PIC16F1717 I2C
 * Compiler: XC8 (v1.35, MPLAX X v3.10)
 * Program Version 1.3
 *                 * Separated file into Header and C source
 *                 * Replace fixed hex with macros
 *                 * Added additional comments
 *
 *
 * Program Description: This program header will allow setup of SSD 1306
OLEDs
 *
 * Created on March 10th, 2017, 8:00 PM
 */

// Define OLED dimensions
#define OLED_WIDTH 128
#define OLED_HEIGHT 64

// Define command macros
#define OLED_SETCONTRAST 0x81
#define OLED_DISPLAYALLON_RESUME 0xA4
#define OLED_DISPLAYALLON 0xA5
#define OLED_NORMALDISPLAY 0xA6
#define OLED_INVERTDISPLAY 0xA7
#define OLED_DISPLAYOFF 0xAE
#define OLED_DISPLAYON 0xAF
#define OLED_SETDISPLAYOFFSET 0xD3
#define OLED_SETCOMPINS 0xDA
#define OLED_SETVCOMDETECT 0xDB
#define OLED_SETDISPLAYCLOCKDIV 0xD5
#define OLED_SETPRECHARGE 0xD9
#define OLED_SETMULTIPLEX 0xA8
#define OLED_SETLOWCOLUMN 0x00
#define OLED_SETHIGHCOLUMN 0x10
#define OLED_SETSTARTLINE 0x40
#define OLED_MEMORYMODE 0x20
#define OLED_COLUMNADDR 0x21
#define OLED_PAGEADDR   0x22
#define OLED_COMSCANINC 0xC0
#define OLED_COMSCANDEC 0xC8
#define OLED_SEGREMAP 0xA0
#define OLED_CHARGEPUMP 0x8D

// Header file
#include "PIC16F1719_Internal.h"

// Function declarations
void OLED_Command( uint8_t temp);
void OLED_Data( uint8_t temp);
void OLED_Init();
void OLED_YX(unsigned char Row, unsigned char Column); // *warning!* max
4 rows
void OLED_PutChar( char ch );
void OLED_Clear();
void OLED_Write_String( char *s );
void OLED_Write_Integer(uint16_t i);
void OLED_Write_Float(float f);
```

Listing 9-4 OLED Header File

The OLED source file is given in Listing 9-5.

```
/*
 * File: oled.c
 * Author: Armstrong Subero
 * PIC: 16F1717 w/Int OSC @ 16MHz, 5v
 * Program: Library file for SSD 1306 OLED
 * Compiler: XC8 (v1.38, MPLAX X v3.40)
 * Program Version: 1.1
 *                *Added additional comments
 *
 * Program Description: This Library allows you to control the SSD 1306
OLED
 *
 * Created on March 10th, 2017, 8:05 PM
 */

#include "oled.h"
#include "I2C.h"
#include <string.h>

/*************************************************************
 * Function: const uint8_t OledFont[][8]
 *
 * Returns: Nothing
 *
 * Description: 2 Dimdimensional array containing the the ASCII
characters
 *
 *************************************************************/

const uint8_t OledFont[][8] =
{
  {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
  {0x00,0x00,0x5F,0x00,0x00,0x00,0x00,0x00},
  {0x00,0x00,0x07,0x00,0x07,0x00,0x00,0x00},
  {0x00,0x14,0x7F,0x14,0x7F,0x14,0x00,0x00},
  {0x00,0x24,0x2A,0x7F,0x2A,0x12,0x00,0x00},
  {0x00,0x23,0x13,0x08,0x64,0x62,0x00,0x00},
  {0x00,0x36,0x49,0x55,0x22,0x50,0x00,0x00},
  {0x00,0x00,0x05,0x03,0x00,0x00,0x00,0x00},
  {0x00,0x1C,0x22,0x41,0x00,0x00,0x00,0x00},
  {0x00,0x41,0x22,0x1C,0x00,0x00,0x00,0x00},
  {0x00,0x08,0x2A,0x1C,0x2A,0x08,0x00,0x00},
  {0x00,0x08,0x08,0x3E,0x08,0x08,0x00,0x00},
  {0x00,0xA0,0x60,0x00,0x00,0x00,0x00,0x00},
  {0x00,0x08,0x08,0x08,0x08,0x08,0x00,0x00},
  {0x00,0x60,0x60,0x00,0x00,0x00,0x00,0x00},
  {0x00,0x20,0x10,0x08,0x04,0x02,0x00,0x00},
  {0x00,0x3E,0x51,0x49,0x45,0x3E,0x00,0x00},
  {0x00,0x00,0x42,0x7F,0x40,0x00,0x00,0x00},
  {0x00,0x62,0x51,0x49,0x49,0x46,0x00,0x00},
  {0x00,0x22,0x41,0x49,0x49,0x36,0x00,0x00},
  {0x00,0x18,0x14,0x12,0x7F,0x10,0x00,0x00},
  {0x00,0x27,0x45,0x45,0x45,0x39,0x00,0x00},
  {0x00,0x3C,0x4A,0x49,0x49,0x30,0x00,0x00},
  {0x00,0x01,0x71,0x09,0x05,0x03,0x00,0x00},
  {0x00,0x36,0x49,0x49,0x49,0x36,0x00,0x00},
  {0x00,0x06,0x49,0x49,0x29,0x1E,0x00,0x00},
  {0x00,0x00,0x36,0x36,0x00,0x00,0x00,0x00},
  {0x00,0x00,0xAC,0x6C,0x00,0x00,0x00,0x00},
  {0x00,0x08,0x14,0x22,0x41,0x00,0x00,0x00},
  {0x00,0x14,0x14,0x14,0x14,0x14,0x00,0x00},
  {0x00,0x41,0x22,0x14,0x08,0x00,0x00,0x00},
  {0x00,0x02,0x01,0x51,0x09,0x06,0x00,0x00},
  {0x00,0x32,0x49,0x79,0x41,0x3E,0x00,0x00},
  {0x00,0x7E,0x09,0x09,0x09,0x7E,0x00,0x00},
  {0x00,0x7F,0x49,0x49,0x49,0x36,0x00,0x00},
  {0x00,0x3E,0x41,0x41,0x41,0x22,0x00,0x00},
  {0x00,0x7F,0x41,0x41,0x22,0x1C,0x00,0x00},
  {0x00,0x7F,0x49,0x49,0x49,0x41,0x00,0x00},
  {0x00,0x7F,0x09,0x09,0x09,0x01,0x00,0x00},
  {0x00,0x3E,0x41,0x41,0x51,0x72,0x00,0x00},
  {0x00,0x7F,0x08,0x08,0x08,0x7F,0x00,0x00},
  {0x00,0x41,0x7F,0x41,0x00,0x00,0x00,0x00},
  {0x00,0x20,0x40,0x41,0x3F,0x01,0x00,0x00},
  {0x00,0x7F,0x08,0x14,0x22,0x41,0x00,0x00},
  {0x00,0x7F,0x40,0x40,0x40,0x40,0x00,0x00},
  {0x00,0x7F,0x02,0x0C,0x02,0x7F,0x00,0x00},
  {0x00,0x7F,0x04,0x08,0x10,0x7F,0x00,0x00},
  {0x00,0x3E,0x41,0x41,0x41,0x3E,0x00,0x00},
  {0x00,0x7F,0x09,0x09,0x09,0x06,0x00,0x00},
  {0x00,0x3E,0x41,0x51,0x21,0x5E,0x00,0x00},
  {0x00,0x7F,0x09,0x19,0x29,0x46,0x00,0x00},
  {0x00,0x26,0x49,0x49,0x49,0x32,0x00,0x00},
  {0x00,0x01,0x01,0x7F,0x01,0x01,0x00,0x00},
  {0x00,0x3F,0x40,0x40,0x40,0x3F,0x00,0x00},
  {0x00,0x1F,0x20,0x40,0x20,0x1F,0x00,0x00},
  {0x00,0x3F,0x40,0x38,0x40,0x3F,0x00,0x00},
  {0x00,0x63,0x14,0x08,0x14,0x63,0x00,0x00},
  {0x00,0x03,0x04,0x78,0x04,0x03,0x00,0x00},
  {0x00,0x61,0x51,0x49,0x45,0x43,0x00,0x00},
  {0x00,0x7F,0x41,0x41,0x00,0x00,0x00,0x00},
  {0x00,0x02,0x04,0x08,0x10,0x20,0x00,0x00},
  {0x00,0x41,0x41,0x7F,0x00,0x00,0x00,0x00},
  {0x00,0x04,0x02,0x01,0x02,0x04,0x00,0x00},
  {0x00,0x80,0x80,0x80,0x80,0x80,0x00,0x00},
  {0x00,0x01,0x02,0x04,0x00,0x00,0x00,0x00},
  {0x00,0x20,0x54,0x54,0x54,0x78,0x00,0x00},
  {0x00,0x7F,0x48,0x44,0x44,0x38,0x00,0x00},
  {0x00,0x38,0x44,0x44,0x28,0x00,0x00,0x00},
  {0x00,0x38,0x44,0x44,0x48,0x7F,0x00,0x00},
  {0x00,0x38,0x54,0x54,0x54,0x18,0x00,0x00},
  {0x00,0x08,0x7E,0x09,0x02,0x00,0x00,0x00},
  {0x00,0x18,0xA4,0xA4,0xA4,0x7C,0x00,0x00},
  {0x00,0x7F,0x08,0x04,0x04,0x78,0x00,0x00},
  {0x00,0x00,0x7D,0x00,0x00,0x00,0x00,0x00},
  {0x00,0x80,0x84,0x7D,0x00,0x00,0x00,0x00},
  {0x00,0x7F,0x10,0x28,0x44,0x00,0x00,0x00},
  {0x00,0x41,0x7F,0x40,0x00,0x00,0x00,0x00},
  {0x00,0x7C,0x04,0x18,0x04,0x78,0x00,0x00},
  {0x00,0x7C,0x08,0x04,0x7C,0x00,0x00,0x00},
  {0x00,0x38,0x44,0x44,0x38,0x00,0x00,0x00},
  {0x00,0xFC,0x24,0x24,0x18,0x00,0x00,0x00},
  {0x00,0x18,0x24,0x24,0xFC,0x00,0x00,0x00},
  {0x00,0x00,0x7C,0x08,0x04,0x00,0x00,0x00},
  {0x00,0x48,0x54,0x54,0x24,0x00,0x00,0x00},
  {0x00,0x04,0x7F,0x44,0x00,0x00,0x00,0x00},
  {0x00,0x3C,0x40,0x40,0x7C,0x00,0x00,0x00},
  {0x00,0x1C,0x20,0x40,0x20,0x1C,0x00,0x00},
  {0x00,0x3C,0x40,0x30,0x40,0x3C,0x00,0x00},
  {0x00,0x44,0x28,0x10,0x28,0x44,0x00,0x00},
  {0x00,0x1C,0xA0,0xA0,0x7C,0x00,0x00,0x00},
  {0x00,0x44,0x64,0x54,0x4C,0x44,0x00,0x00},
  {0x00,0x08,0x36,0x41,0x00,0x00,0x00,0x00},
  {0x00,0x00,0x7F,0x00,0x00,0x00,0x00,0x00},
  {0x00,0x41,0x36,0x08,0x00,0x00,0x00,0x00},
  {0x00,0x02,0x01,0x01,0x02,0x01,0x00,0x00},
  {0x00,0x02,0x05,0x05,0x02,0x00,0x00,0x00},
};

/*************************************************************
 * Function: void OLED_Command( uint8_t temp)
 *
 * Returns: Nothing
 *
 * Description: sends commands to the OLED
 *
 *************************************************************/

void OLED_Command( uint8_t temp){

    Send_I2C_StartBit();                    // send start bit
    Send_I2C_Data(0x3C << 1);               // send word address
    Send_I2C_Data(0x00);
    Send_I2C_Data(temp);                    // send data byte
    Send_I2C_StopBit();                     // send stop bit
}

/*************************************************************
 * Function: void OLED_Data ( uint8_t temp)
 *
 * Returns: Nothing
 *
 * Description: sends data to the OLED
 *
 *************************************************************/

void OLED_Data( uint8_t temp){

    Send_I2C_StartBit();                    // send start bit
    Send_I2C_Data(0x3C << 1);               // send word address
    Send_I2C_Data(0x40);
    Send_I2C_Data(temp);                    // send data byte
    Send_I2C_StopBit();                     // send stop bit
}
```

```
/*******************************************************
 * Function: void OLED_Init ()
 *
 * Returns: Nothing
 *
 * Description: Initializes OLED
 *
 *******************************************************/

void OLED_Init() {

    OLED_Command(OLED_DISPLAYOFF);          // 0xAE
    OLED_Command(OLED_SETDISPLAYCLOCKDIV); // 0xD5
    OLED_Command(0x80);                     // the suggested ratio 0x80
    OLED_Command(OLED_SETMULTIPLEX);        // 0xA8
    OLED_Command(0x1F);
    OLED_Command(OLED_SETDISPLAYOFFSET);    // 0xD3
    OLED_Command(0x0);                      // no offset
    OLED_Command(OLED_SETSTARTLINE | 0x0); // line #0
    OLED_Command(OLED_CHARGEPUMP);          // 0x8D
    OLED_Command(0xAF);
    OLED_Command(OLED_MEMORYMODE);          // 0x20
    OLED_Command(0x00);                     // 0x0 act like ks0108
    OLED_Command(OLED_SEGREMAP | 0x1);
    OLED_Command(OLED_COMSCANDEC);
    OLED_Command(OLED_SETCOMPINS);          // 0xDA
    OLED_Command(0x02);
    OLED_Command(OLED_SETCONTRAST);         // 0x81
    OLED_Command(0x8F);
    OLED_Command(OLED_SETPRECHARGE);        // 0xd9
    OLED_Command(0xF1);
    OLED_Command(OLED_SETVCOMDETECT);       // 0xDB
    OLED_Command(0x40);
    OLED_Command(OLED_DISPLAYALLON_RESUME);// 0xA4
    OLED_Command(OLED_NORMALDISPLAY);       // 0xA6
    OLED_Command(OLED_DISPLAYON);           //--turn on oled panel

}

/********************************************************
 * Function: void OLED_YX(unsigned char Row, unsigned char Column)
 *
 * Returns: Nothing
 *
 * Description: Sets the X and Y coordinates
 *
 ********************************************************/

void OLED_YX(unsigned char Row, unsigned char Column)
{
    OLED_Command( 0xB0 + Row);
    OLED_Command( 0x00 + (8*Column & 0x0F) );
    OLED_Command( 0x10 + ((8*Column>>4)&0x0F) );
}

/*************************************************************
 * Function: void OLED_PutChar(char ch)
 *
 * Returns: Nothing
 *
 * Description: Writes a character to the OLED
 *
 ************************************************************/

void OLED_PutChar( char ch )
{
    if ( ( ch < 32 ) || ( ch > 127 ) ){
        ch = ' ';
    }

    const uint8_t *base = &OledFont[ch - 32][0];

    uint8_t bytes[9];
    bytes[0] = 0x40;
    memmove( bytes + 1, base, 8 );

    Send_I2C_StartBit();                    // send start bit
    Send_I2C_Data(0x3C << 1);            // send word address
    Send_I2C_Data(0x40);

    int i;

    for (i = 1; i <= 8; i++){
         Send_I2C_Data(bytes[i]);
    }

    Send_I2C_StopBit();                     // send stop bit
}

/************************************************************
 * Function: void OLED_Clear()
 *
 * Returns: Nothing
 *
 * Description: Clears the OLED
 *
 *************************************************************/

void OLED_Clear()
{
    for ( uint16_t row = 0; row < 8; row++ ) {
        for ( uint16_t col = 0; col < 16; col++ ) {
            OLED_YX( row, col );
            OLED_PutChar(' ');
        }
    }
}
```

```
/*********************************************************
 * Function:  void OLED_Write_String( char *s )
 *
 * Returns: Nothing
 *
 * Description: Writes a string to the OLED
 *
 *********************************************************/

void OLED_Write_String( char *s )
{
    while (*s) OLED_PutChar( *s++);
}
```

Listing 9-5 OLED Source File

Touch Screen LCD

Thanks to the smartphone revolution, every user wants a screen that they can touch and interact with. It is for this very reason that I chose to include interfacing touch screens in this book. Smartphones have encouraged the wide availability of touch screens. Many people think that you need a lot of processing power to use touch screens. In the past, this was true, but with the advent of intelligent display modules, this is no longer the case. Intelligent displays have a processor onboard that handles drawing and updating the display. The application processor can thus interact with the display using simple commands. What this means is that even 8-bit microcontroller solutions can utilize a touch screen display.

When integrating touch-based displays into your design, it is important to consider the following factors:

- The touch screen must have a GUI interface that can be developed quickly.
- The touch screen must be easy to integrate into your projects.
- The touch screen must be cost-effective. In addition to these factors, we must also examine the types of touch displays that are available on the market today.

There are two main types of touch displays available: resistive touch screens and capacitive touch screens. We take a look at each of these types of displays in the next sections.

Resistive Touch

The resistive touch screen essentially consists of two layers of flexible sheets, which are then placed on a piece of glass. These sheets are clad in a substance that has a certain resistance and is kept apart by small dots. When a part of the screen is pressed, the two layers are pressed together, and this change in resistance at that touch point is measured. Resistive touch screens require a hard object to press them together, such as a stylus, fingernail, or a sufficiently hard object. A major advantage of a resistive touch screen is that it can be used through electrically insulating materials such as when wearing gloves. A major disadvantage is that they are not as responsive as capacitive touch screens.

Capacitive Touch

The human body is known to have electrical properties. One of these properties is the fact that the human body is a conductor. Capacitive touch screens exploit this aspect of the human body. The capacitive touch screen consists of glass covered with a conductive material. When the material is touched, it produces a change in capacitance, which is measured and used to determine where the touch took place. A major advantage of capacitive touch screens is that they are very responsive, and a major disadvantage is that, unlike resistive touch screens, they cannot be used through electrically insulating materials.

Selecting a Touch Screen LCD

Now that you have a basic understanding of the factors to be determined when selecting a touch screen and have learned about the types of touch screens, you can select a touch screen to use in your project. The Nextion series of displays was chosen for this example because they have an editor tool that enables you to quickly develop the GUI. They also communicate via the ubiquitous UART protocol and are some of the lowest-cost displays available on the market today. They are resistive touch screens. Before you continue with this section, I highly recommend you go through the tutorials on the Nextion website at `https://nextion.itead.cc/`.

Using the Touch LCD

Here are the steps required to use the Nextion-type displays:

1. Create the layout in your photo editor of choice.

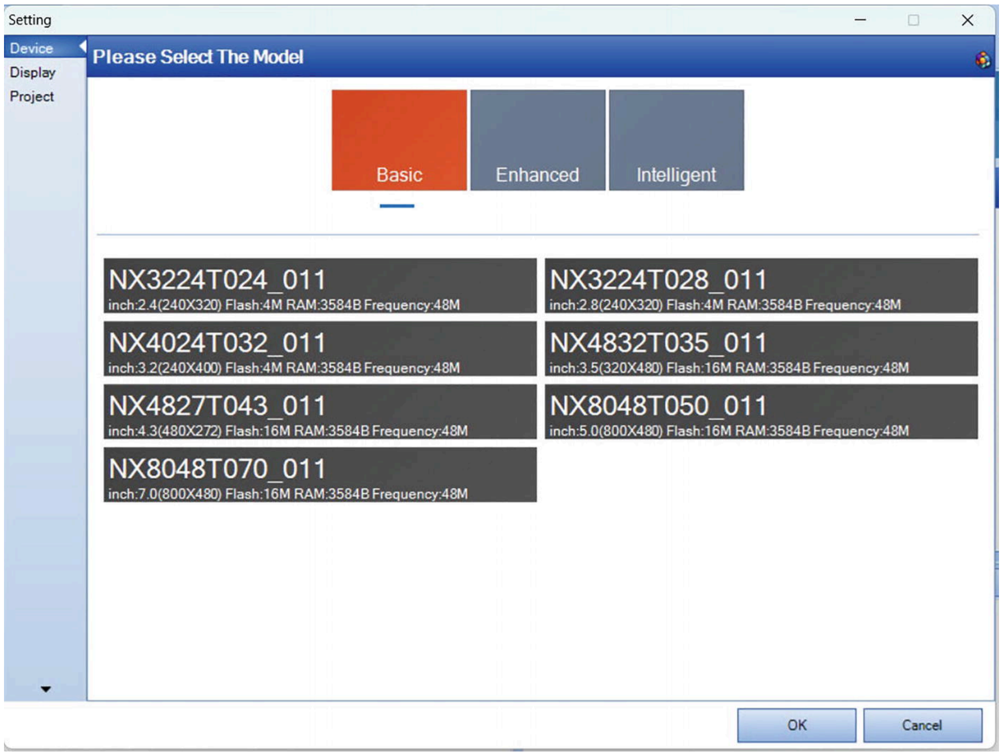
2. Add widgets to your layout with the Nextion Editor.

3. Add code to those widgets via the Nextion Editor.

4. Read information sent from the display by the microcontroller.

Creating a Layout

The first step is to create the layout of your choice (see Figure 9-4). If you want to use solid colors only in your design, you do not need to use any photo editing software to create a layout. However, if you plan on adding a decent-looking background to your design, you will need images. I recommend that you purchase images from the many sites available that provide such a service. This will ensure that you are not violating any copyrights and you will have high-quality images. Create a new project and select “Basic,” “Enhanced,” or “Intelligent” based on the display type you have. In this book, I use a display from the “Basic” section as shown in Figure 9-4.



NX3224T024_011
inch:2.4(240X320) Flash:4M RAM:3584B Frequency:48M

NX3224T028_011
inch:2.8(240X320) Flash:4M RAM:3584B Frequency:48M

NX4024T032_011
inch:3.2(240X400) Flash:4M RAM:3584B Frequency:48M

NX4832T035_011
inch:3.5(320X480) Flash:16M RAM:3584B Frequency:48M

NX4827T043_011
inch:4.3(480X272) Flash:16M RAM:3584B Frequency:48M

NX8048T050_011
inch:5.0(800X480) Flash:16M RAM:3584B Frequency:48M

NX8048T070_011
inch:7.0(800X480) Flash:16M RAM:3584B Frequency:48M

Figure 9-4 Selecting a display

We then select a display direction as a 90-degree horizontal display as shown in Figure 9-5.

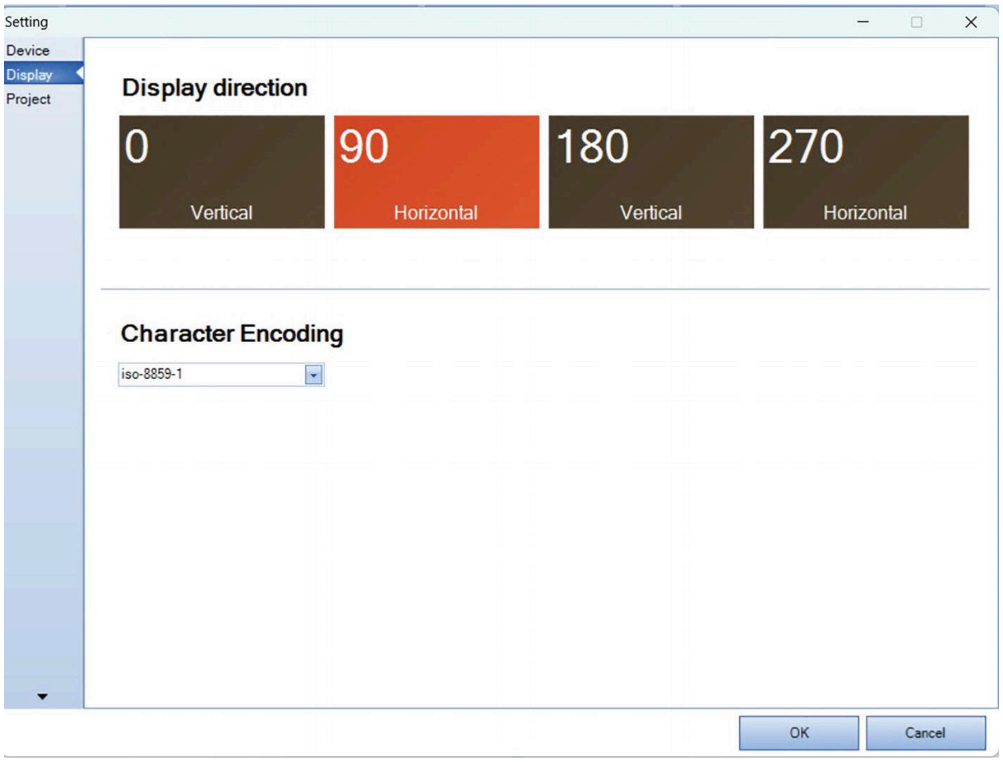


Figure 9-5 Selecting a display direction

Once we set up our display and direction, add four buttons from the toolbox by dragging them to page0 as shown in Figure 9-6.

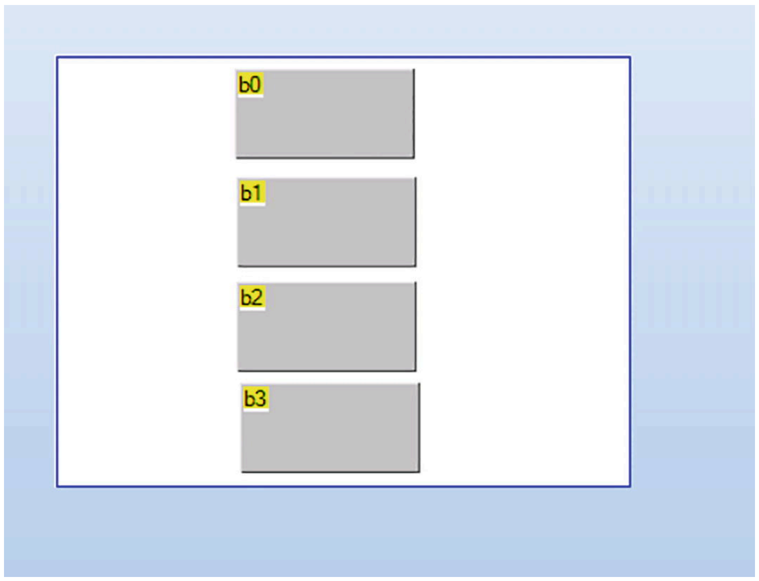
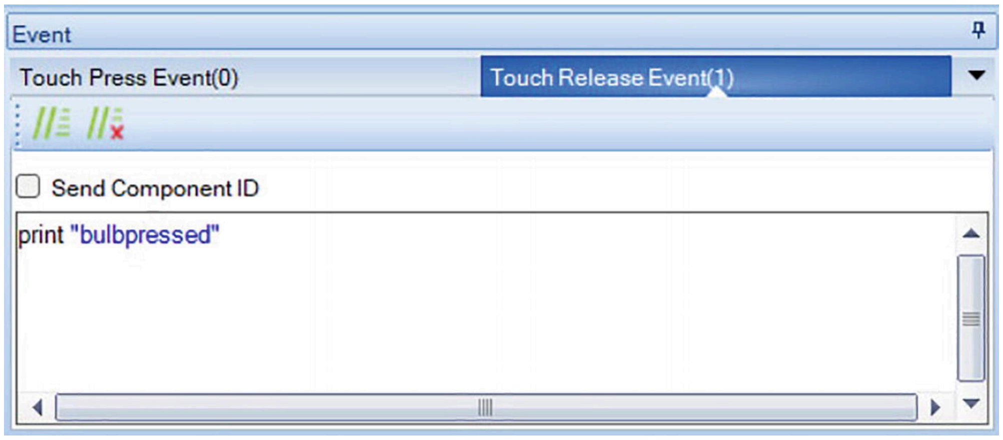


Figure 9-6 Adding buttons to page0

Adding Code

The touch screens we are using are intelligent displays. Thus, code can be written on the display itself and sent to the microcontroller. The Nextion editor includes a section for adding code to the display. You add the code in the "Touch Release Event" section of each button.

The code is added to each widget (see Figure 9-7). The print command in the editor allows you to print a string of characters when a particular touch event of that particular widget is triggered. In this example, they are bulbpressed, motopressed, planpressed, and compressed. So to add bulbpressed, for example, you would type print bulbpressed in the editor as shown in Figure 9-7.



```
print "bulbpressed"
```

Figure 9-7 Adding an event

If you try to compile your current setup, you will get an error. So you will need to add a font before. Select "Tools" and then "Font Generator", then add a name for your font, and select a save location. I named my font "sans". Once you click the "Generate Font" button, you will be able to compile your project as shown in Figure 9-8.

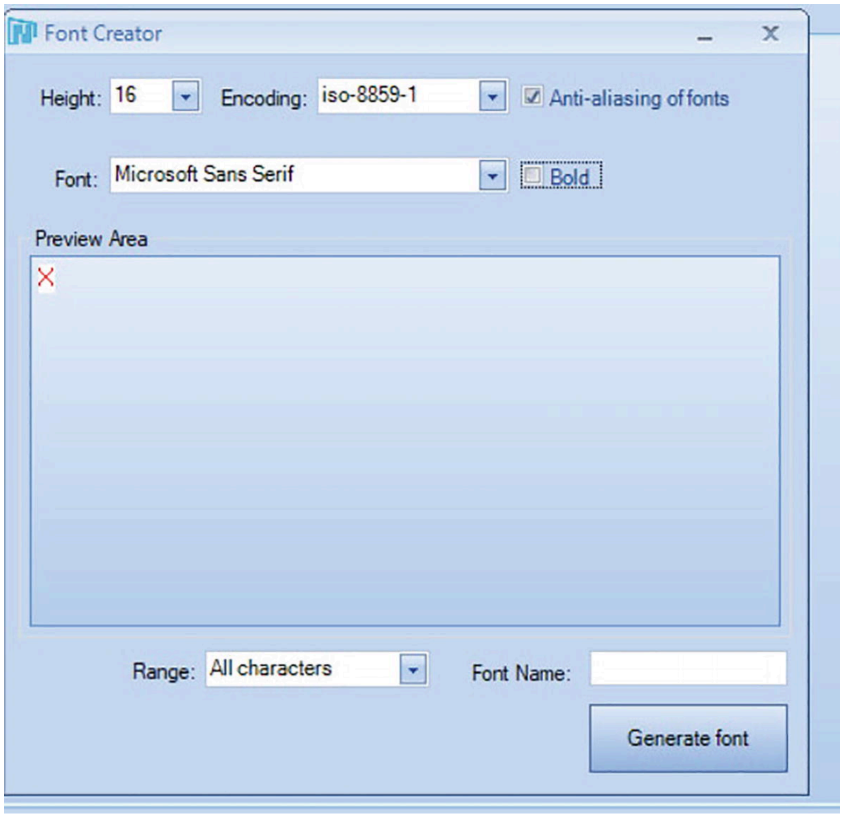
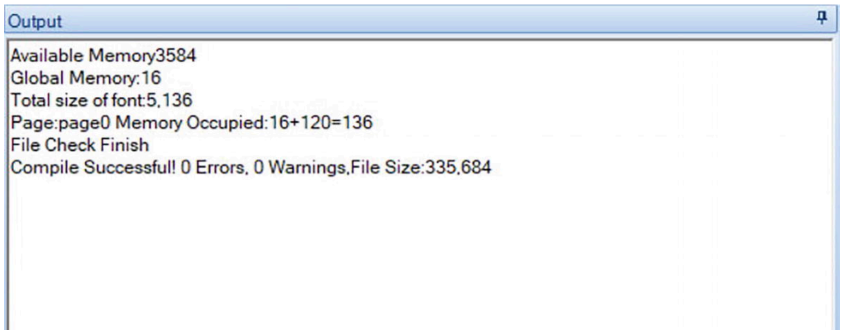


Figure 9-8 Generating a font

Once you followed all the steps correctly, hit the "Compile" button and the project will be compiled correctly, and you will get the output of Figure 9-9.



```
Available Memory3584
Global Memory:16
Total size of font:5,136
Page:page0 Memory Occupied:16+120=136
File Check Finish
Compile Successful! 0 Errors, 0 Warnings,File Size:335,684
```

Figure 9-9 Compiling our application

Read on Microcontroller

Once we have the display set up, we can interface the display to our microcontroller as shown in Listing 9-6.

```
/*
 * File: Main.c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/Int OSC @ 16MHz, 5v
 * Program: 48_Touch_Display
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 * Program Version: 1.0
 *
 *
 * Program Description: This Program Allows PIC16F1719 to communicate
with a NX3224T024_011 2.4 inch Nextion Display. The display communicates
with the microcontroller via UART and  sends messages to the
microcontroller which is displayed on the SSD1306 OLED.
 *
 * Hardware Description: A Nextion 2.4 inch touch screen and SSD1306
OLED is connected to the microcontroller as per header file.
 *
 * Created April 15th, 2017, 9:30 PM
 * Updated Match 28th, 2024, 1:37 AM
 */

/*******************************************************************
 *Includes and defines
 *******************************************************************/

#include "PIC16F1719_Internal.h"
#include "I2C.h"
#include "EUSART.h"
#include "oled.h"
#include <string.h>
#include <stdbool.h>

// buffer for UART
char buf[50];

// Function prototypes
void touchscreen_command(char* string);

void moto_func(char* buf);
void plan_func(char* buf);
void conn_func(char* buf);
void bulb_func(char* buf);

////////////////////////////
// Bool support
///////////////////////////
#define true 1
#define false 0

// boolean for current state
bool on = false;

/*******************************************************************
 * Function: void initMain()
 *
 * Returns: Nothing
 *
 * Description: Contains initializations for main
 *
 * Usage: initMain()
 ************************************************************/

void initMain(){
    // Run at 16 MHz
    internal_16();

    // Setup pins for EUSART
    TRISBbits.TRISB2 = 0;
    ANSELBbits.ANSB2 = 0;

    TRISBbits.TRISB3 = 1;
    ANSELBbits.ANSB3 = 0;

    // Setup pins for I2C
    ANSELCbits.ANSC4 = 0;
    ANSELCbits.ANSC5 = 0;

    TRISCbits.TRISC4 = 1;
    TRISCbits.TRISC5 = 1;

    ///////////////////////
    // Setup Serial Comms
    ///////////////////////
    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    PPSLOCKbits.PPSLOCKED = 0x00; // unlock PPS

    RC4PPSbits.RC4PPS = 0x0011;   //RC4->MSSP:SDA;
    SSPDATPPSbits.SSPDATPPS = 0x0014;   //RC4->MSSP:SDA;
    SSPCLKPPSbits.SSPCLKPPS = 0x0015;   //RC5->MSSP:SCL;
    RC5PPSbits.RC5PPS = 0x0010;   //RC5->MSSP:SCL;

    RB2PPSbits.RB2PPS = 0x14;   //RB2->EUSART:TX;
    RXPPSbits.RXPPS = 0x0B;   //RB3->EUSART:RX;

    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    PPSLOCKbits.PPSLOCKED = 0x01; // lock PPS
}

/*************************************************************
 * Function: Main
 *
 * Returns: Nothing
 *
 * Description: Program entry point
 *************************************************************/

void main(void) {
    initMain();

     // Initialize I2C
    I2C_Init();
    __delay_ms(500);
    // Initialize OLED
    OLED_Init();

    // clear OLED
    OLED_Clear();

    __delay_ms(1000);

    // Initialize EUSART module with 9600 baud
    EUSART_Initialize(9600);
    __delay_ms(2000);

    // Dim Touchscreen
    OLED_YX(0, 0);
    OLED_Write_String("Dim Screen");

    touchscreen_command("dim=30");

    // Update Touchscreen
    OLED_YX(0, 0);
    OLED_Write_String("Update Screen");

    touchscreen_command("t3.txt=\"16\"");

    OLED_Clear();

    while(1){

        // Read EUSART
        EUSART_Read_Text(buf, 11);

        // Check for which checkbox triggered
        bulb_func(buf);
        moto_func(buf);
        plan_func(buf);
        conn_func(buf);
    }

    return;
}

/*
 Send commands to Touchscreen
 */
void touchscreen_command(char* string)
{
    EUSART_Write_Text(string);
    EUSART_Write(0xFF);
    EUSART_Write(0xFF);
    EUSART_Write(0xFF);
    __delay_ms(1000);
}

/*
 Bulb Function Routines
 */
void bulb_func(char* buf)
{
   char* bulb1;

   bulb1 = strstr(buf, "bulb");

   if (bulb1 == NULL)
   {
       return;
   }

   else
   {
       if (!on){
       OLED_YX(0, 0);
       OLED_Write_String("Bulb On");
       __delay_ms(1000);
       OLED_Clear();

        on = true;
       }

       else {
           OLED_YX(0, 0);
           OLED_Write_String("Bulb Off");
           __delay_ms(1000);
           OLED_Clear();

          on = false;
       }
   }
}

/*
 Motor function Routines
 */
void moto_func(char* buf)
{
   char* moto1;

   moto1 = strstr(buf, "moto");

   if (moto1 == NULL)
   {
       return;
   }

   else
   {
       if (!on){
       OLED_YX(0, 0);
       OLED_Write_String("Motor On");
       __delay_ms(1000);
       OLED_Clear();

        on = true;
       }

       else {
           OLED_YX(0, 0);
           OLED_Write_String("Motor Off");
           __delay_ms(1000);
           OLED_Clear();

          on = false;
       }
   }
}

/*
 Plant function routines
 */
void plan_func(char* buf)
{
   char* plant1;

   plant1 = strstr(buf, "plan");

   if (plant1 == NULL)
   {
       return;
   }

   else
   {
       if (!on){
       OLED_YX(0, 0);
       OLED_Write_String("Plant On");
       __delay_ms(1000);
       OLED_Clear();

        on = true;
       }

       else {
           OLED_YX(0, 0);
           OLED_Write_String("Plant Off");
           __delay_ms(1000);
           OLED_Clear();

          on = false;
       }
   }
}

/*
 Connection Function Routines
 */
void conn_func(char* buf)
{
   char* conn1;

   conn1 = strstr(buf, "conn");

   if (conn1 == NULL)
   {
       return;
   }

   else
   {
       if (!on){
       OLED_YX(0, 0);
       OLED_Write_String("Connected");
       __delay_ms(1000);
       OLED_Clear();

        on = true;
       }

       else {
           OLED_YX(0, 0);
           OLED_Write_String("Disconnected");
           __delay_ms(1000);
           OLED_Clear();

          on = false;
       }
   }
}
```

Listing 9-6 Interfacing the Touch Display

At this stage in your embedded systems journey, you should be able to recognize what is happening. The UART is used for communication with the Nextion display, while I2C is utilized for interfacing with the OLED display. Once the initial setup is complete, the code initializes the I2C communication, setting up the OLED display by clearing any existing data on the screen. Following a brief delay, the UART module is initialized with a baud rate of 9600, preparing the microcontroller for communication with the Nextion display. The program then sends a command to the touch screen to dim its brightness as an initial demonstration of control.

The core of the program resides in an infinite loop where the microcontroller continuously reads text messages from the Nextion display via UART. Depending on the received messages, which are stored in a buffer, the program triggers specific functions like turning a bulb on/off, controlling a motor, managing a plant's needs, or indicating connection status. These actions are reflected on the OLED display through messages like "Bulb On", "Motor Off", "Plant On", or "Connected", providing visual feedback for the interactions with the touch screen.

Each function dedicated to a specific task (bulb, motor, plant, connection) checks the buffer for relevant keywords. Upon finding a match, it toggles the state of a boolean variable and updates the OLED display accordingly. This boolean serves as a simple way to track the on/off status of devices or connections being controlled, ensuring that the OLED display shows the current status accurately.

Conclusion

In this chapter, we looked at using LCD, OLED, and touch screen displays. Although other types of displays are available, users now expect to be able to touch their displays to interact with them. In addition, OLED technology is rapidly taking the place of traditional LCDs. At this point, I am confident that you can interface your microcontroller to any type of display that will be thrown at you. In the upcoming chapters throughout this book, we'll use these displays in various projects.

A. Subero, *Programming PIC Microcontrollers with XC8*, Maker Innovations Series
https://doi.org/10.1007/979-8-8688-0467-0_10

10. ADC and DAC

Armstrong Subero[1]

(1) Moruga, Trinidad and Tobago

The world that we live in is analog; however, computers operate within the digital domain. This divide creates a need for a field of study known as data conversion. With that being said, welcome to the fascinating realm of analog-to-digital conversion (ADC) and digital-to-analog conversion (DAC). This is where microcontrollers shine as the ability to convert real-world data such as temperature and light intensity into digital signals allows us to analyze and manipulate these signals, effectively allowing us to have digital interaction with our analog world. Smart devices and embedded systems where we take subsequent actions based on data analysis are common. ADC is essential for anyone interested in microcontrollers. In this chapter, we embark on a journey into the fundamental processes that bridge the worlds of analog and digital signals. These conversions, integral to modern electronics, empower devices to interact with and interpret the analog environment around them.

Data Conversion

Data conversation is a crucial process that involves transforming data from one form to another. There is an entire branch of electronics called signal processing that focuses almost exclusively on dealing with this conversion process. This concept as we are considering it is in terms of analog-to-digital conversion (ADC) and digital-to-analog conversion (DAC). When we perform ADC conversion, we are transforming continuous analog signals into discrete digital values. Remember, we live in an analog world, so as long as biological humans exist in a physical world, there will be a need to convert those real-world analog signals into digital ones.

Conversely, once we store those analog signals in digital form, we need some way to convert them back into an analog representation of the original signal, which is where DAC comes in. Think about something like audio playback where digital audio files are converted into analog sound waves for being reproduced through speakers. Both ADC and DAC share common principles in the data conversion process. The accuracy and precision of the conversion are needed to maintain the fidelity of the signal and play a crucial role in maintaining the integrity of the original information.

Regardless of which type of data conversion we are performing, probably one of the most important things we look at is the resolution. Resolution is the name we give to the ability to represent fine details in data. For ADC, this is the number of bits we use to represent our digital value, and for DAC, it is the precision with which our digital value can be converted back into an analog one.

Sampling is another facet we must consider when doing data conversion. ADC sampling involves capturing discrete samples of an analog signal at regular intervals, and in DAC, it encompasses reconstructing a continuous analog signal from discrete digital samples.

Challenges of Data Conversion

Having real-world experience working with data conversion, one thing I would like to share with you is challenges surrounding data conversion. There are challenges in data conversion that are common to both ADC and DAC and require careful consideration when we are doing our designs.

By far, the most important challenge in the data conversation process is noise. Analog signals are susceptible to noise, which can be introduced when we are doing sampling. Noise during ADC in particular can be troublesome as noise can distort the accuracy of the captured digital representation of the analog signal we are trying to process and store. What makes this troublesome is that many times we have to use DAC to reconstruct the original analog signal. If the ADC process is noisy, then unwanted noise can influence the DAC reconstruction, giving us subpar results.

The way we combat noise is with filtering. In both ADC and DAC, using a filter helps to enhance the accuracy and reliability of the conversion processes. However, before we get into filtering, I wanted to cover the concept of the Nyquist frequency. When we sample a signal, something we need to pay attention to is the sampling rate. The sampling rate is the number of samples we take per unit of time from a continuous analog signal to convert it into a discrete signal. We measure this signal in hertz.

The Nyquist frequency is the value of half of the sampling rate. This is important because the maximum frequency that can be accurately represented or reconstructed in a digital system is known as the Nyquist frequency.

The most common filter type we tend to use is an anti-aliasing filter. This is a filter that removes high-frequency components beyond the Nyquist frequency. There are also quantization errors that occur when converting continuous analog signals into discrete digital values. This error is caused due to the difference between the actual analog signal and the closest representable digital value. If we use a limited number of bits, then it can impact the accuracy of the digitized signal. This also affects DACs in the form of precision limitations where a finite number of bits can lead to discrepancies between the actual analog signal and its digital representation.

The final most common challenge I experienced designing real-world devices is in accuracy limitations of the ADC. The accuracy of the ADC is determined by its ability to represent the range and characteristics of an analog signal in digital form. Each ADC module, as we will see later, has a specific number of bits. This leads to a finite ADC resolution, which sets a limit on the accuracy of the digitized signal. Similarly, with a DAC, the accuracy can affect how faithfully the analog signal is reconstructed.

ADC (Analog-to-Digital Conversion)

One of the most important onboard modules of the PIC microcontrollers is the analog-to-digital converter module. If you read the last section, you will realize that analog sensors are responsible for the conversion of real-world sensor data into digital ones. The ADC has a specific number of bits of resolution. These can be from 8-bit to 32-bit. The higher the resolution of the ADC, the greater the number of steps there will be from the minimum voltage.

We can examine how ADC conversion works by looking at one of the most common types of ADC conversion circuits: the successive approximation ADC. The successive approximation ADC uses a multistep binary search algorithm to approximate the input voltage level. While the binary search algorithm works on an array of values, in the context of an ADC circuit, it works by approximating the voltage level of an analog input signal. This voltage level, as we stated, is the digital representation of some analog signal.

The function of the ADC is rather simple if we look at it from an algorithmic perspective, which is to say as a series of steps. If we start with the Most Significant Bit (MSB) and iteratively compare the analog input with a digital-to-analog converted value of the current approximation, we can adjust this approximation based on the comparison outcome. If we successively set or clear bits from the MSB to the Least Significant bit (LSB), the ADC refines its approximation of the input signal. This process iterates until all the bits have been adjusted, giving our final digital representation of the analog input. The reason we say it's like the binary search algorithm is because it mirrors the binary search algorithm's principle of halving the search range with each step.

The way a successive approximation ADC module accomplishes this is with a few hardware circuits as shown in Figure 10-1.

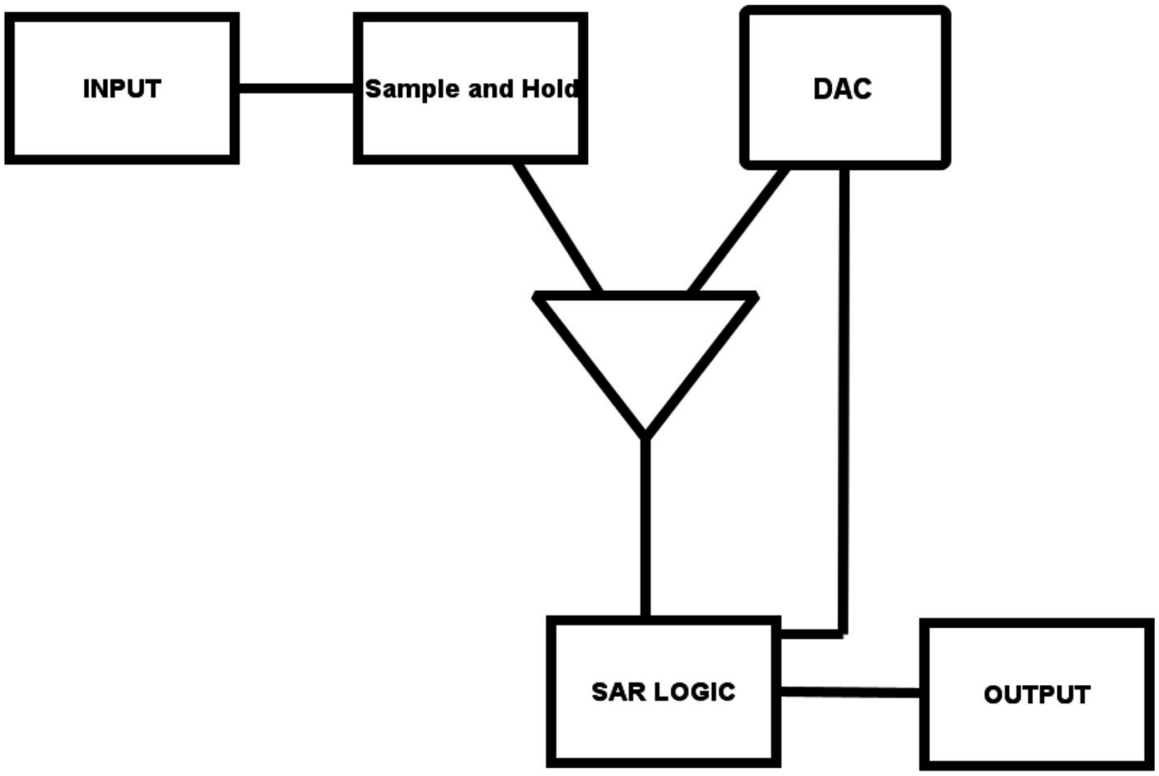


Figure 10-1 Successive approximation ADC block diagram

The first of these is the sample and hold circuit. The sample and hold circuit captures and holds the analog input voltage at a constant level during the conversion process. The Successive Approximation Register (SAR) is a register that holds the approximation of the digital value. This is the part of the converter that starts with the most significant bit and works its way down to the least significant bit. There is also a DAC that converts the output of the SAR to an analog voltage for comparison with the input signal. The comparator compares the analog input voltage with the output of the DAC. The comparator's output determines whether the SAR will set or clear the next bit. There is also the control logic (shown as integrated with the SAR) that manages the operation sequence of the SAR, DAC, and sample and hold circuit. That conversion process follows the successive approximation algorithm.

The PIC16F1719 has a 10-bit-resolution successive approximation ADC; what this means is that the ADC can read a voltage in steps from 0 to 1023, which is 1024 distinct steps.

When talking about ADC conversion, something that always comes up is the sampling rate. To understand the sampling rate, we need to discuss sampling in general. Sampling is the name we give to the process of periodically measuring the amplitude of an analog signal and converting it to digital values for processing. Generally speaking, the more samples we capture from the ADC, the more accurately we can represent the signal.

If the sampling rate is too low, we will not have enough data to accurately represent the signal, and if the sampling rate is too high, then we may take too long to have real-time performance; in addition, we may waste valuable computing and power in trying to achieve an unnecessarily high sample rate. To solve this problem, a special sampling rate known as the Nyquist sampling rate is used as a way of giving us the minimum sampling rate. The Nyquist sampling rate is stated as having the minimum sampling rate being at least twice the highest sampling rate. So if we have a signal that is expected to have a maximum frequency of 5 kHz, then we need to have a minimum sampling rate of 10 kHz.

It is also common to see the sampling rate of the ADC being given in samples per second. This is usually kilo samples per second (kSPS) and mega samples per second (MSPS). Usually, kilo samples per second are the most common frequency encountered. For example, a 48 kSPS sampling rate means the ADC can sample 48,000 times a second. This sampling rate is used in audio processing. For most purposes though, we usually use a lesser sampling rate.

Reading a Potentiometer

The first analog device we will look at reading with the ADC module is the potentiometer. The potentiometer is used in devices such as volume controls, light dimmers, tuning and calibration in electronic circuits, and general control knobs in electronics. You would use them to adjust the volume on audio devices such as stereos and amplifiers, just as you would use them to adjust the brightness or contrast of a monitor or control temperature settings. These applications make the potentiometer a versatile device, and it is relatively easy to interface to a microcontroller. We see how we can connect the potentiometer to the microcontroller in Figure 10-2.

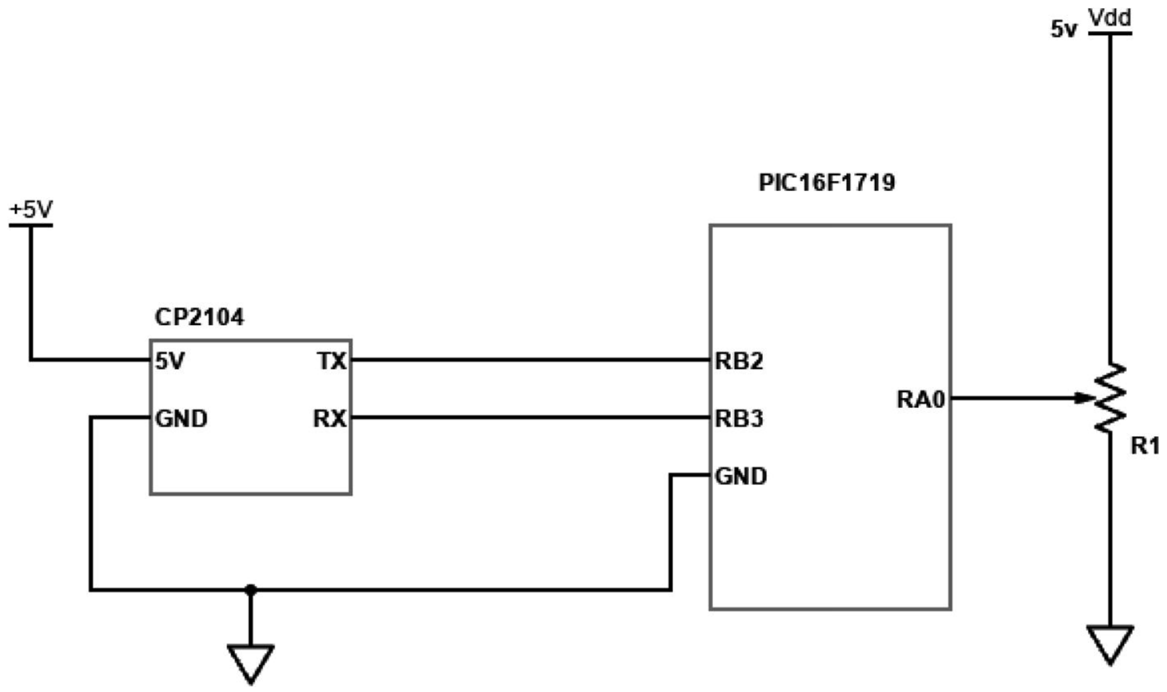


Figure 10-2 Reading a potentiometer

Once we have the potentiometer connected, we can write our program. The program is given in Listing 10-1.

```
/*
 * File: main.c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/int OSC @ 32MHz, 5v
 * Program: 21_ADC
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 * Program Version: 1.2
 *                  * Updated from PIC16F1717 to PIC16F1719
 *                  * Updated clock speed to 32 MHz
 *                  * Added PLL stabilization
 *
 * Program Description: This program allows the PIC16F1719 to
use the ADC module
 *
 * Hardware Description: PIN RB2 of a the PIC16F1719 MCU is
connected to a PL2303XX USB to UART converter cable and a 10k
potentiometer is connected to pin RA0
 *
 * Created November 7th, 2016, 7:05 PM
 * Last Updated: November 24th, 2023, 4:58 AM
 */

/*************************************************************
 *Includes and defines
 ************************************************************/

#include "PIC16F1719_Internal.h"
#include "EUSART.h"

/*************************************************************
 * Function: void initMain()
 *
 * Returns: Nothing
 *
 * Description: Contains initializations for main
 *
 * Usage: initMain()
 *************************************************************/

void initMain(){

    // Run at 32 MHz
    internal_32();

    // Allow PLL startup time ~2 ms
    __delay_ms(10);
```

```
// Set PIN D1 as output
TRISDbits.TRISD1 = 0;

// Turn off LED
LATDbits.LATD1 = 0;

// Setup PORTD
TRISD = 0;
ANSELD = 0;

TRISBbits.TRISB2 = 0;
ANSELBbits.ANSB2 = 0;

/////////////////////
// Setup EUSART
////////////////////
PPSLOCK = 0x55;
PPSLOCK = 0xAA;
PPSLOCKbits.PPSLOCKED = 0x00; // unlock PPS

RB2PPSbits.RB2PPS = 0x14;   //RB2->EUSART:TX;
RXPPSbits.RXPPS = 0x0B;   //RB3->EUSART:RX;

PPSLOCK = 0x55;
PPSLOCK = 0xAA;
PPSLOCKbits.PPSLOCKED = 0x01; // lock PPS

////////////////////
// Configure ADC
///////////////////

// Set A0 as input
TRISAbits.TRISA0 = 1;

// Set A0 as analog
ANSELAbits.ANSA0 = 1;

// Fosc/32 ADC conversion time is 1.0 us
ADCON1bits.ADCS = 0b010;

// Right justified
ADCON1bits.ADFM = 1;

// Vref- is Vss
ADCON1bits.ADNREF = 0;

// Vref+ is Vdd
```

```
    ADCON1bits.ADPREF = 0b00;

    // Set input channel to AN0
    ADCON0bits.CHS = 0b00000;

    // Zero ADRESL and ADRESH
    ADRESL = 0;
    ADRESH = 0;
}

/***********************************************************
 * Function: Main
 *
 * Returns: Nothing
 *
 * Description: Program entry point
 *******************************************************/

void main(void) {
    initMain();

    // Initialize EUSART module with 19200 baud
    EUSART_Initialize(19200);

    // variable to store conversion result
    int result;

    while(1)
    {
      // Turn ADC on
      ADCON0bits.ADON = 1;

      // Sample CH0
      __delay_us(10);
      ADCON0bits.GO = 1;
      while (ADCON0bits.GO_nDONE);

      // Store ADC result
      result = ((ADRESH<<8)+ADRESL);

      EUSART_Write_Text(", Read Data: ");
      EUSART_Write_Integer(result);
      EUSART_Write_Text("\n");
      __delay_ms(1000);
    }

    return;
```

```
}
```

Listing 10-1 Program for Reading a Potentiometer

Within our program, the main section we have to focus on is the ADC configuration section. In the ADC configuration section, pin RA0 is set as an analog input to connect with the potentiometer. The ADC is configured for a conversion clock of Fosc/32, right-justified result format, Vss as the negative voltage reference (Vref-), and Vdd as the positive voltage reference (Vref+). The input channel is set to AN0, corresponding to RA0, and the ADC result registers (ADRESH and ADRESL) are cleared.

The main function initializes the system and the EUSART module with a baud rate of 19200. Inside an infinite loop, the ADC is turned on, and the program waits for a short delay before starting the conversion on channel 0. It then waits for the conversion to complete, combines the high and low bytes of the ADC result, and sends the result over the UART connection every second, formatted as a readable text string indicating the read data value.

Light Sensing with a Photoresistor

While the potentiometer is a great device for exploring the use of the ADC module, we can also look at the ADC module in terms of using a light sensor. The light sensor we can use is the photoresistor. The photoresistor is a kind of resistor that feels the light much like we feel the warmth of the sun on our skin. Unlike regular resistors that have a fixed value, a photoresistor changes its resistance based on the light it receives. In the shadows, it's as if it's asleep, showing high resistance, but when light shines upon it, it wakes up, lowering its resistance and allowing more electrical current to pass through. Using this behavior, we can perform very rudimentary light sensing.

The photoresistor has a lot of uses; for example, it can tell street lights when it's time to illuminate the roads as dusk falls or wake up your phone screen when you uncover it. Its simplicity is its strength, requiring no power to operate; it merely sits, waiting for light to dictate its resistance. Despite its straightforward operation, the photoresistor is a fundamental building block in creating interactive gadgets and systems, serving as the eyes for many electronic circuits, enabling them to perceive and react to the brightness of their surroundings. We can use the photoresistor with the PIC microcontroller by connecting it as shown in Figure 10-3.

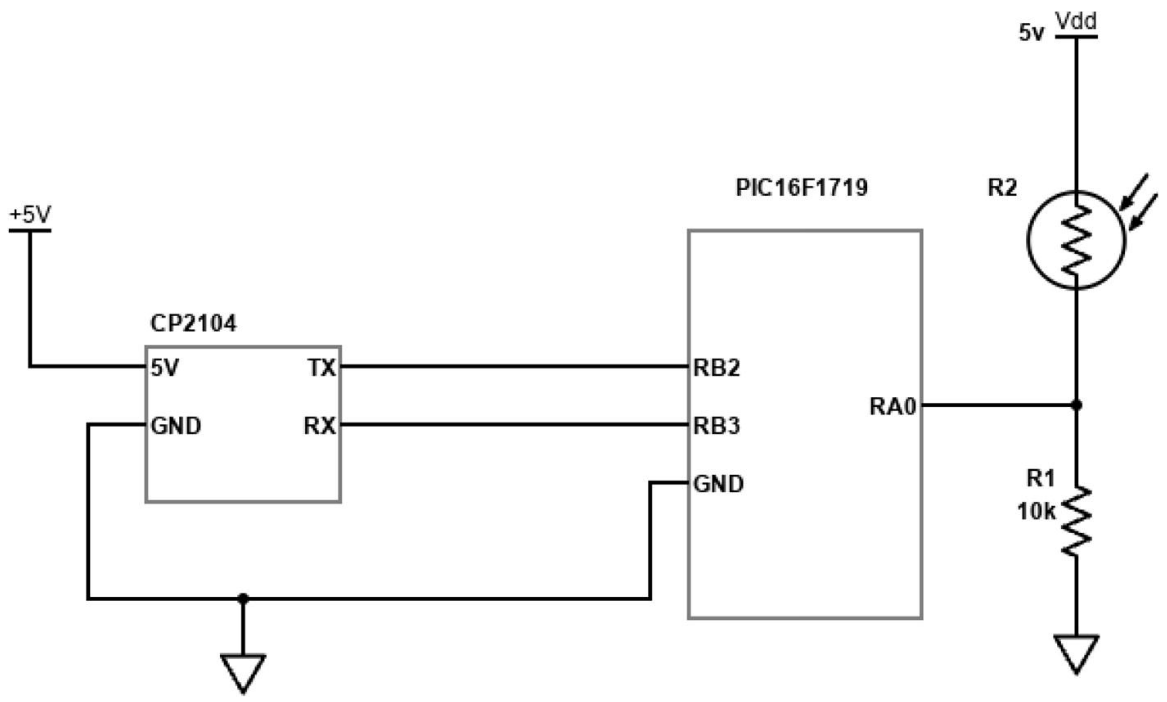


Figure 10-3 Reading a photoresistor

The photoresistor circuit requires a 10k resistor to be used along with the photoresistor. The value of the photoresistor itself is not important as for simple light/no light detection, any value can work. The program to use the photoresistor is given as Listing 10-2. Since the setup is identical to using the potentiometer, we shall focus solely on the main function.

```
/************************************************************
 * Function: Main
 *
 * Returns: Nothing
 *
 * Description: Program entry point
 ***********************************************************/

void main(void) {
    initMain();

    // Initialize EUSART module with 19200 baud
    EUSART_Initialize(19200);

    // variable to store conversion result
    int result;

    while(1)
    {
      // Turn ADC on
```

```
        ADCON0bits.ADON = 1;

        // Sample CH0
        __delay_us(10);
        ADCON0bits.GO = 1;
        while (ADCON0bits.GO_nDONE);

        // Store ADC result
        result = ((ADRESH<<8)+ADRESL);

        if (result < 100)
        {
            EUSART_Write_Text("Dark");
            EUSART_Write_Text("\n");
        }

        else
        {
            EUSART_Write_Text("Light");
            EUSART_Write_Text("\n");
        }

        __delay_ms(1000);
    }

    return;
}
```

Listing 10-2 Light Detection with a Photoresistor

At this point in your journey, you should be comfortable with this program. At its core, the program enters an infinite loop, continually monitoring light levels through the ADC channel. Once we get the value from the ADC converter, the program then makes a simple decision based on the value of the result: if the value is below 100, it interprets this as a “Dark” condition and sends this text over the serial connection. If the value is 100 or above, it deems the environment “Light” and sends this text instead. Using this program, we can have a rudimentary light detection circuit.

Using an Analog Joystick

The next device we will look at controlling is the analog joystick. An analog joystick is a control device commonly used in gaming consoles, industrial controls, and various electronic projects. Anyone who played with a video game console will be well familiar with how this device operates. The analog joystick can be thought of as two potentiometers in a single package.

At its heart, the joystick pivots on a central point, allowing movement in two dimensions, typically forward and backward, left and right. This movement is translated into electrical signals through potentiometers aligned with each axis. As the stick moves away from its neutral position, the resistance changes proportionally, allowing the device to capture the precise angle and magnitude of the stick's movement. What this means is that to capture movement from the joystick, we need to use multiple channels of the microcontroller ADC. In our case, we will use the microcontroller to read the joystick performing four basic actions, detecting when the joystick is moved left, right, up, or down. We see the program for using an analog joystick in Listing 10-3.

```
/*
 * File: main.c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/int OSC @ 32MHz, 5v
 * Program: 37_Analog_Joystick
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 * Program Version: 1.2
 *                   * Updated from PIC16F1717 to PIC16F1719
 *                   * Updated clock speed to 32 MHz
 *                   * Added PLL stabilization
 *
 * Program Description: This program allows the PIC16F1719 to
use the ADC module to read an analog joystick
 *
 * Hardware Description: PIN RB2 of a the PIC16F1719 MCU is
connected to a PL2303XX USB to UART converter cable and a
analog joystick to AN12 (RB0) and AN10 (RB1)
 *
 * Created November 7th, 2016, 7:05 PM
 * Last Updated: MArch 25th, 2024, 3:34 AM
 */

/*************************************************************
 *Includes and defines
 ************************************************************/

#include "PIC16F1719_Internal.h"
#include "EUSART.h"

/**************************************************************
 * Function: void initMain()
 *
 * Returns: Nothing
 *
 * Description: Contains initializations for main
 *
 * Usage: initMain()
 ***********************************************************/
```

```
void initMain(){

    // Run at 32 MHz
    internal_32();

    // Allow PLL startup time ~2 ms
    __delay_ms(10);

    // configure analog input pins
    TRISBbits.TRISB0 = 1;
    ANSELBbits.ANSB0 = 1;

    TRISBbits.TRISB1 = 1;
    ANSELBbits.ANSB1 = 1;

    /////////////////////
    // Setup EUSART
    ////////////////////
    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    PPSLOCKbits.PPSLOCKED = 0x00; // unlock PPS

    RB2PPSbits.RB2PPS = 0x14;   //RB2->EUSART:TX;
    RXPPSbits.RXPPS = 0x0B;   //RB3->EUSART:RX;

    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    PPSLOCKbits.PPSLOCKED = 0x01; // lock PPS

    ////////////////////
    // Configure ADC
    ///////////////////

    // Set A0 as input
    TRISAbits.TRISA0 = 1;

    // Set A0 as analog
    ANSELAbits.ANSA0 = 1;

    // Fosc/32 ADC conversion time is 1.0 us
    ADCON1bits.ADCS = 0b010;

    // Right justified
    ADCON1bits.ADFM = 1;

    // Vref- is Vss
    ADCON1bits.ADNREF = 0;
    // Vref+ is Vdd
    ADCON1bits.ADPREF = 0b00;

    // Zero ADRESL and ADRESH
    ADRESL = 0;
    ADRESH = 0;
}

/*************************************************************
 * Function: Main
 *
 * Returns: Nothing
 *
 * Description: Program entry point
 ************************************************************/

void main(void) {
    initMain();

    // Initialize EUSART module with 19200 baud
    EUSART_Initialize(19200);

    // variables to store conversion result
    int result;
    int result1;

    while(1)
    {
     // Turn ADC on
      ADCON0bits.ADON = 1;

     // Set input channel to AN12
     ADCON0bits.CHS = 0b01100;
      // Sample CH0
      __delay_us(10);
      ADCON0bits.GO = 1;
      while (ADCON0bits.GO_nDONE);

      // Store ADC result
      result = ((ADRESH<<8)+ADRESL);

     // Set input channel to AN10
     ADCON0bits.CHS = 0b01010;
      // Sample CH0
      __delay_us(10);
      ADCON0bits.GO = 1;
      while (ADCON0bits.GO_nDONE);
```

```
        // Store ADC result
        result1 = ((ADRESH<<8)+ADRESL);

        if (result == 0)
        {
            EUSART_Write_Text("Up");
            EUSART_Write_Text("\n");
        }

        else if (result == 1023)
        {
            EUSART_Write_Text("Down");
            EUSART_Write_Text("\n");
        }

        if (result1 == 0)
        {
           EUSART_Write_Text("Right");
           EUSART_Write_Text("\n");

        }

        else if(result1 == 1023)
        {
           EUSART_Write_Text("Left");
           EUSART_Write_Text("\n");
        }

        // Update every second
        __delay_ms(1000);
    }

    return;
}
```

Listing 10-3 Using an Analog Joystick

The main function continuously reads the joystick's position by alternating the ADC input channel between AN12 and AN10 to measure the vertical and horizontal positions, respectively. After initiating a conversion and waiting for its completion, the ADC result is stored in variables (result for the vertical axis and result1 for the horizontal axis). Based on these results, the program determines the joystick's direction: "Up" or "Down" for vertical movements and "Right" or "Left" for horizontal movements. This determination is based on the ADC values, where a result of 0 indicates one extreme direction (Up or Right) and a result of 1023 indicates the opposite extreme (Down or Left). After detecting the direction, the program sends the corresponding direction as text over the UART connection, allowing for remote monitoring and interaction. The process repeats every second, providing continuous feedback on the joystick’s position.

DAC (Digital-to-Analog Converter)

Another microcontroller peripheral we can look at that complements the ADC is the DAC. The DAC is the brother of the ADC except instead of converting an analog signal into a digital one, it converts a digital signal to an analog one. The PIC16F1719 device we are using has two onboard DACs including an 8-bit DAC and a 5-bit DAC.

Anyone who has been involved in electronics for a while can appreciate the value of having a DAC onboard the PIC microcontroller. Before having a DAC onboard a specialized IC like the XR2206, for example, would be the choice for waveform generation. When building circuits, the majority of use we have for a low-resolution DAC is in waveform generation. For our example, we will use the 8-bit DAC to output waveforms on pin RA2, which we will view on the oscilloscope. We will generate all the common waveforms including ramp, sine, square, and triangle waveforms.

The first waveform we can generate is the ramp waveform given in Listing 10-4.

```
/*
 * File: main.c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/int OSC @ 32MHz, 5v
 * Program: 12_DAC
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 * Program Version: 1.2
 *
 * Program Description: This Program Allows PIC16F1717 DAC1 to
generate a waveform on PIN RA2
 *
 * Hardware Description: An Oscilloscope probe is connected to
pin RA2
 *
 * Created November 4th, 2016, 1:00 PM
 * Last Updated: November 18th, 2023, 2:51 AM
 */

/*************************************************************
 *Includes and defines
 ************************************************************/
#include "PIC16F1719_Internal.h"

/************************************************************
 * Function: void initMain()
 *
 * Returns: Nothing
 *
 * Description: Contains initializations for main
 *
 * Usage: initMain()
 ***************************************************************/

void initMain(){
    // Run at 32 MHz
    internal_32();

    // Set PIN D1 as output
    TRISDbits.TRISD1 = 0;
    TRISDbits.TRISD2 = 0;

    // Turn off LED
    LATDbits.LATD1 = 0;
    LATDbits.LATD1 = 0;

    //////////////////////
    // Configure DAC
    /////////////////////

    // DAC enabled
    DAC1CON0bits.DAC1EN = 1;

    // DACOUT pin enabled
    DAC1CON0bits.DAC1OE1 = 1;

    // +ve source is Vdd
    DAC1CON0bits.DAC1PSS = 0;

    // -ve source is Vss
    DAC1CON0bits.DAC1NSS = 0;

    // Initial output is 0v
    DAC1CON1bits.DAC1R = 0;
}

/**************************************************************
 * Function: Main
 *
 * Returns: Nothing
 *
 * Description: Program entry point
 *************************************************************/

void main(void) {
    initMain();
    while(1){
        DAC1CON1++;
    }

    return;
}
```

Listing 10-4 Generating a Ramp Waveform

Sinusoidal Waveform Generation

We can also use the DAC to generate a sine wave as given in Listing 10-5. To generate a sinusoidal waveform, you'll need a lookup table that contains the values of a sine wave at various points along its period. You then iterate through this table, setting the DAC output to each value in turn, looping back to the start of the table once you reach the end. We see the program for generating a sine wave in Listing 10-5.

```
/*
 * File: main.c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/int OSC @ 32MHz, 5v
 * Program: 38_Sine_Wave_Generation
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 * Program Version: 1.2
 *
 * Program Description: This Program Allows PIC16F1719 DAC1 to
generate a sine waveform on PIN RA2
 *
 * Hardware Description: An Oscilloscope probe is connected to
pin RA2
 *
 * Created November 4th, 2016, 1:00 PM
 * Last Updated: March 25th, 2024, 2:46 AM
 */

/***********************************************************
 *Includes and defines
 ***********************************************************/
#include "PIC16F1719_Internal.h"

#define TABLE_SIZE  256
unsigned char sineTable[TABLE_SIZE];

/***********************************************************
 * Function: void generateSineTable
 *
 * Returns: Nothing
 *
 * Description: Generates a sine table
 *
 * Usage: generateSineTable()
 **********************************************************/
void generateSineTable()
{
    for(int i = 0; i < TABLE_SIZE; i++)
    {
        // Scale to DAC range
        sineTable[i] = (sin(i * 2 * M_PI / TABLE_SIZE) + 1) *
127.5;
    }
}

/***********************************************************
 * Function: void initMain()
 *
 * Returns: Nothing
 *
 * Description: Contains initializations for main
 *
 * Usage: initMain()
 **********************************************************/

void initMain(){
    // Run at 32 MHz
    internal_32();
    __delay_ms(2);

    // Set PIN D1 as output
    TRISDbits.TRISD1 = 0;
    TRISDbits.TRISD2 = 0;

    // Turn off LED
    LATDbits.LATD1 = 0;
    LATDbits.LATD1 = 0;

    //////////////////////
    // Configure DAC
    /////////////////////

    // DAC enabled
    DAC1CON0bits.DAC1EN = 1;

    // DACOUT pin enabled
    DAC1CON0bits.DAC1OE1 = 1;

    // +ve source is Vdd
    DAC1CON0bits.DAC1PSS = 0;

    // -ve source is Vss
    DAC1CON0bits.DAC1NSS = 0;

    // Initial output is 0v
    DAC1CON1bits.DAC1R = 0;
}

/**************************************************************
 * Function: Main
 *
 * Returns: Nothing
 *
 * Description: Program entry point
 **************************************************************/

void main(void) {
    initMain();
    generateSineTable();

    int index = 0;
    while(1)
    {
        DAC1CON1 = sineTable[index++];

        if(index >= TABLE_SIZE)
        {
            index = 0;
            __delay_us(50); // Adjust for waveform frequency
        }
    }

    return;

}
```

Listing 10-5 Generating a Sine Wave

The ingenious part of the program lies in the generateSineTable function. This function populates an array, sineTable, with values representing a sine wave. By iterating through the array and adjusting each value to fit within the DAC's output range, it effectively creates a digital representation of a sine wave. The values are calculated based on the sine function, scaled, and offset to match the 0−255 range of the DAC, enabling the generation of an analog sine wave upon output.

The main function kickstarts the system by initializing the hardware and generating the sine table. Then, within an infinite loop, it cyclically outputs values from the sineTable to DAC1, thereby translating digital signals into a continuous analog sine wave observable on PIN RA2. The index variable iterates through the sine table, and once it reaches the end, it resets, ensuring an endless loop of sine wave generation. A delay is introduced to control the frequency of the waveform, allowing for adjustments to how fast or slow the sine wave cycles.

Square Wave Generation

Another simple waveform we can generate using the DAC is a square wave. Generating a square waveform involves toggling the DAC output between its minimum and maximum values at regular intervals as given in Listing 10-6.

```
/*
 * File: main.c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/int OSC @ 32MHz, 5v
 * Program: 39_Square_Wave_Generation
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 * Program Version: 1.2
 *
 * Program Description: This Program Allows PIC16F1719 DAC1 to
generate a square waveform on PIN RA2
 *
 * Hardware Description: An Oscilloscope probe is connected to
pin RA2
 *
 * Created November 4th, 2016, 1:00 PM
 * Last Updated: March 25th, 2024, 3:03 AM
 */

/*************************************************************
 *Includes and defines
 ************************************************************/
#include "PIC16F1719_Internal.h"

#define MAX_DAC_VALUE  255 // Max DAC value for PIC16F1719
#define MIN_DAC_VALUE  0   // Min DAC value

/************************************************************
 * Function: void initMain()
 *
 * Returns: Nothing
 *
 * Description: Contains initializations for main
 *
 * Usage: initMain()
 ***********************************************************/

void initMain(){
    // Run at 32 MHz
    internal_32();
    __delay_ms(2);

    // Set PIN D1 as output
    TRISDbits.TRISD1 = 0;
    TRISDbits.TRISD2 = 0;

    // Turn off LED
    LATDbits.LATD1 = 0;
    LATDbits.LATD1 = 0;

    //////////////////////
    // Configure DAC
    /////////////////////

    // DAC enabled
    DAC1CON0bits.DAC1EN = 1;

    // DACOUT pin enabled
    DAC1CON0bits.DAC1OE1 = 1;

    // +ve source is Vdd
    DAC1CON0bits.DAC1PSS = 0;

    // -ve source is Vss
    DAC1CON0bits.DAC1NSS = 0;

    // Initial output is 0v
    DAC1CON1bits.DAC1R = 0;
}

/***********************************************************
 * Function: Main
 *
 * Returns: Nothing
 *
 * Description: Program entry point
 **********************************************************/

void main(void) {
    initMain();
    while(1)
    {
       DAC1CON1 = MAX_DAC_VALUE;
       __delay_ms(1); // High duration
       DAC1CON1 = MIN_DAC_VALUE;
       __delay_ms(1); // Low duration
    }

    return;
}
```

Listing 10-6 Generating a Square Wave

The essence of our program unfolds within the main function, where an infinite loop perpetuates the generation of the square wave. This is achieved by setting the DAC output to its maximum value, corresponding to the highest voltage level, followed by a brief delay to determine the duration of the high state. Subsequently, the DAC output is adjusted to its minimum value, representing the lowest voltage level, with another delay to define the low state's duration. These alternating high and low states, coupled with the precise timing between transitions, produce a square wave observable on PIN RA2.

Triangle Wave Generation

Finally, the last waveform we will look at is the triangle wave. A triangular waveform can be generated by linearly incrementing the DAC output until it reaches a maximum value and then linearly decrementing it back to the minimum value. The program to generate a triangle wave is given in Listing 10-7.

```
/*
 * File: main.c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/int OSC @ 32MHz, 5v
 * Program: 40_Triangle_Wave_Generation
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 * Program Version: 1.2
 *
 * Program Description: This Program Allows PIC16F1719 DAC1 to
generate a triangle waveform on PIN RA2
 *
 * Hardware Description: An Oscilloscope probe is connected to
pin RA2
 *
 * Created November 4th, 2016, 1:00 PM
 * Last Updated: March 25th, 2024, 3:13 AM
 */

/*************************************************************
 *Includes and defines
 **********************************************************/
#include "PIC16F1719_Internal.h"

#define MAX_DAC_VALUE  255 // Max DAC value for PIC16F1719
#define MIN_DAC_VALUE  0   // Min DAC value

/**********************************************************
 * Function: void initMain()
 *
 * Returns: Nothing
 *
 * Description: Contains initializations for main
 *
 * Usage: initMain()
 **********************************************************/

void initMain(){
    // Run at 32 MHz
    internal_32();
    __delay_ms(2);

    // Set PIN D1 as output
    TRISDbits.TRISD1 = 0;
    TRISDbits.TRISD2 = 0;

    // Turn off LED
    LATDbits.LATD1 = 0;
    LATDbits.LATD1 = 0;

    //////////////////////
    // Configure DAC
    /////////////////////

    // DAC enabled
    DAC1CON0bits.DAC1EN = 1;

    // DACOUT pin enabled
    DAC1CON0bits.DAC1OE1 = 1;

    // +ve source is Vdd
    DAC1CON0bits.DAC1PSS = 0;

    // -ve source is Vss
    DAC1CON0bits.DAC1NSS = 0;

    // Initial output is 0v
    DAC1CON1bits.DAC1R = 0;
}

 /**********************************************************
 * Function: Main
 *
 * Returns: Nothing
 *
 * Description: Program entry point
 **********************************************************/

void main(void) {
    initMain();

    int value = MIN_DAC_VALUE;
    int direction = 1; // Start by incrementing

    while(1) {
        DAC1CON1 = value;
        value += direction;
        if(value >= MAX_DAC_VALUE || value <= MIN_DAC_VALUE) {
            direction = -direction; // Change direction
        }
        __delay_us(10); // Adjust for waveform frequency and
slope
    }
    return;
}
```

Listing 10-7 Generating a Triangle Wave

If we look at our program, we see that the program dynamically adjusts the DAC1 output to create the characteristic linear rise and fall of a triangle waveform. This is achieved by incrementally adjusting the DAC1 output value stored in value, initially starting at the minimum DAC value (MIN_DAC_VALUE).

A key variable, direction, is used to control whether the output value should be incremented or decremented, starting with a value of 1 to indicate an upward slope. As the output value alters with each loop iteration, the program checks if the value has reached either the maximum (MAX_DAC_VALUE) or minimum DAC value, indicating the need to reverse the direction of the waveform's slope. This reversal is managed by flipping the sign of direction, thus changing the incrementation into decrementation and vice versa.

The loop includes a microsecond-level delay (__delay_us(10)) to control the frequency and slope of the triangle waveform. By adjusting this delay, the speed of the waveform's rise and fall can be finely tuned to the desired specifications. This delay, alongside the dynamic adjustment of the DAC output, allows the program to smoothly transition the output voltage up and down, thereby generating a continuous triangle waveform observable on an oscilloscope.

Conclusion

In this chapter, we covered analog-to-digital conversion and looked at how we can perform analog-to-digital conversion on PIC microcontrollers. After learning how ADC works, we looked at using the ADC module to perform various tasks including reading a potentiometer, reading a photoresistor, and interacting with an analog joystick. We then moved on to the configuration of the DAC module onboard the microcontroller with a focus on waveform generation, and we learned how we can generate several waveforms including ramp, sine, square, and triangle. After working through the examples in this chapter, you will be well equipped to use the ADC and DAC module in your projects.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2024
A. Subero, *Programming PIC Microcontrollers with XC8*, Maker Innovations Series
https://doi.org/10.1007/979-8-8688-0467-0_11

11. CLC, NCO, Comparator, and FVR

Armstrong Subero[1]

(1) Moruga, Trinidad and Tobago

In the grand scheme of things, where the tangible meets the digital, there's a fascinating interplay at work, one that microcontrollers masterfully navigate, connecting the digital and analog worlds. While modules like the DAC and ADC help a lot with that, we can get greater connectivity, thanks to some pretty clever features like Configurable Logic Cells (CLC), Numerically Controlled Oscillators (NCO), comparators, and Fixed Voltage References (FVR). This chapter is more than just a deep dive into technical terms; it's a journey into understanding how these components give microcontrollers their ability to make smart decisions based on the analog signals that define our daily experiences.

Core Independent Peripherals (CIPs)

As we hinted at throughout the book, core independent peripherals (CIPs) in PIC microcontrollers represent a revolutionary step in microcontroller design, enabling robust hardware-based operation with minimal CPU intervention. In this chapter, we take a look at some of these core independent peripherals. These peripherals are designed to handle their tasks independently, freeing the CPU to execute the main application code or to enter a power-saving sleep mode while the peripherals continue to operate. This approach significantly enhances the efficiency and performance of PIC microcontrollers in real-time applications. CIPs cover many functionalities, including timing, communication, and control systems, allowing for complex operations like sensor reading, motor control, and communication protocols to be offloaded from the CPU. By reducing the workload on the CPU and decreasing power consumption, CIPs enable the development of more efficient, reliable, and versatile embedded systems, perfectly aligning with the demands of modern electronics and microcontroller technology.

Configurable Logic Cell (CLC)

For years, the peripherals that were available on microcontrollers have been fairly standard. One of the features that sets PIC microcontrollers apart from other devices is the addition of the Configurable Logic Cell (CLC) module. CLCs are like the microcontroller's Swiss Army knife, adapting to various needs with a simple software configuration. The CLC module has basic gates, flip flops, and latches that you can configure. Using the CLC eliminates the need for external glue logic in most cases. With the CLC, you can combine logic, both internal and external, to the chip to produce a particular function. You can even use the CLC to wake the microcontroller from sleep mode.

The CLC module is tightly integrated with the MPLAB Code Configurator (MCC) plug-in for the MPLAB X IDE. To keep our use case simple, we will use MCC to configure the CLC as a 4-input AND gate. The inputs of the CLC are connected to switches, and if any of the switches are pressed, the respective input of the AND gate becomes a logical low and the LED is switched off. We begin by selecting the MCC module in the toolbar as shown in Figure 11-1.

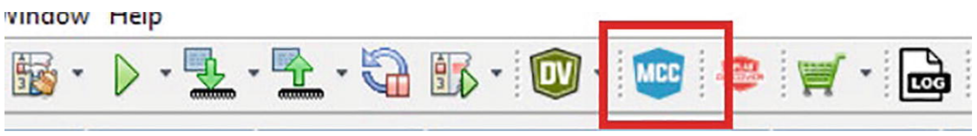
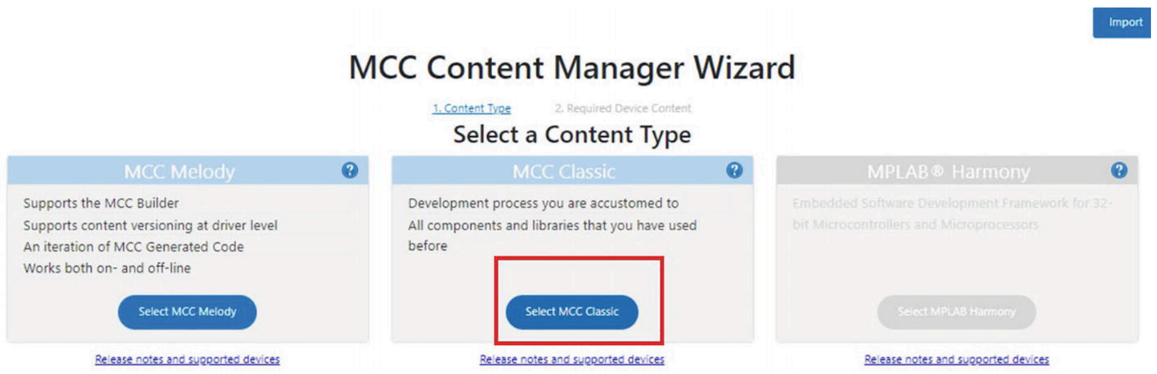


Figure 11-1 The MCC button

Once you select the MCC button, you should see the MCC content manager wizard as shown in Figure 11-2.


Import

MCC Content Manager Wizard

1. Content Type 2. Required Device Content

Select a Content Type

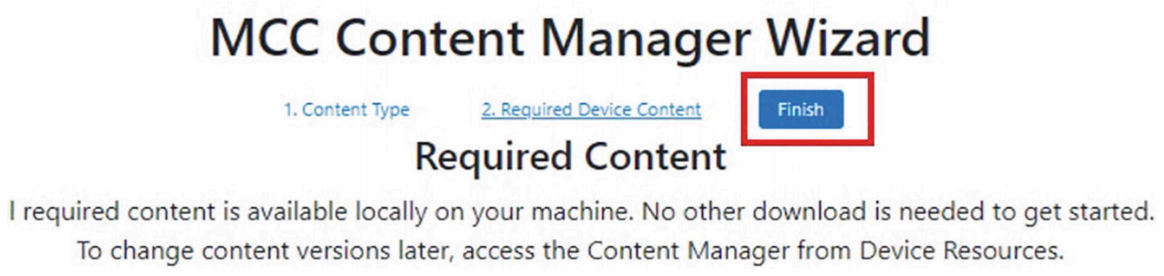
MCC Melody
Supports the MCC Builder
Supports content versioning at driver level
An iteration of MCC Generated Code
Works both on- and off-line
Select MCC Melody
Release notes and supported devices

MCC Classic
Development process you are accustomed to
All components and libraries that you have used before
Select MCC Classic
Release notes and supported devices

MPLAB® Harmony
Embedded Software Development Framework for 32-bit Microcontrollers and Microprocessors
Select MPLAB Harmony
Release notes and supported devices

Figure 11-2 MCC content manager wizard

Once you have the connect manager open, select "MCC Classic" from the options and click the finish button as shown in Figure 11-3.


MCC Content Manager Wizard

1. Content Type 2. Required Device Content Finish

Required Content

l required content is available locally on your machine. No other download is needed to get started.
To change content versions later, access the Content Manager from Device Resources.

Figure 11-3 Clicking the Finish button

After we click the Finish button, you should see the Device Resources window come up. When this window opens, we select CLC1 as shown in Figure 11-4.

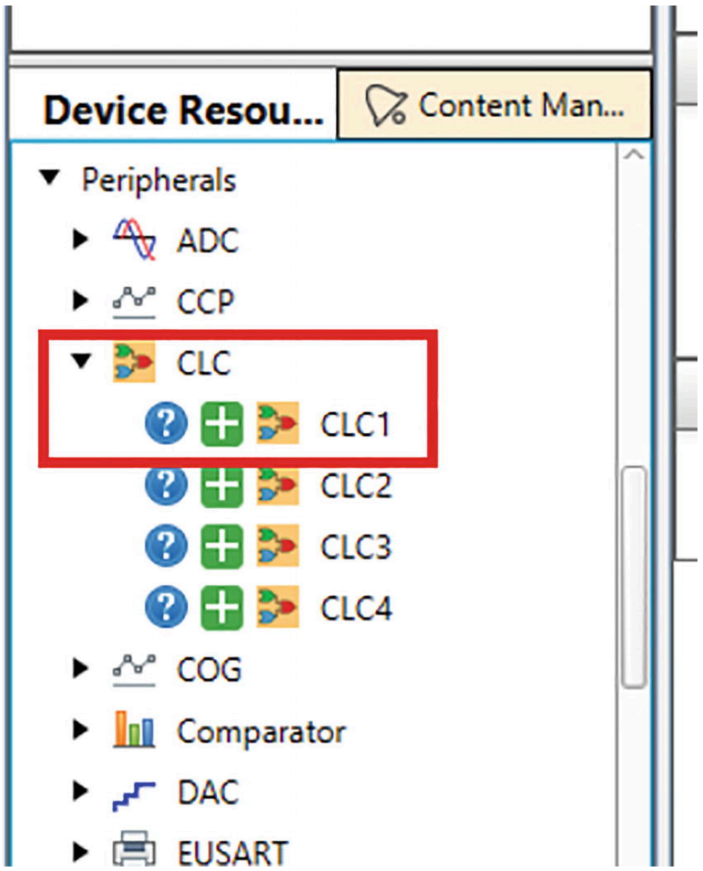
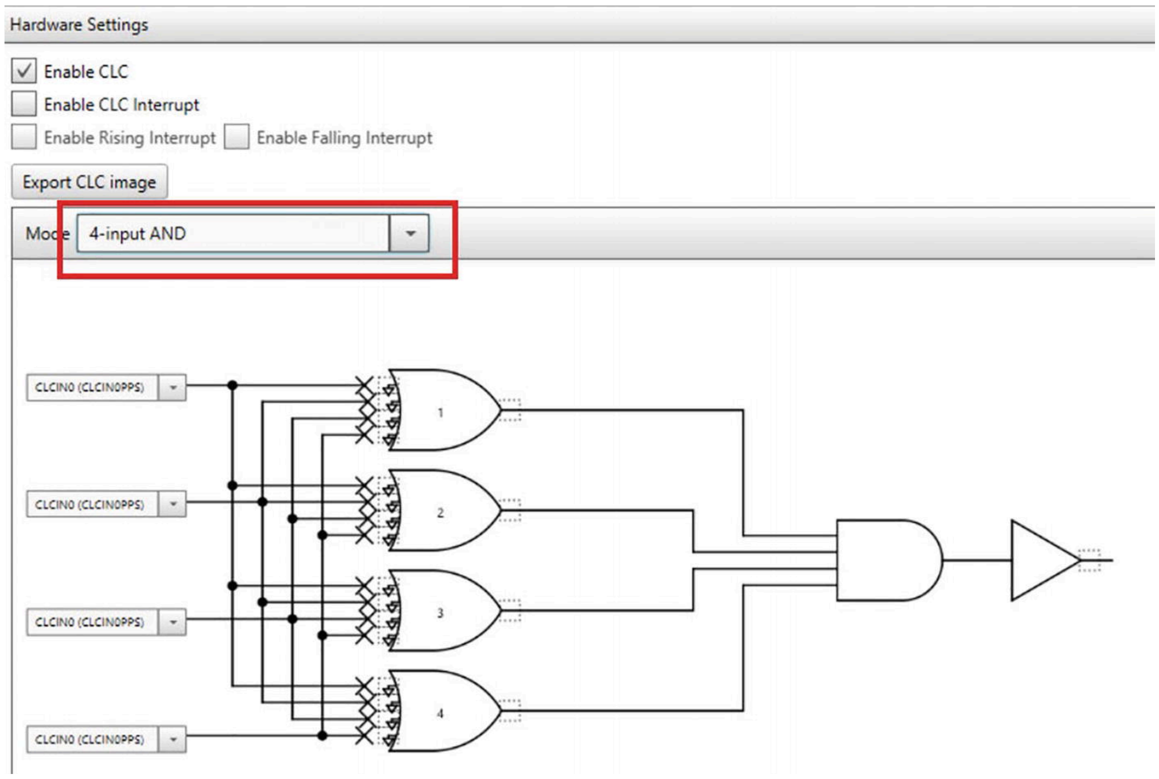


Figure 11-4 Selecting the CLC module

After selecting the CLC module, we need to set the mode to 4-input AND and then ensure the input is not inverted as shown in Figure 11-5.


Hardware Settings

☑ Enable CLC

☐ Enable CLC Interrupt

☐ Enable Rising Interrupt ☐ Enable Falling Interrupt

Export CLC image

Mode 4-input AND

Figure 11-5 Selecting 4-input AND

At each of the four OR gates, you will notice four lines marked with X. Click the first, second, third, and fourth of these and select the inputs as CLCIN1, CLCIN2, CLCIN3, and CLCIN4, respectively. We can then select which pins are used for the input from the pin manager as shown in Figure 11-6.

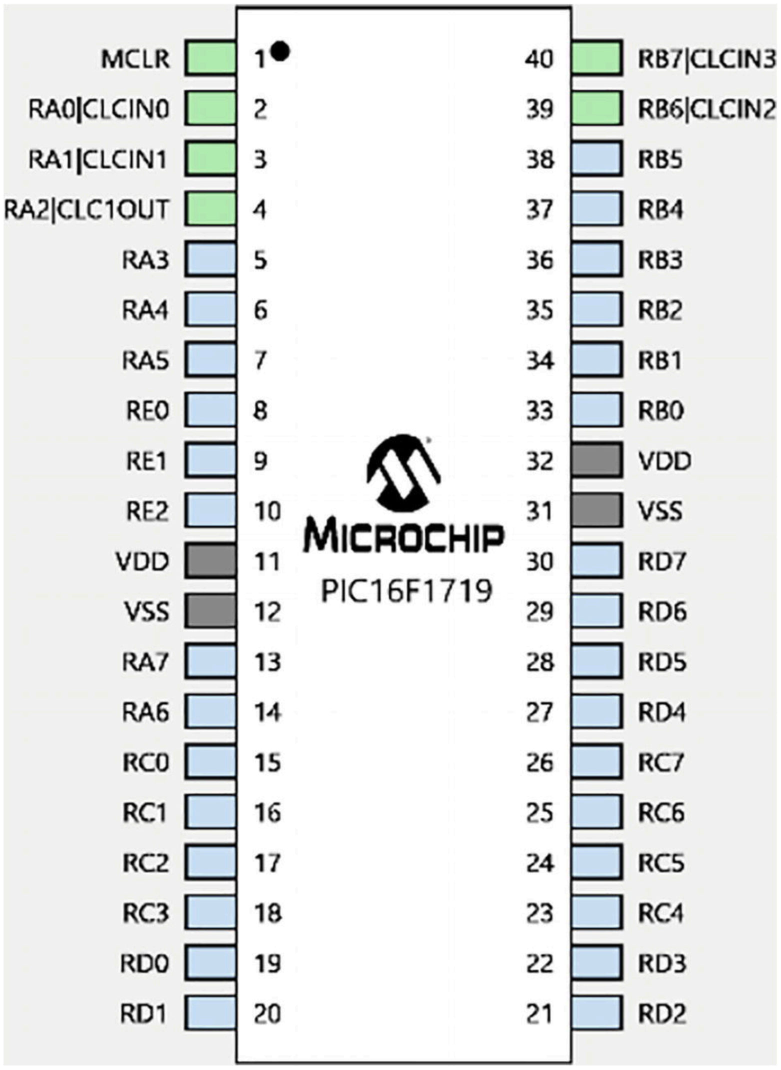


Figure 11-6 The pin manager

We then select the need to generate the code by clicking the Generate button as shown in Figure 11-7.

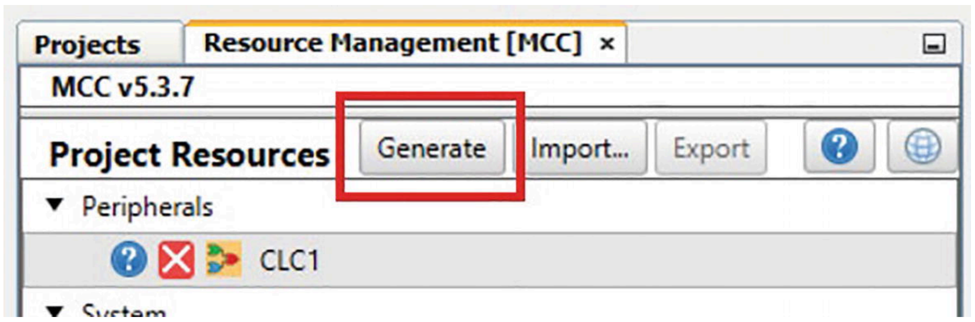


Figure 11-7 Using the Generate button

Once the code is finished generating, the program should look like that shown in Figure 11-8.

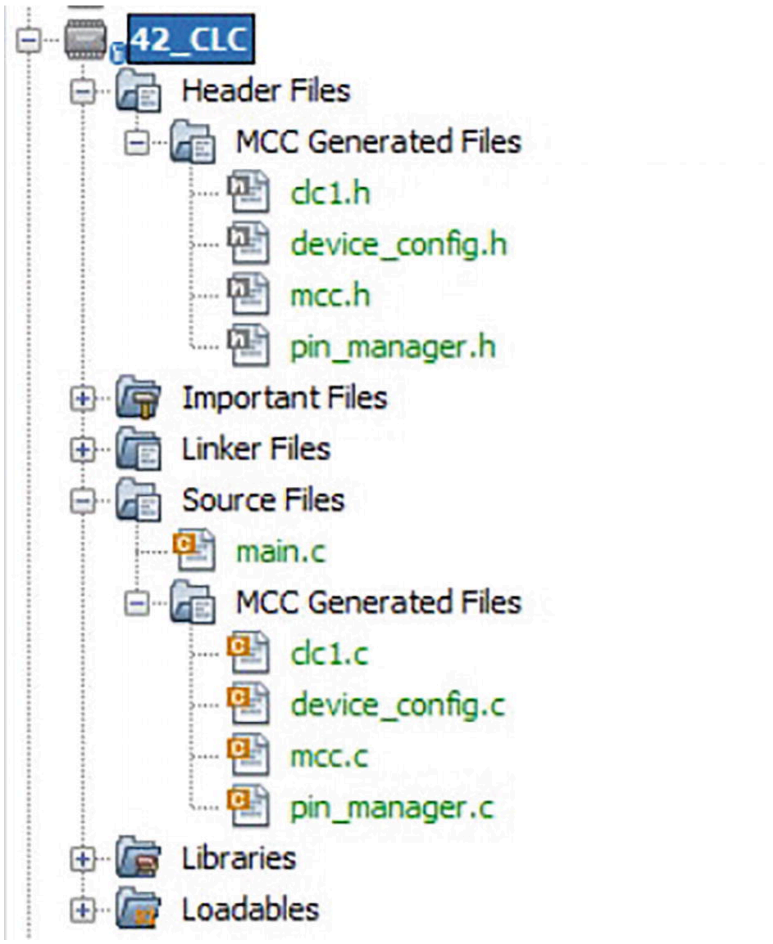


Figure 11-8 The generated program

MCC makes it possible to easily generate code. There is no main code to be written by the developer. However, in this book, I deliberately avoided the use of the MCC to allow you to get a good understanding of the underlying hardware by writing and configuring modules and peripherals. In the case of the CLC, the MCC has a visual configuration tool that is very simple to use, and it is easier to leverage for this project than writing the configuration code manually.

Configurable Logic Block (CLB)

The CLC peripheral has been on the PIC microcontroller for a while. Once you start using the CLCs, you start to realize that you run out of logic or are limited by the available features of a CLC block. For this reason, some newer PIC microcontrollers have a CLB that helps with these shortcomings. To understand the configurable logic blocks on the PIC microcontroller, it's important to understand the first-ever FPGA: the Xilinx XC2064.

The XC2064 was architected around a novel concept: an array of configurable logic blocks (CLBs), surrounded by a programmable interconnect matrix and I/O blocks. This design allowed for the implementation of custom logic functions by configuring the connections between these blocks, effectively enabling designers to create and iterate on complex digital circuits without the need for fabricating a new chip for each design iteration. Each CLB in the XC2064 contained a small amount of logic, which could be interconnected to perform a vast array of digital functions, from simple logic gates to more complex combinational and sequential logic.

With approximately 1200 gates, the XC2064 was not just a proof of concept but a practical tool that could be used in a variety of applications, from prototyping to actual product deployment. Its programmability was facilitated through a hardware description language (HDL), allowing designers to describe the desired logic functions at a high level without worrying about the physical layout of the chip. This significantly reduced development time and costs, democratizing access to custom IC design and laying the groundwork for the FPGA industry's explosive growth.

The XC2064's introduction was a disruptive innovation, challenging the traditional ASIC (Application-Specific Integrated Circuit) and custom IC design that was available at the time. It provided unparalleled design flexibility, allowing for rapid prototyping and modifications, which was particularly valuable in the fast-evolving tech landscape of the 1980s and beyond. Moreover, the XC2064 set the stage for the development of increasingly complex FPGAs, featuring millions of gates, sophisticated on-chip resources like DSP blocks and high-speed transceivers, and support for complex system-on-a-chip (SoC) designs.

The new PIC microcontroller CLBs start with having around 32 basic logic elements (BLEs) that are similar to the BLEs of the XC2064. The CLBs can have flip flops, hardware counters, multiplexers, buffers, and 4-input LUTs. What makes it interesting is that the logic is programmable on the fly. You can create a lot with these as they are like little CLPDs on the microcontroller. You can look at PIC devices that have CLBs if you wish to explore this peripheral.

Numerically Controlled Oscillator

A Numerically Controlled Oscillator (NCO) is an essential component in the domain of digital signal processing and communication systems. It generates a precise waveform, typically a sine wave or a square wave, at a specific frequency that can be digitally adjusted. Unlike traditional oscillators, which rely on analog components such as resistors, capacitors, or crystals to determine their output frequency, an NCO utilizes digital logic and arithmetic to synthesize the desired signal. If you are new to electronics design, you may think that an NCO is similar to the PWM we explored earlier. However, NCOs have greater resolution and precision than a PWM. NCOs are so precise that we can use them in things like RF signal generation and digital synthesis.

At the heart of an NCO's functionality is a phase accumulator. The phase accumulator updates its value at each clock cycle by adding a phase increment value, which corresponds to the desired output frequency. By continuously cycling through these values, the NCO generates a digital representation of the waveform, which can be converted to an analog signal using a DAC if needed. This process allows for the precise generation of waveforms at a wide range of frequencies with minimal phase noise and jitter. This is a powerful addition to our PIC microcontroller peripherals.

One of the key advantages of NCOs is their ability to rapidly switch between output frequencies without the settling time associated with analog oscillators. This feature is particularly valuable in applications such as frequency hopping in spread spectrum communication systems, where the ability to quickly change frequencies is crucial for security and interference avoidance. Additionally, the digital nature of NCOs enables easy integration with other digital systems and microcontrollers, facilitating complex modulation schemes, signal generation, and processing tasks with high precision and minimal external components.

Moreover, NCOs are highly programmable, allowing for dynamic adjustment of frequency, phase, and waveform shape. The use of digital techniques also ensures that NCOs maintain their accuracy and stability over temperature and supply voltage variations, unlike their analog counterparts, which can be more susceptible to environmental changes.

On the PIC16F1719, the onboard NCO has three registers that allow it to have up to a 20-bit increment, which is a very fine granularity for frequency control. The program for using the NCO is given in Listing 11-1.

```
/*
 * File: main.c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/int OSC @ 32MHz, 5v
 * Program: 16_NCO
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 * Program Version: 1.2
 *
 * Program Description: This generates a 1kHz NCO Signal on PA4
 *
 * Hardware Description: An Oscilloscope probe is connected to
pin PA4
 *
 * Created November 4th, 2016, 1:00 PM
 * Last Updated: November 18th, 2023, 4:55 AM
 */

/*************************************************************
 *Includes and defines
 *************************************************************/
#include "PIC16F1719_Internal.h"

/*************************************************************
 * Function: void initMain()
 *
 * Returns: Nothing
 *
 * Description: Contains initializations for main
 *
 * Usage: initMain()
 *************************************************************/
```

```
void initMain(){
    // Run at 16 MHz
    internal_32();

    /////////////////
    // Set UP NCO
    ////////////////
    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    PPSLOCK = 0x00; // unlock PPS

    RA4PPSbits.RA4PPS = 0x03; // NCO1 OUTPUT to PORTA4

    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    PPSLOCK = 0x01;

    TRISAbits.TRISA4 = 0;
    ANSELAbits.ANSA4 = 0;
}

/*************************************************************
 * Function: Main
 *
 * Returns: Nothing
 *
 * Description: Program entry point
 *************************************************************/

void main(void) {
    initMain();

    // run NCO at 1000 Hz
    NCO1INCH = 0;
    NCO1INCL = 65;

    // Enable NCO
    NCO1CONbits.N1EN = 1;

    // Operate in Fixed Duty cycle mode
    NCO1CONbits.N1PFM = 0;

    // Output signal active high
    NCO1CONbits.N1POL = 0;

    // Clock is 32 MHz System Oscillator (FOSC))
    NCO1CLKbits.N1CKS = 0b01;
    while(1){

    }

    return;

}
```

Listing 11-1 Using the NCO Module

In our main program, we configure the NCO to generate a 1 kHz signal. The NCO frequency is determined by loading values into the NCO1INCH and NCO1INCL registers, with NCO1INCL being set to 65. These registers define the increment value for the NCO's phase accumulator, effectively setting the output frequency. The NCO is then enabled by setting the N1EN bit of the NCO1CON register.

Additional configurations include setting the NCO to operate in fixed duty cycle mode by clearing the N1PFM bit, ensuring the output signal is active high by clearing the N1POL bit and selecting the clock source as the 32 MHz system oscillator (FOSC) by setting the N1CKS bits to 0b01. These settings fine-tune the NCO's operation, ensuring it generates the desired signal characteristics.

The program then enters an infinite loop, during which the NCO continuously outputs the 1 kHz signal on pin RA4. We can view the output with an oscilloscope.

Comparator

A comparator is an electronic device that compares two voltages or currents and outputs a digital signal indicating which is higher. It essentially operates as a basic decision-making device within an electronic circuit, delivering one of two levels of output voltage based on the comparison of the input signals. At its core, a comparator consists of a high-gain differential amplifier without any feedback, making it ideally suited for open-loop operation. This simplicity in function helps the critical role comparators play in both analog and digital electronics, where they serve as the fundamental building blocks for more complex circuits.

Comparators are pivotal in systems that require a transition between analog and digital domains. They enable the conversion of analog signals into digital form, making them essential for various applications that involve measurement, detection, and digital control of analog signals. For instance, in an analog-to-digital conversion (ADC) process, comparators are used to determine the bit levels of the digital output by comparing the input analog signal against a set of reference voltages. This capability allows for the precise translation of analog phenomena into digital data, which can then be processed, stored, or transmitted by digital systems.

Moreover, comparators find extensive use in control systems. By comparing a process variable to a set point or reference, they can generate a signal that indicates whether the process variable is above or below the desired threshold. This signal can then trigger corrective actions, such as turning a heater on or off in a temperature control system. Such applications underscore the comparator's role in implementing feedback and control mechanisms, where maintaining a certain condition within a specific range is necessary for system stability and performance.

In addition to ADC and control systems, comparators are instrumental in timing and oscillation circuits. They can generate precise timing signals by comparing the voltage across a charging capacitor to a reference voltage, thus creating oscillators and timers. These applications exploit the comparator's ability to produce a clean, abrupt transition at a specific threshold, making them ideal for generating stable, periodic signals in clocks, pulse generators, and waveform synthesizers.

Comparators also enhance system reliability and safety through overvoltage and undervoltage detection. By monitoring power supply voltages and generating an alert or shutdown signal when these voltages fall outside safe operating limits, comparators help protect sensitive electronic components from damage due to improper power conditions. This protective function is crucial in embedded systems, where operational integrity and longevity are paramount.

In the realm of embedded design, comparators are versatile components used across a wide array of applications:

Battery Level Monitoring – To ensure devices operate within their battery's safe voltage range and to trigger low-battery warnings
Light/Dark Sensor – Comparing ambient light levels to a preset threshold for applications like automatic headlights or night lights
Temperature Control Systems – Activating a cooling or heating device when temperature deviates from a set point
Voltage Regulation – Maintaining a stable output voltage by comparing the regulator output to a reference voltage
Motor Speed Control – Detecting speed by comparing the back EMF of a motor to a reference value and adjusting power accordingly
Touch Sensors – Detecting touch by comparing the change in capacitance or resistance to a threshold
Pulse Width Modulation (PWM) Control – For generating PWM signals based on a comparison of a sawtooth wave to a control voltage
Water Level Detection – Monitoring water levels in tanks or reservoirs by comparing the sensor output to predetermined levels
Position Sensors – Determining position by comparing the output of position-sensitive devices to reference values
Current Sensing and Overcurrent Protection – Detecting overcurrent conditions by comparing sensed current levels to a safe threshold

The program for using the comparator is in Listing 11-2.

```
/*
 * File: main.c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/int OSC @ 32MHz, 5v
 * Program: 10_Comparator
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 * Program Version: 1.2
 *
 * Program Description: This Program uses the onboard
comparator module of the PIC microcontroller. When the voltage
at Vin+ is more than the voltage at Vin- the comparator outputs
a logic level high and vice versa.
 *
 * Hardware Description: An LED is connected via a 10k resistor
to PIN RA3 and the output of a 10k voltage divider is fed into
the - input (PIN RA0) of the comparator with the + input (PIN
RA2) being the output of a 10k pot is fed into the positive
end.
 *
 * Created November 4th, 2016, 1:00 PM
 * Last Updated: November 18th, 2023, 2:28 AM
 */

/*************************************************************
 *Includes and defines
 *************************************************************/
#include "PIC16F1719_Internal.h"

/*************************************************************
 * Function: void initMain()
 *
 * Returns: Nothing
 *
 * Description: Contains initializations for main
 *
 * Usage: initMain()
 *************************************************************/

void initMain(){
    // Run at 32 MHz
    internal_32();

    //////////////////////
    // Setup Comparator1
    /////////////////////

    //_____________
    // RA0 = C1IN0-
    //_____________

    // Set as input
    TRISAbits.TRISA0 = 1;
```

```
// Set analog mode on
ANSELAbits.ANSA0 = 1;

//______________
// RA2 = C1IN0+
//______________

// Set as input
TRISAbits.TRISA2 = 1;

// Set analog mode on
ANSELAbits.ANSA2 = 1;

TRISAbits.TRISA3 = 0;

//___________________
// RA3 = C1OUT
//___________________

PPSLOCK = 0x55;
PPSLOCK = 0xAA;
PPSLOCKbits.PPSLOCKED = 0x00; // unlock PPS

RA3PPSbits.RA3PPS = 0x16;   //RA3->CMP1:C1OUT;

PPSLOCK = 0x55;
PPSLOCK = 0xAA;
PPSLOCKbits.PPSLOCKED = 0x01; // lock PPS

TRISAbits.TRISA3 = 0;

/////////////////////////
// configure comparator1
/////////////////////////

// enable comparator
CM1CON0bits.C1ON = 1;

// output not inverted
CM1CON0bits.C1POL = 0;

// normal power mode
CM1CON0bits.C1SP = 1;

// asynchronous output
CM1CON0bits.C1SYNC = 0;
```

```
    // turn on the zero latency filter
    CM1CON0bits.C1ZLF = 1;

    //Enable comparator hysteresis (45 mV)
    CM1CON0bits.C1HYS = 1;

    // + in = C1IN+ pin
    CM1CON1bits.C1PCH = 0b00;

    // - in = C1IN- PIN
    CM1CON1bits.C1NCH = 0b00;
}

/**************************************************************
 * Function: Main
 *
 * Returns: Nothing
 *
 * Description: Program entry point
 **************************************************************/

void main(void) {
    initMain();

    while(1){
        // Have the CPU sleep!
        SLEEP();
    }

    return;
}
```

Listing 11-2 Using the Comparator

The program then proceeds to configure the comparator by setting pins RA0 and RA2 as inputs and enabling their analog mode to accommodate the voltage levels to be compared. Pin RA3 is set as an output to drive the LED. The Peripheral Pin Select (PPS) system is unlocked, and pin RA3 is configured to output the comparator's result. After this setup, the PPS system is locked again to secure the configuration.

The comparator itself is configured to be enabled, with its output polarity set to non-inverted, indicating that a high output will occur when the positive input voltage is greater than the negative input voltage. The program selects normal power mode for the comparator, disables output synchronization (making the output asynchronous), enables a zero-latency filter to allow immediate response to input changes, and activates hysteresis to improve noise immunity. The inputs for the comparator are explicitly set to the designated pins: RA2 for positive and RA0 for negative.

Following initialization, the main() function engages an infinite loop where the microcontroller is put into sleep mode to conserve power, relying on the comparator's output to activate the LED as per the voltage comparison. The circuit for testing this program is given in Figure 11-9.

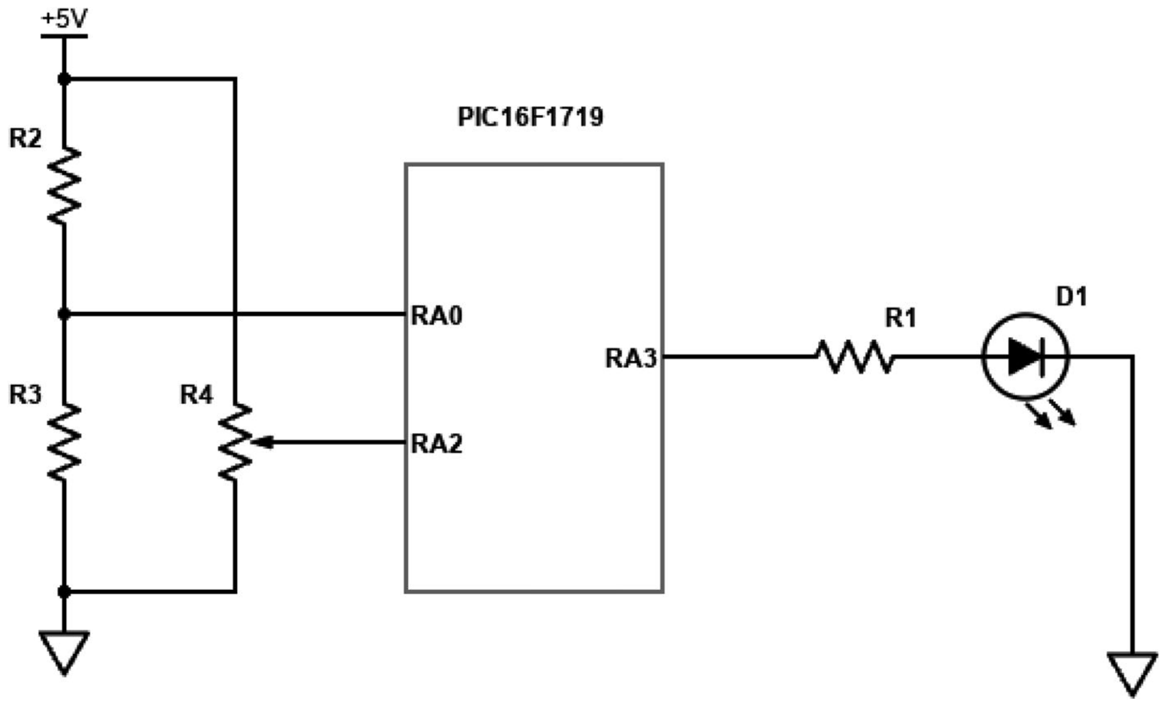


Figure 11-9 Testing the comparator

The circuit is rather simple. We have our voltage divider connected to pin RA0 and a potentiometer connected to pin RA2. Pin RA3 is the output pin that is connected via resistor R1 to an LED.

Fixed Voltage Reference (FVR)

A Fixed Voltage Reference (FVR) is an integrated circuit feature or stand-alone component that provides a stable and precise voltage reference regardless of variations in power supply voltage, temperature, and load conditions. This reference voltage is crucial for the accurate operation of analog-to-digital converters (ADCs), digital-to-analog converters (DACs), and other sensitive electronic circuits that require a stable reference point for accurate measurements and signal processing. The FVR is particularly valuable in embedded systems and precision electronics, where consistent and reliable voltage levels are critical for maintaining system accuracy and performance. By offering a fixed, known voltage level, an FVR enables these systems to perform complex analog and digital computations with high precision, making it a fundamental component in the design of robust and reliable electronic devices. The FVR can be used for modules such as the ADC or comparator. We see an example of how to use the FVR with the comparator in Listing 11-3.

```
/*
 * File: main.c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/int OSC @ 32MHz, 5v
 * Program: 11_FVR
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 * Program Version: 1.2
 *
 * Program Description: This Program utilizes the FVR to
provide a reference for the positive input of the comparator.
 *
 * Hardware Description: An LED is connected via a 10k resistor
to PIN RA3 and the output of a 10k pot is fed into the - input
(PIN RA0) of the comparator.
 *
 * Created November 4th, 2016, 1:00 PM
 * Last Updated: November 18th, 2023, 2:43 AM
 */

/*************************************************************
 *Includes and defines
 *************************************************************/
#include "PIC16F1719_Internal.h"

/*************************************************************
 * Function: void initMain()
 *
 * Returns: Nothing
 *
 * Description: Contains initializations for main
 *
 * Usage: initMain()
 *************************************************************/

void initMain(){
    // Run at 32 MHz
    internal_32();

    //////////////////////
    // Setup Comparator1
    /////////////////////

    //_____________
    // RA1 = C1IN1-
    //_____________

    // Set as input
    TRISAbits.TRISA0 = 1;
```

```
// Set analog mode on
ANSELAbits.ANSA0 = 1;

//___________________
// RA3 = C1OUT
//___________________

PPSLOCK = 0x55;
PPSLOCK = 0xAA;
PPSLOCKbits.PPSLOCKED = 0x00; // unlock PPS

RA3PPSbits.RA3PPS = 0x16;   //RA3->CMP1:C1OUT;

PPSLOCK = 0x55;
PPSLOCK = 0xAA;
PPSLOCKbits.PPSLOCKED = 0x01; // lock PPS

TRISAbits.TRISA3 = 0;

/////////////////////////
// configure comparator1
/////////////////////////

// enable comparator
CM1CON0bits.C1ON = 1;

// output not inverted
CM1CON0bits.C1POL = 1;

// normal power mode
CM1CON0bits.C1SP = 1;

// hysteresis disabled
CM1CON0bits.C1HYS =0;

// asynchronous output
CM1CON0bits.C1SYNC = 0;

// turn on the zero latency filter
CM1CON0bits.C1ZLF = 1;

// Set IN+ to fixed voltage reference
CM1CON1bits.C1PCH = 0b110;

// - in = C1IN- PIN (C1OUT = 1 if < 2.048v)
CM1CON1bits.C1NCH = 0b00;
```

```
    ///////////////////////////
    // Configure FVR
    //////////////////////////

    // Enable the FVR
    FVRCONbits.FVREN = 1;

    // Output 2.048v to comparators
    FVRCONbits.CDAFVR = 0b10;
}

/*************************************************************
 * Function: Main
 *
 * Returns: Nothing
 *
 * Description: Program entry point
 *************************************************************/

void main(void) {
    initMain();

    while(1){
        // Put the CPU to sleep!
        SLEEP();
    }

    return;
}
```

Listing 11-3 Using the FVR with the Comparator

It sets up the comparator (Comparator1) by configuring PIN RA0 as an analog input for the comparator's negative input and PIN RA3 for the comparator's output. The Peripheral Pin Select (PPS) feature is unlocked and configured to route the comparator's output to RA3 and then re-locked, ensuring the microcontroller's pin configurations are secure.

The comparator is configured to be active, with its output polarity not inverted, operating in normal power mode without hysteresis, and set for asynchronous output with zero-latency filtering enabled. The positive input of the comparator is connected to the FVR, set to output a fixed voltage of 2.048 V, to serve as a stable reference voltage for comparison.

The FVR itself is enabled and configured to provide a 2.048 V output specifically for use by the comparator. This setup allows for the comparison of an external voltage against a precise reference voltage, useful in various applications where voltage threshold detection is required.

Finally, the main function calls initMain() to execute the initialization routine and then enters an infinite loop where it continuously puts the CPU to sleep. The schematic for this program is given in Figure 11-10.

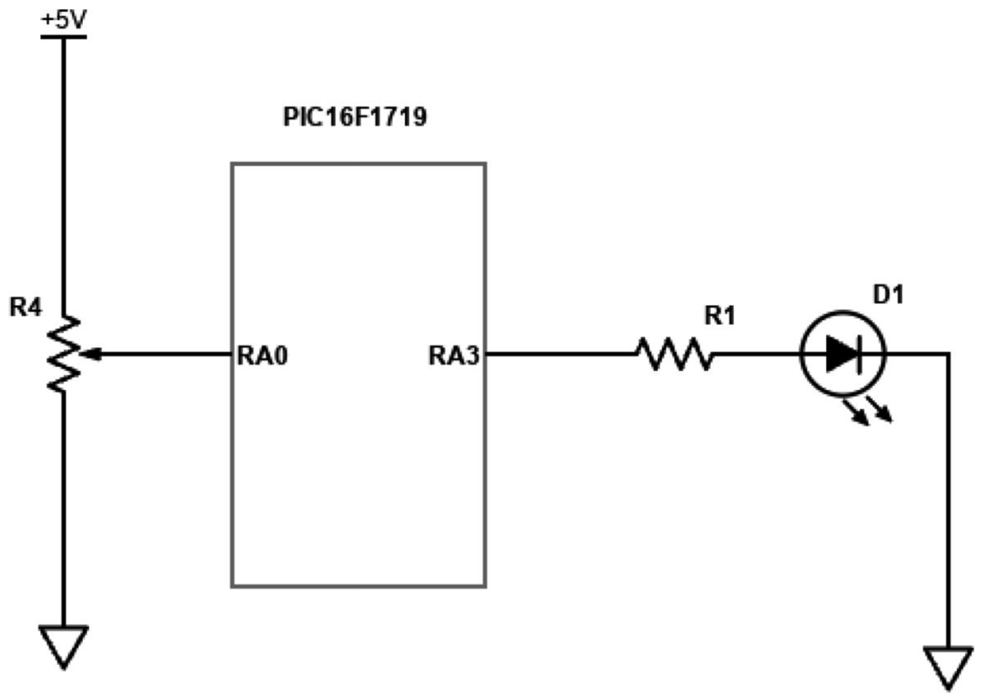


Figure 11-10 Using the FVR circuit schematic

Our schematic diagram is similar to our circuit setup when we were using an external reference, only this time since we are using an internal voltage reference; we just have a potentiometer connected to pin RA0. The output of the comparator is viewed on the LED connected to pin RA3.

Conclusion

In this chapter, we covered core independent peripherals, taking a look at the Configurable Logic Cell, Numerically Controlled Oscillator, comparator, and Fixed Voltage Reference modules.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2024
A. Subero, *Programming PIC Microcontrollers with XC8*, Maker Innovations Series
https://doi.org/10.1007/979-8-8688-0467-0_12

12. Wi-Fi and Bluetooth

Armstrong Subero[1]

(1) Moruga, Trinidad and Tobago

As embedded systems evolve, connectivity is at the cornerstone of most modern embedded systems. This chapter delves into the world of Wi-Fi and Bluetooth, the dual pillars of wireless communication that have reshaped the contours of connectivity in the digital age. Long gone are the days when wireless meant AM or FM connectivity. In this chapter, we explore low-cost options for adding Wi-Fi and Bluetooth to your project designs.

Low-Cost Wireless Connectivity

Low-cost connectivity has become a cornerstone in the development and proliferation of embedded systems, drastically transforming their capabilities and the scope of their applications. When I first started with embedded systems, adding Wi-Fi or Bluetooth was seen as an incredible luxury, reserved for only the most premium of systems. Usually when you wanted to add wireless connectivity, ISM (Industrial, Scientific, Medical) band radios were your options. A 2.4 GHz transceiver or a 433 MHz radio with Manchester protocol for encoding and decoding was usually your go-to choice for any sort of wireless connectivity.

In a world increasingly driven by the Internet of Things (IoT), the ability to add wireless communication functionalities at a minimal cost has not only democratized the development process but also catalyzed innovation across various sectors. For embedded systems, this means that even small, resource-constrained devices can now be part of a larger network, communicate with other devices, and access the Internet.

Low-cost connectivity facilitates the rapid prototyping and testing of ideas, lowering the barrier to entry for startups and hobbyists alike. This has led to a vibrant ecosystem of innovators experimenting with embedded systems in ways previously not feasible, driving forward technological advancements and creating new markets. At the heart of this are Wi-Fi and Bluetooth. While there are many modules available today, I will examine two of the most common options available on the market.

Wi-Fi

Wi-Fi is a type of protocol used for wireless networking. Wi-Fi allows a device to communicate over TCP/IP wirelessly. The most important parts of the Wi-Fi network are the Wireless Access Point (AP), which is the epicenter of communications, and a station, which is a device that can connect to an access point. In your home or office, this access point usually allows you to connect to the Internet. Each device on your Wi-Fi network is assigned a MAC address, which is a unique 48-bit value that allows a particular node on a network to distinguish itself from another node. One of the benefits of Wi-Fi is that it allows you to set up a network more cheaply and easily than when using a wired network. In the embedded systems context, it is easier to integrate Wi-Fi into your systems vs. Ethernet since the Wi-Fi module we will examine is readily available. Writing your TCP/IP tasks takes a lot of work. In this section, we examine one of the simplest ways to integrate networking into your embedded systems. While there are ways to set up an Ethernet stack with something like the ENC28J60, Wi-Fi is a bit less cumbersome to use.

Expanding further into the domain of Wi-Fi and its integration into embedded systems, it's crucial to understand the advancements and the flexibility Wi-Fi offers. Beyond the basic architecture of access points and stations, Wi-Fi technology has seen significant enhancements, such as the introduction of Wi-Fi 6, which provides higher data rates, increased capacity, performance in environments with many connected devices, and improved power efficiency. These improvements are particularly beneficial in embedded systems, where the need for reliable data transmission and efficient power use is often critical. For instance, IoT devices, smart home appliances, and industrial sensors can leverage Wi-Fi 6's capabilities to operate more effectively within a networked environment, enhancing the user's experience and the system's overall efficiency.

Furthermore, the security aspect of Wi-Fi cannot be overstated, especially in the context of embedded systems. With technologies like WPA3, Wi-Fi networks offer robust protection against common cyber threats, ensuring that data transmitted over the airwaves remains secure. This is paramount in applications handling sensitive information, such as personal data in healthcare devices or operational data in industrial control systems. Developers integrating Wi-Fi into their projects must pay close attention to implementing these security measures to safeguard their systems against unauthorized access and data breaches.

Lastly, the ecosystem surrounding Wi-Fi technology, including development tools, software libraries, and community support, plays a significant role in simplifying the integration process for embedded systems developers. Numerous resources are available to help developers navigate the complexities of networking, from selecting the right Wi-Fi module to implementing the TCP/IP stack and ensuring seamless connectivity. Open-source projects and platforms offer pre-built libraries and examples that drastically reduce the time and effort required to add networking capabilities to embedded devices. We will look at how we can add low-cost Wi-Fi to our projects using the ESP8266 module.

The ESP8266

The ESP8266, developed by Espressif Systems, is a low-cost Wi-Fi module that has gained substantial popularity in the world of Internet of Things (IoT) projects and embedded systems applications. Though the newer ESP32 has been introduced that includes Wi-Fi capability, the ESP8266 is still widely available. With its fully integrated TCP/IP protocol stack, it can offer Wi-Fi networking to any microcontroller, making it an ideal solution for developers looking to enable Internet connectivity in their projects. The ESP8266 is not just a simple Wi-Fi module but a self-contained SOC (system on a chip) that can be programmed using software development kits (SDKs) provided by Espressif Systems in C or Lua scripting language. Its impressive capabilities, combined with its low cost, have made it a favorite choice for hobbyists and professionals alike, who seek to add wireless networking capabilities to their devices without significantly increasing the cost.

One of the most compelling features of the ESP8266 is its versatility. It can be used as a stand-alone microcontroller or as a slave device, controlled by a host microcontroller through AT commands over a UART interface. This flexibility allows it to fit into a wide array of applications, from simple projects like home automation systems and weather stations to more complex ones like smartwatches and drones. The module supports various modes, including Station mode, Access Point mode, and both simultaneously, providing the ability to connect to an existing Wi-Fi network and to act as an access point to which other Wi-Fi devices can connect. Furthermore, its deep sleep mode ensures that battery-powered applications can remain functional for months or even years on a single charge.

The development community around the ESP8266 is robust and active, with countless tutorials, forums, and third-party tools available to help beginners and experts alike. The module is supported by the Arduino IDE, among others, which significantly lowers the barrier to entry for those new to programming or those transitioning from other platforms. Open source projects and libraries have greatly expanded their capabilities beyond simple web servers, enabling everything from MQTT clients for home automation to complex sensor networks. As IoT continues to grow, the ESP8266 remains at the forefront, providing an accessible, affordable, and flexible platform for wireless connectivity in a multitude of applications. Though we can use the device stand-alone on its own, we can leverage our knowledge of PIC microcontrollers to make interfacing with this module simple.

Testing the ESP8266

The ESP8266 has a built-in processor that allows you to communicate with it via AT commands. Let's look at some of these commands, which can be used to control the ESP8266. Table 12-1 lists some commands you can use to test the ESP8266.

Table 12-1 Commands for Testing the ESP8266

Command	Function
AT	Tests if the AT system works OK
AT+RST	Resets the module

Command	Function
AT+GMR	Prints the version of the firmware installed on the ESP8266
AT+CWMODE?	Wi-Fi mode of ESP8266
AT+CWJAP = SSID, PWD	Connects to SSID with password specified
AT+CWLAP	Lists all available access points

Project: Wi-Fi Data Logger

In this project, we use the PIC16F1719 to send wireless data over Wi-Fi, which can be viewed in any web browser. We will set up the ESP8266 in server mode for a single connection. The PIC16F1719 does not have the RAM, ROM, or processing power to build a full web page. Since it only uses a baud rate of 9600 to communicate with the ESP8266, it would take too long to send a full web page and perform all the necessary checks to ensure that the ESP8266 is receiving commands. A web page would also use a good portion of the onboard storage of the microcontroller. To compensate for this limitation, we will send the minimum commands necessary to set up the ESP8266 as a web server. We will also use the watchdog timer. The WDT will be used initially at a timeout of four seconds to ensure that the server starts up properly. Parsing all the required strings sent by the ESP8266 would add extra overhead. After the server is set up, the watchdog timer will be set to have a higher timeout of 128 seconds, up from the original four seconds. The output of the ESP8266 can be viewed in any web browser once you get the IP address via your router. In your web address bar, type the IP of the device followed by :80/ and wait for it to load. The schematic is shown in Figure 12-1.

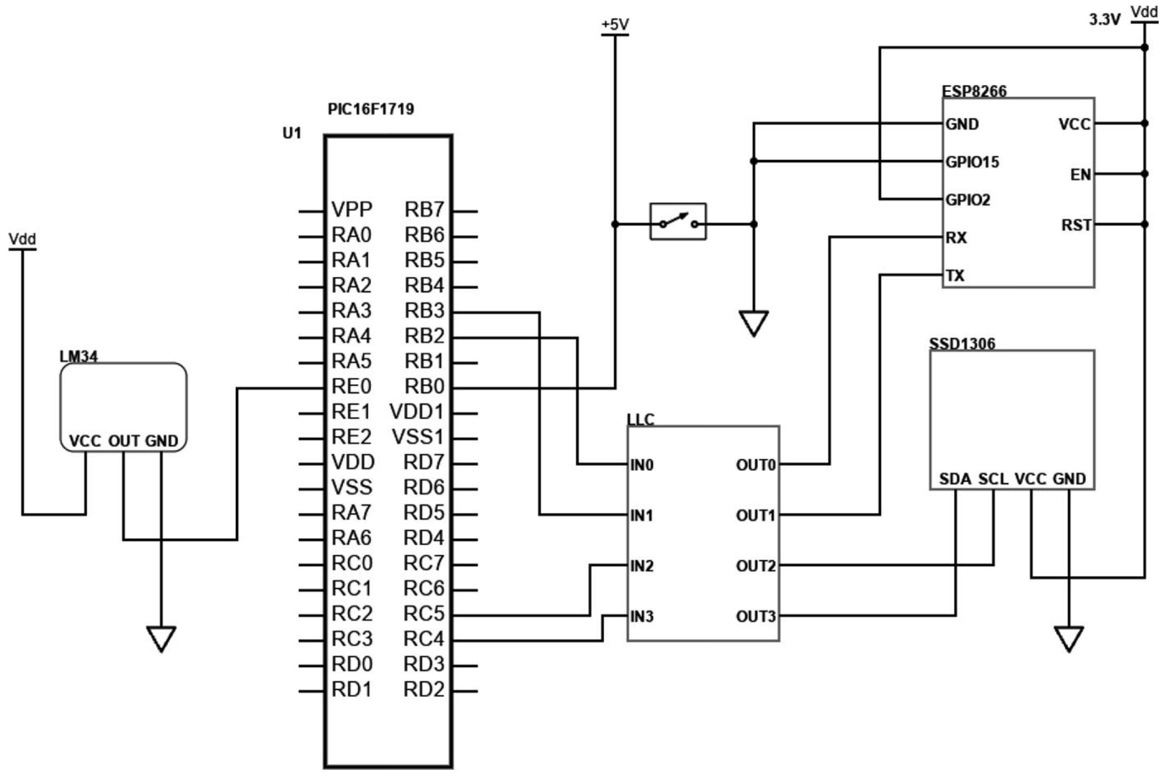


Figure 12-1 Our Wi-Fi data logger schematic

The program for our Wi-Fi logger is given in Listing 12-1.

```
/*
 * File: Main.c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/Int OSC @ 16MHz, 5v
 * Program: 45_ESP8266
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 * Program Version: 1.0
 *
 *
 * Program Description: This Program gives a reading in Celsius
based on the output of a LM34 temperature sensor which is then
sent via WiFi using the ESP8266 ESP-12-F the output of which
can be read in an web browser. The program uses the watchdog
timer initially with a timeout of 4s then once the server is
operational has a timeout of 128s after which the server will
reboot.
 *
 * Hardware Description: A LM34 is connected to PIN E0 and a
SSD1306 OLED is connected to the I2C bus. The ESP8266 is
connected as follows:
 *                          GND    -> GND
 *                          TX     -> RB3
 *                          RX     -> RB2
 *                          GPIO15 -> GND
 *                          GPIO2  -> VCC
 *                          RST    -> VCC
 *                          EN     -> VCC
 *                          VCC    -> VCC
 *
 *                          External interrupt is connected to
PINB0
 *
 * Created March 31st, 2017, 10:57 AM
 * Updated March 30th, 2024, 2:56 PM
 */

/***************************************************************
 *Includes and defines
 ***************************************************************/

#include "PIC16F1719_Internal.h"
#include "I2C.h"
#include "oled.h"
```

```
#include "EUSART.h"
#include <string.h>

// Buffer for UART transactions
char buf[50];

// Function prototypes
float Read_Temperature();
void server_Initialize();

/*************************************************************
 * Function: void initMain()
 *
 * Returns: Nothing
 *
 * Description: Contains initializations for main
 *
 * Usage: initMain()
 *************************************************************/

void initMain(){
    // Run at 16 MHz
    internal_16();

    /////////////////////
    // Setup All Serial
    ////////////////////

     // Setup pins for I2C
     ANSELCbits.ANSC4 = 0;
     ANSELCbits.ANSC5 = 0;

     TRISCbits.TRISC4 = 1;
     TRISCbits.TRISC5 = 1;

     PPSLOCK = 0x55;
     PPSLOCK = 0xAA;
     PPSLOCKbits.PPSLOCKED = 0x00; // unlock PPS

     RC4PPSbits.RC4PPS = 0x0011;   //RC4->MSSP:SDA;
     SSPDATPPSbits.SSPDATPPS = 0x0014;   //RC4->MSSP:SDA;
     SSPCLKPPSbits.SSPCLKPPS = 0x0015;   //RC5->MSSP:SCL;
     RC5PPSbits.RC5PPS = 0x0010;   //RC5->MSSP:SCL;

     // Setup pins for EUSART
     RB2PPSbits.RB2PPS = 0x14;   //RB2->EUSART:TX;
     RXPPSbits.RXPPS = 0x0B;   //RB3->EUSART:RX;
     PPSLOCK = 0x55;
     PPSLOCK = 0xAA;
     PPSLOCKbits.PPSLOCKED = 0x01; // lock PPS

    ////////////////////
    // Configure ADC
    ///////////////////

    // Fosc/32 ADC conversion time is 2.0 us
    ADCON1bits.ADCS = 0b010;

    // Right justified
    ADCON1bits.ADFM = 1;

    // Vref- is Vss
    ADCON1bits.ADNREF = 0;

    // Vref+ is Vdd
    ADCON1bits.ADPREF = 0b00;

    // Set input channel to AN0
    ADCON0bits.CHS = 0x05;

    // Zero ADRESL and ADRESH
    ADRESL = 0;
    ADRESH = 0;

    // Set E0 as ADC input channel5
    ANSELEbits.ANSE0 = 1;

    /////////////////////
    // Setup EUSART Pins
    /////////////////////

    // Setup PINS
    TRISBbits.TRISB3 = 1;
    ANSELBbits.ANSB3 = 0;

    TRISBbits.TRISB2 = 0;
    ANSELBbits.ANSB2 = 0;

    //////////////////////////////////
    // Configure watchdog timer
    //////////////////////////////////

    // Set watchdog timeout for 4 seconds
    WDTCONbits.WDTPS = 0b01100;
```

```
    TRISDbits.TRISD1 = 1;
    ANSELDbits.ANSD1 = 1;

    // Set PIN B0 as input
    TRISBbits.TRISB0 = 1;

    // Configure ANSELB0
    ANSELBbits.ANSB0 = 0;

    /////////////////////////
    /// Configure Interrupts
    ////////////////////////

    // unlock PPS
    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    PPSLOCK = 0x00;

    // Set Interrupt pin to pin B0
    INTPPSbits.INTPPS = 0b01000;

    // lock   PPS
    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    PPSLOCK = 0x01;

    // Trigger on falling edge
    OPTION_REGbits.INTEDG = 0;

    // Clear external interrupt flag
    INTCONbits.INTF = 0;

    //  Enable external interrupt
    INTCONbits.INTE = 1;

    // Enable global interrupt
    ei();

}

/************************************************************
 * Function: Main
 *
 * Returns: Nothing
 *
 * Description: Program entry point
 ************************************************************/
```

```
void main(void) {
    initMain();

    // Initialize I2C
    I2C_Init();
    __delay_ms(500);

    // Initialize OLED
    OLED_Init();

    // clear OLED
    OLED_Clear();

    __delay_ms(1000);

    CLRWDT();

    // Initialize EUSART
    EUSART_Initialize(9600);

    // Indicate start of server
    OLED_YX(0, 0);
    OLED_Write_String("START SERVER");
    __delay_ms(2000);

    CLRWDT();

    // Initialize the sever
    server_Initialize();

    // temperature variable
    float temp;

    while(1){

         // Clear OLED
         OLED_Clear();

         //////////////////////////////////
         // Read and Display temperature
         /////////////////////////////////
         temp = Read_Temperature();

         OLED_YX(0, 0);
         OLED_Write_String("Temperature:");
         OLED_YX(1, 0);
         OLED_Write_Integer(temp);
```

```
 __delay_ms(1000);
 OLED_Clear();

 //////////////////////////////////
 // Convert temperature to string
 /////////////////////////////////
 char* buff11;
 int status;

 buff11 = c_itoa(&status, (int)temp, 10);
 strcat(buff11, "\r\n");

 //////////////////////////////////
 // Wait for connection request
 //////////////////////////////////
 EUSART_Read_Text(buf, 20);

 //////////////////////////////////////
 // Display some of the received data
 /////////////////////////////////////
 OLED_YX(1, 0);
 OLED_Write_String(buf);
 __delay_ms(3000);
 OLED_Clear();

 //////////////////////////////////////////
 // Send the temperature as 2 bytes of data
 /////////////////////////////////////////
 OLED_YX(0, 0);
 OLED_Write_String("Sending Data");
 EUSART_Write_Text("AT+CIPSEND=0,2\r\n");
__delay_ms(5000);

 EUSART_Write_Text(buff11);

 EUSART_Read_Text(buf, 10);
 OLED_YX(1, 0);
 OLED_Write_String(buf);
 __delay_ms(3000);
 OLED_Clear();

 /////////////////////////
 // Close connection
 ////////////////////////

 EUSART_Write_Text("AT+CIPCLOSE=0\r\n");
 __delay_ms(1000);
```

```
        EUSART_Read_Text(buf, 10);
        OLED_YX(1, 0);
        OLED_Write_String(buf);
        __delay_ms(3000);
        OLED_Clear();

        // Reset EUSART
        RC1STAbits.SPEN = 0;
        RC1STAbits.SPEN = 1;

        // one this is complete clear watchdog
        CLRWDT();
    }

    return;
}

/*************************************************************
 * Function: void interrupt isr(void)
 *
 * Returns: Nothing
 *
 * Description: Interrupt triggered on pushbutton press
 *************************************************************/

void __interrupt() isr(void)
{
    // Clear interrupt flag
    INTCONbits.INTF = 0;

    // Set watchdog timeout for 4 seconds
    WDTCONbits.WDTPS = 0b01100;

    // Re-initialize server
    server_Initialize();
}

/*************************************************************
 * Function: void server_Initialize(void)
 *
 * Returns: Nothing
 *
 * Description: Sets up ESP8266 as a single connection server
on port 80
 ************************************************************/
float Read_Temperature()
{
     float conversion10;
     float farenheit;
     float celsius;
     float result;

    // Turn ADC on
      ADCON0bits.ADON = 1;

      // Sample CH0
      __delay_us(10);
      ADCON0bits.GO = 1;
      while (ADCON0bits.GO_nDONE);

      // Store ADC result
      result = ((ADRESH<<8)+ADRESL);

      // 10 bit conversion
      conversion10 = (result * 5000)/1024 ;

      // to Fahrenheit
      farenheit = conversion10 / 10;

      // to Celsius
      celsius = (farenheit - 32) * 5/9;

      return celsius;
}

/*************************************************************
 * Function: void server_Initialize(void)
 *
 * Returns: Nothing
 *
 * Description: Sets up ESP8266 as single connection server on
port 80
 *************************************************************/

void server_Initialize()
{

        ///////////////////////
        // Send AT Command
        ///////////////////////
         CLRWDT();
         OLED_YX(0, 0);
         OLED_Write_String("Sending AT");
         EUSART_Write_Text("AT\r\n");
         EUSART_Read_Text(buf, 11);

         OLED_YX(1, 0);
         OLED_Write_String(buf);
         __delay_ms(3000);
         OLED_Clear();

        ////////////////////////////////
        // Enable Single Connection
        ////////////////////////////////

         CLRWDT();
         OLED_YX(0, 0);
         OLED_Write_String("Sending CIPMUX");
         EUSART_Write_Text("AT+CIPMUX=0\r\n");
         EUSART_Read_Text(buf, 15);

        OLED_YX(1, 0);
        OLED_Write_String(buf);
        __delay_ms(3000);
        OLED_Clear();

        CLRWDT();

        //////////////////////////////////
        // Configure as server on port 80
        //////////////////////////////////
        OLED_YX(0, 0);
        OLED_Write_String("Sending CIPSERVER");
        EUSART_Write_Text("AT+CIPSERVER=1,80\r\n");

        EUSART_Read_Text(buf, 15);

        OLED_YX(1, 0);
        OLED_Write_String(buf);

        __delay_ms(3000);
        OLED_Clear();

        CLRWDT();

        // Set watchdog timeout for 128 seconds
        WDTCONbits.WDTPS = 0b10001;
}
```

Listing 12-1 Wi-Fi Logger Source Code

Our program is designed to cope with the limited memory of the PIC microcontroller. The watchdog timer is initially set with a four-second timeout, which is extended to 128 seconds once the server is operational, to ensure the system remains responsive and can recover from any errors by rebooting automatically if necessary.

After initialization, the program enters a loop where it reads the temperature, converts it to Celsius, displays the temperature on the OLED, and checks for incoming connections via the ESP8266. Upon receiving a connection request, the temperature data is formatted into a string and sent over Wi-Fi. The program uses AT commands to control the ESP8266, setting it up as a server on port 80 and allowing temperature data to be viewed in a web browser. This setup demonstrates the integration of various components and technologies—sensors, displays, and Wi-Fi modules—into a cohesive system capable of monitoring and sharing environmental data remotely.

The inclusion of an external interrupt, triggered by a pushbutton press, allows for the dynamic reconfiguration of the system, such as reinitializing the server settings, which demonstrates the program's ability to handle user inputs and system events seamlessly.

Bluetooth

Bluetooth is another wireless protocol we will examine. Bluetooth can replace wired communication between electronic devices with the attributes of low power consumption and low cost. These traits make it lucrative for the embedded systems designer since they go hand in hand with general embedded development. A few years ago, it would have been very expensive to add Bluetooth connectivity to your system, because the modules available to the average developer were relatively expensive. Thanks to its popularity, the cost of adding Bluetooth connectivity to a project has rapidly declined. The primary reason for this is due to the creation of low-cost Bluetooth modules by companies and manufacturers in the Chinese market. One such low-cost module is the HC05 Bluetooth module.

Expanding upon the introduction to Bluetooth as a pivotal wireless protocol, it's essential to delve deeper into how Bluetooth technology has evolved and its impact on the embedded systems landscape. Initially designed for short-range communication, Bluetooth has undergone numerous enhancements, broadening its use cases beyond simple device-to-device connectivity. The introduction of Bluetooth Low Energy (BLE) with version 4.0 marked a significant turning point, offering even lower power consumption and making it ideal for battery-powered or energy-harvesting devices. This innovation opened the door to a myriad of applications in health monitoring, home automation, and wearable technology, where power efficiency is paramount.

Moreover, the development of Bluetooth 5 introduced improvements in speed and range, alongside the capacity for broadcasting to multiple devices, further cementing its role in the burgeoning field of the Internet of Things (IoT). This leap in technology facilitated the creation of more sophisticated and interconnected systems, enabling devices to communicate over greater distances and with improved reliability. The seamless connectivity and the ability to exchange data across a mesh network have significantly enhanced user experiences, allowing for more intuitive and responsive smart environments.

Furthermore, the widespread adoption of Bluetooth technology has spurred a vibrant ecosystem of developers and innovators, eager to explore its potential. Open source projects and community-driven platforms have lowered the barrier to entry, empowering hobbyists and professionals alike to prototype and develop Bluetooth-enabled solutions with ease. The support from this community, coupled with the availability of comprehensive development kits and tools, has accelerated the growth of Bluetooth applications.

Using the HC05

Using the HC05 is very simple. You simply connect the RX and TX pins of the module to the microcontroller through a logic-level converter. The VCC and GND pins of the HC05 are connected to 5v and GND, respectively. The code for receiving commands for the HC05 is very simple. We will use this code to toggle an LED. When the ONPR command is sent to the module, the microcontroller turns the LED on. When OFFPR is sent to the module, the microcontroller turns the LED off. We will use the C needle in a haystack command, called strstr, to search the received command for ON or OFF. The commands can be sent from a PC or a mobile device. On a PC or Android, when pairing the device, enter the default passcode 1234. On a PC, the device shows up as a COM port. I recommend the Termite program by CompuPhase to communicate via Bluetooth. On Android, there are a lot of Bluetooth terminals; however, I recommend an app named Bluetooth Terminal HC05, which works quite well and has preset buttons to make things simple. This module does not work with iOS.

```
/*
 * File: Main.c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/Int OSC @ 16MHz, 5v
 * Program: 49_Bluetooth
 * Compiler: XC8 (v2.45, MPLAX X v6.45)
 * Program Version: 1.1
 *
 *
 * Program Description: This Program Allows PIC16F1719 to
communicate via Bluetooth
 *
 *
 * Hardware Description: A HC-05 is connected to the PIC16F1719
as follows:
 *
 *                       RX -> RB3
 *                       TX -> RB2
 *
 * Created May 15th, 2017, 5:00 PM
 * Updated March 20th, 2024, 3:16 AM
 */
```

```
/************************************************************
 *Includes and defines
 ************************************************************/

#include "PIC16F1719_Internal.h"
#include "EUSART.h"
#include <string.h>

#define LED LATDbits.LATD1

/************************************************************
 * Function: void initMain()
 *
 * Returns: Nothing
 *
 * Description: Contains initializations for main
 *
 * Usage: initMain()
 ************************************************************/

void initMain(){
    // Run at 16 MHz
    internal_16();

    // Set PIN D1 as output
    TRISDbits.TRISD1 = 0;

    // Turn off LED
    LATDbits.LATD1 = 0;

    // Setup PORTD
    TRISD = 0;
    ANSELD = 0;

    // Setup pins for EUSART
    TRISBbits.TRISB2 = 0;
    ANSELBbits.ANSB2 = 0;

    TRISBbits.TRISB3 = 1;
    ANSELBbits.ANSB3 = 0;

    /////////////////////
    // Setup EUSART
    ////////////////////
    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    PPSLOCKbits.PPSLOCKED = 0x00; // unlock PPS
    RB2PPSbits.RB2PPS = 0x14;   //RB2->EUSART:TX;
    RXPPSbits.RXPPS = 0x0B;   //RB3->EUSART:RX;

    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    PPSLOCKbits.PPSLOCKED = 0x01; // lock PPS

}

/**************************************************************
 * Function: Main
 *
 * Returns: Nothing
 *
 * Description: Program entry point
 **************************************************************/

void main(void) {
    initMain();

    // Initialize EUSART module with 9600 baud
    EUSART_Initialize(9600);

    char buf[20];
    char* ON;
    char* OFF;

    while(1){

        // Send start so we'll know it's working
        EUSART_Write_Text("Start");

        // Read UART messages
        EUSART_Read_Text(buf, 4);

        // Test received string
        ON =  strstr(buf, "ON");
        OFF = strstr(buf, "OFF");

        // If ON string, turn LED on
        if (ON)
        {
          EUSART_Write_Text("LED ON");
          LED = 1;
        }

        // If OFF string, turn LED off
        else if(OFF)
        {
          EUSART_Write_Text("LED OFF");
          LED = 0;
        }
    }

    return;

}
```

In our program, once the initial setup is completed, the program enters an infinite loop where it first sends out a “Start” message via EUSART to signal that it’s ready to receive commands. It then listens for incoming data strings. The program specifically looks for the strings “ON” and “OFF” within the messages it receives. When “ON” is detected, the microcontroller responds by turning the LED on and sending a confirmation message “LED ON” back over Bluetooth. Conversely, if “OFF” is detected, the LED is turned off, and a “LED OFF” message is sent as confirmation.

Conclusion

This chapter looked at using Bluetooth and Wi-Fi and using the PIC microcontroller with the ESP8266. Bluetooth and Wi-Fi are arguably two of the most important wireless protocols available today. The information presented in this chapter was just enough so that you will be able to add these protocols to your systems.

A. Subero, *Programming PIC Microcontrollers with XC8*, Maker Innovations Series
https://doi.org/10.1007/979-8-8688-0467-0_13

13. Watchdog Timer and Low Power

Armstrong Subero[1]

(1) Moruga, Trinidad and Tobago

In the embedded systems domain, achieving an optimal balance between operational reliability and power efficiency is paramount. This chapter delves into two pivotal technologies that stand at the forefront of this balance: the watchdog timer and low-power modes. Both elements are instrumental in creating systems that not only conserve energy but also maintain robust performance and reliability over time. This chapter aims to explore the intricacies of implementing watchdog timers and leveraging low-power modes within PIC microcontrollers. By understanding these concepts, developers can design systems that not only optimize energy usage but also enhance reliability and longevity. Through practical examples and theoretical insights, we will uncover the strategies to harness these technologies effectively, paving the way for the creation of sophisticated, energy-conscious, and dependable embedded systems.

Low Power 8 vs. 32-Bit

It's nearly impossible to write a chapter on power consumption without comparing 8-bit and 32-bit architectures. The reason is that since ARM Cortex cores have become so prevalent, many designers opt to utilize 32-bit microcontrollers in place of 8-bit microcontrollers. The distinction between 8-bit and 32-bit microcontroller architectures is crucial in understanding their performance, capabilities, and power consumption characteristics. This differentiation primarily revolves around the width of the data path, registers, and the memory bus of the microcontroller, which fundamentally influences its computational speed, efficiency, and energy usage. Once you understand these facets, you'll better be able to select which one you need for your designs. Of course, since this is a book about 8-bit devices, I am inclined to say that 8-bit devices will meet all your requirements for all low-power applications.

These 8-bit microcontrollers, characterized by their simpler architecture, have been the backbone of many low-power and cost-sensitive applications for decades. The "8-bit" designation implies that these microcontrollers process data in 8-bit chunks, meaning that their data bus and registers are 8 bits wide. This architecture is particularly well suited to applications with straightforward control tasks, such as sensor reading, basic device control, and simple communication interfaces. If you had a weekend project to do, as we have seen throughout this book, it would be far easier to select and use an 8-bit device, since in general, they require less time to configure.

One of the main advantages of 8-bit microcontrollers is their low power consumption. The simplicity of their design allows them to operate with minimal energy requirements, which is essential for battery-powered or energy-harvesting devices. Additionally, the limited processing capabilities inherently reduce the amount of power drawn during operation, as they typically run at lower clock speeds and have fewer complex operations to perform. This simplicity of design also has an inherent value. Without much trouble, even an inexperienced developer can know what every part of the core is doing without too much hassle. As cores become more complex, it becomes harder to understand what the device is doing at any given time.

In contrast, 32-bit microcontrollers offer significantly more processing power and memory addressability, capable of handling complex algorithms, high-speed data processing, and multitasking more efficiently. The “32-bit” architecture means that these microcontrollers can process 32 bits of data at once, making them faster and more capable of managing large amounts of data or running complex applications. If you’re doing graphical user interfaces, detailed sensor data analysis, and connectivity stacks for IoT devices, a 32-bit device is generally recommended as a first choice. However, this increased capability comes at the cost of higher power consumption. The larger data path and registers, higher clock speeds, and greater complexity of the tasks they can perform mean that 32-bit microcontrollers typically consume more power than their 8-bit counterparts. Despite this, advances in semiconductor technology and power management techniques have enabled 32-bit microcontrollers to achieve remarkable levels of power efficiency, narrowing the gap in power consumption between the two architectures.

Power consumption in microcontrollers is influenced by several factors, including clock speed, operating voltage, active vs. sleep mode power draw, and the efficiency of the code being executed. While 8-bit microcontrollers may inherently consume less power due to their simpler architecture and lower processing capabilities, 32-bit microcontrollers can leverage advanced power-saving modes, dynamic voltage scaling, and other techniques to minimize energy usage during idle or low-activity periods. Additionally, the increased processing power of 32-bit architectures can allow a task to be completed more quickly and the microcontroller to return to a low-power sleep mode sooner, potentially offsetting the higher power usage during active periods. This is not to be taken lightly, as a few instructions on a high-performance 32-bit core may require dozens or even hundreds of instructions on an 8-bit device, eliminating the power-saving capability that is inherent to the architecture core.

My recommendation is that for applications that require minimal processing and prioritize long battery life, an 8-bit microcontroller might be the most efficient choice. Conversely, for more complex applications demanding higher computational throughput and advanced functionalities, the benefits of a 32-bit microcontroller, along with its power management features, might outweigh the inherent power consumption increase, offering a more capable and flexible solution.

Sleep Mode

The last section began our discussion of power consumption in 8-bit and 32-bit microcontroller cores. We can delve more into that topic by shifting our attention to sleep mode on microcontrollers. Sleep mode in microcontrollers, including those in the PIC family, is a critical feature designed to conserve power, making it essential for battery-operated and low-power applications. When a microcontroller enters sleep mode, it significantly reduces its power consumption by shutting down unnecessary subsystems while retaining the state of registers and certain peripherals. This ability is particularly valuable in applications where the device operates intermittently, waking up only to perform a task or respond to an external event, such as a sensor input or a timed interrupt. For instance, in remote sensing applications or devices that need to operate over extended periods on a single battery charge, leveraging sleep mode can dramatically extend battery life.

The PIC microcontrollers have a sleep mode that allows developers to balance between the lowest possible power consumption and the need for the microcontroller to remain responsive to external events. When a PIC microcontroller enters sleep mode, its CPU is turned off, and the clock oscillators can be stopped, drastically reducing power usage. However, certain peripherals can be configured to remain active, such as timers, watchdog timers, and of course, the core independent peripherals (CIPs), enabling the microcontroller to wake up in response to specific triggers. This nuanced control over power consumption vs. responsiveness is a key advantage of PIC microcontrollers in power-sensitive applications.

In the context of the PIC16F1719, sleep mode is particularly easy to use. This microcontroller can enter sleep mode through a simple software instruction, after which it halts the CPU and most internal clocks, transitioning to a state where power consumption is minimized. The PIC16F1719's sleep mode is designed to retain the contents of registers and certain peripheral configurations, allowing the device to resume operation without the need for a complete restart. As developers, we can tailor the wake-up conditions, such as an interrupt from an external event, a change in state on a selected pin, or a timeout from the watchdog timer, to suit their application's requirements.

Moreover, the integration of the watchdog timer with sleep mode in the PIC16F1719 exemplifies the microcontroller's capabilities for self-reliant operation. The watchdog timer can be configured to wake the microcontroller from sleep mode at predetermined intervals (we'll look at this in the next section). This ensures that the device can perform routine checks, maintenance tasks, or updates without continuous CPU activity. This feature is invaluable in applications where periodic monitoring is required, but constant operation is unnecessary or impractical, such as in environmental monitoring stations or smart sensors. We can see how the sleep mode works in Listing 13-1.

```
/*
 * File: main.c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/int OSC @ 32MHz, 5v
 * Program: 07_Sleep
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 * Program Version: 1.2
 *
 * Program Description: This Program Allows PIC16F1719 to
demonstrate the use of sleep Mode
 *
 * Hardware Description: An LED is connected via a 1K resistor
to PIN D1 and a switch is connected to PIN B0
 *
 * Created November 4th, 2016, 1:00 PM
 * Last Updated: November 18th, 2023, 1:57 AM
 */

/**************************************************************
 Change History:
 Revision     Description
 v1.2         Updated and recompiled with MPLABX v6.15 using
XC8 v2.45
 v1.1         Changed from PIC16F1717 to PIC16F1719 and
recompiled with
              MPLABX v5.15 using v2.05 of the XC8 compiler
**************************************************************/

/**************************************************************
 *Includes and defines
 **************************************************************/

#include "PIC16F1719_Internal.h"

/**************************************************************
 * Function: void initMain()
 *
 * Returns: Nothing
 *
 * Description: Contains initializations for main
 *
 * Usage: initMain()
 **************************************************************/

void initMain(){
    //////////////////////
    // Configure Ports
    /////////////////////

    // Run at 32 MHz
    internal_32();

    // Set PIN D1 as output
    TRISDbits.TRISD1 = 0;

    // Turn off LED
    LATDbits.LATD1 = 0;

    // Set PIN B0 as input
    TRISBbits.TRISB0 = 1;

    // Configure ANSELB0
    ANSELBbits.ANSB0 = 0;

    /////////////////////////
    /// Configure Interrupts
    ////////////////////////

    // Set Interrupt pin to pin B0
    INTPPSbits.INTPPS = 0b01000;

    // lock   PPS
    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    PPSLOCK = 0x01;

    // Trigger on the falling edge
    OPTION_REGbits.INTEDG = 0;

    // Clear external interrupt flag
    INTCONbits.INTF = 0;

    //  Enable external interrupt
    INTCONbits.INTE = 1;

    // Enable global interrupt
    ei();
}

/*************************************************************
 * Function: Main
 *
 * Returns: Nothing
 *
 * Description: Program entry point
 *************************************************************/

void main(void) {
    initMain();
    uint8_t i= 0;

    while(1){

        // Blink LED
        for(i=0; i!=10; i++)
        {
            // Toggle LED Free running
            LATDbits.LATD1 = ~LATDbits.LATD1;

            __delay_ms(250);
        }

        // set LED off
        LATDbits.LATD1 = 0;

        // Enable sleep mode
        SLEEP();
    }

    return;
}

/**************************************************************
 * Function: void interrupt isr(void)
 *
 * Returns: Nothing
 *
 * Description: Use Timer0 External Interrupt
 *************************************************************/
void __interrupt() isr(void)
{
    // Clear interrupt flag
    INTCONbits.INTF = 0;

    // Toggle led
    LATDbits.LATD1 = 1;
    __delay_ms(3000);
}
```

Listing 13-1 Using Sleep Mode on the PIC16F1719

Within our main function, after our usual initialization, it enters a loop where it blinks the LED connected to PIN D1 ten times with a 250 ms delay between toggles. After completing the blinking sequence, the LED is explicitly turned off, and the microcontroller is put into sleep mode using the SLEEP() macro. In this state, the microcontroller's power consumption is significantly reduced until an external event, such as a button press, triggers an interrupt causing the microcontroller to wake up.

The interrupt service routine (ISR), void __interrupt() isr(void), is defined to handle the external interrupt caused by a change on PIN B0 (the button press). Upon entering the ISR, the interrupt flag is cleared first to acknowledge and reset the interrupt condition. Then, the LED on PIN D1 is turned on for three seconds, indicating that the microcontroller has successfully exited sleep mode in response to the interrupt before it will potentially return to sleep mode again if the main loop execution continues as before. If you like, you can measure the power consumption using your multimeter to see the difference in power consumption.

Watchdog Timer

Watchdog timers play a crucial role in ensuring the reliability and robustness of embedded systems. Essentially, a watchdog timer is a hardware or software timer that, when enabled, must be regularly reset before it counts down to zero. If the system fails to reset the timer—typically because of a software hang or a failure to execute tasks within expected time frames—the timer expires and automatically triggers a system reset or other predefined recovery actions. This mechanism serves as a safeguard against system malfunctions that can arise from software errors, hardware faults, or unexpected operational conditions. By monitoring the system's ability to perform regular resets, the watchdog timer can detect and respond to system anomalies that might otherwise lead to prolonged failures or erratic behaviors.

In the context of embedded systems, watchdog timers are particularly valuable for applications that require high availability, reliability, or safety. These systems often operate in unattended or remote environments, where manual intervention to reset or repair a malfunctioning device is impractical or impossible. For instance, in industrial control systems, automotive electronics, and telecommunications equipment, the use of watchdog timers helps maintain operational continuity and prevents minor software glitches from escalating into catastrophic failures. The watchdog timer's ability to automatically reset the system ensures that it can attempt to resume normal operation swiftly, minimizing downtime and potentially avoiding dangerous situations. You can also think about a situation where a Mars rover or satellite in space where human interaction is non-existent the watchdog timer can ensure system reliability.

Moreover, the implementation of watchdog timers requires careful consideration in the system design phase. Developers must strategically place reset commands within the software to ensure that the timer is regularly reset under normal operation but not in a manner that masks underlying issues. This often involves structuring the software to confirm the completion of critical tasks or checkpoints before resetting the timer, thereby ensuring that the system is genuinely performing as expected. In addition to automatic system resets, advanced watchdog timer configurations may offer features such as logging the cause of the reset or executing custom recovery procedures, providing valuable insights for debugging and improving system resilience. As embedded systems continue to proliferate across various sectors, the importance of watchdog timers as a fundamental component for enhancing system reliability and safety cannot be overstated.

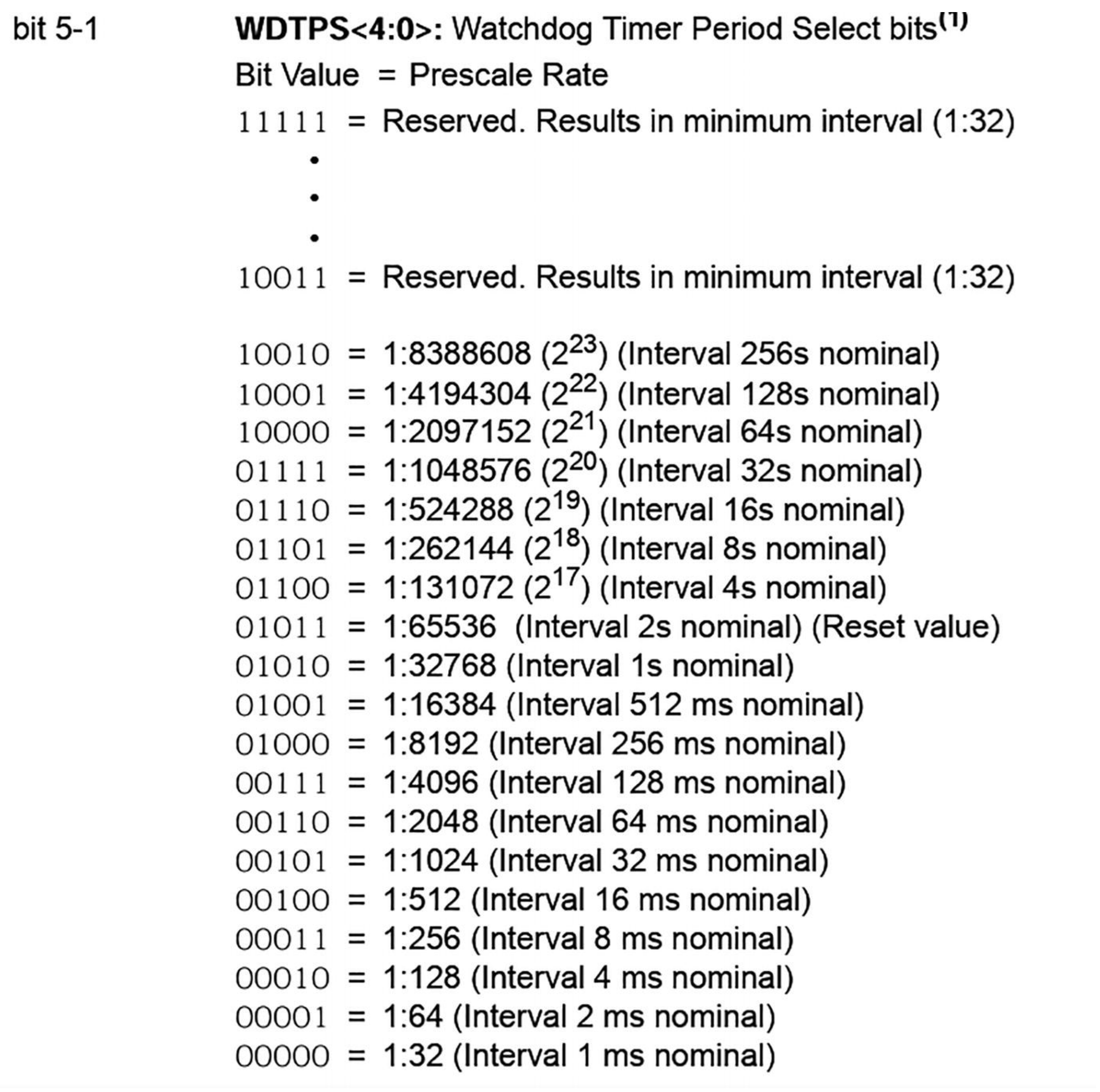
In PIC microcontrollers, watchdog timers are integrated as a standard feature. The WDT on PIC microcontrollers is designed to automatically reset the microcontroller if the software becomes unresponsive or deviates from its expected operational flow, just like any other generic watchdog timer. This is particularly useful in applications where continuous operation is critical and manual reset is not feasible. To prevent the WDT from triggering a reset, the software must periodically clear or "kick" the timer. This action is typically performed within the main program loop or in parts of the code that signify normal operation. This mechanism ensures that the microcontroller can recover from transient errors or unexpected conditions, such as infinite loops or execution stalls caused by unforeseen hardware interactions.

Configuring and using the watchdog timer on PIC microcontrollers involves setting various parameters, such as the timer's prescaler, to adjust the timeout period according to the application's requirements. Moreover, the PIC microcontroller's WDT is often complemented by other safety features, such as brown-out reset and power-on reset circuits, which we discussed in our chapter on PIC microcontrollers. When deploying watchdog timers in PIC microcontroller–based projects, developers must carefully plan their software architecture to ensure that the watchdog is reset under normal conditions but not in a way that overlooks genuine system faults. In Listing 13-2, we have a program that uses the watchdog timer.

```
/*
 * File: main.c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/int OSC @ 32MHz, 5v
 * Program: 08_Watchdog_Timer
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 * Program Version: 1.2
 *
 * Program Description: This Program Allows PIC16F1719 to
demonstrate the use of the watchdog timer. The watchdog timer
toggles the LED every 16 seconds and then goes to sleep. The
CPU is then awoken by a WDT timeout CLRWDT() macro can clear
the watchdog timer
 *
 * Hardware Description: An LED is connected via a 1K resistor
to PIN D1 and a switch is connected to PIN B0
 *
 * Created November 4th, 2016, 1:00 PM
 * Last Updated: November 18th, 2023, 2:05 AM
 */

/**************************************************************
 *Includes and defines
 **************************************************************/
#include "PIC16F1719_Internal.h"

/*************************************************************
 * Function: void initMain()
 *
 * Returns: Nothing
 *
 * Description: Contains initializations for main
 *
 * Usage: initMain()
 ***************************************************************/

void initMain(){
    // Run at 32 MHz
    internal_32();

    // Set PIN D1 as output
    TRISDbits.TRISD1 = 0;

    // Turn off LED
    LATDbits.LATD1 = 0;

    // Set PIN B0 as input
    TRISBbits.TRISB0 = 1;

    // Configure ANSELB0
    ANSELBbits.ANSB0 = 0;

    ////////////////////////////
    // Configure Watchdog timer
    ////////////////////////////
    // Set WDT period to 16s
    WDTCONbits.WDTPS = 0b01110;
}

/**************************************************************
 * Function: Main
 *
 * Returns: Nothing
 *
 * Description: Program entry point
 **************************************************************/

void main(void) {
    initMain();

    while(1){
        // flash LED
        LATDbits.LATD1 = 1;
        __delay_ms(3000);
        LATDbits.LATD1 = 0;
        // Sleep
        SLEEP();
    }

    return;
}
```

Listing 13-2 The Watchdog Timer

In our program, we set the watchdog timer period to 16 seconds. Within the main program, all we have to do is sleep. We set the watchdog timer period based on the watchdog timer control register. In case you are wondering where we got that value to set, if we look at Figure 13-1, we see the bit values for the watchdog timer period.



bit 5-1 **WDTPS<4:0>:** Watchdog Timer Period Select bits(1)

Bit Value		Prescale Rate
11111	=	Reserved. Results in minimum interval (1:32)
•		
•		
•		
10011	=	Reserved. Results in minimum interval (1:32)
10010	=	1:8388608 (2^{23}) (Interval 256s nominal)
10001	=	1:4194304 (2^{22}) (Interval 128s nominal)
10000	=	1:2097152 (2^{21}) (Interval 64s nominal)
01111	=	1:1048576 (2^{20}) (Interval 32s nominal)
01110	=	1:524288 (2^{19}) (Interval 16s nominal)
01101	=	1:262144 (2^{18}) (Interval 8s nominal)
01100	=	1:131072 (2^{17}) (Interval 4s nominal)
01011	=	1:65536 (Interval 2s nominal) (Reset value)
01010	=	1:32768 (Interval 1s nominal)
01001	=	1:16384 (Interval 512 ms nominal)
01000	=	1:8192 (Interval 256 ms nominal)
00111	=	1:4096 (Interval 128 ms nominal)
00110	=	1:2048 (Interval 64 ms nominal)
00101	=	1:1024 (Interval 32 ms nominal)
00100	=	1:512 (Interval 16 ms nominal)
00011	=	1:256 (Interval 8 ms nominal)
00010	=	1:128 (Interval 4 ms nominal)
00001	=	1:64 (Interval 2 ms nominal)
00000	=	1:32 (Interval 1 ms nominal)

Figure 13-1 Watchdog timer period selection bits (reprinted with permission)

To ensure that the program works as intended, you need to set the watchdog timer bits in the configuration bit settings. In the PIC16F1719_Internal.h file, the "CONFIG1" section should have these settings. The snippet is given in Listing 13-3.

```
// CONFIG1
#pragma config FOSC = INTOSC    // Oscillator Selection Bits
(INTOSC oscillator: I/O function on CLKIN pin)

#pragma config WDTE = ON        // Watchdog Timer Enable (WDT
enabled)

#pragma config PWRTE = OFF      // Power-up Timer Enable (PWRT
disabled)
#pragma config MCLRE = OFF      // MCLR Pin Function Select
(MCLR/VPP pin function is MCLR)
#pragma config CP = OFF         // Flash Program Memory Code
Protection (Program memory code protection is disabled)
#pragma config BOREN = OFF      // Brown-out Reset Enable
(Brown-out Reset disabled)
#pragma config CLKOUTEN = OFF   // Clock Out Enable (CLKOUT
function is disabled. I/O or oscillator function on the CLKOUT
pin)
#pragma config IESO = ON        // Internal/External
Switchover Mode (Internal/External Switchover Mode is enabled)
#pragma config FCMEN = OFF      // Fail-Safe Clock Monitor
Enable (Fail-Safe Clock Monitor is enabled)
```

Listing 13-3 Ensuring the Watchdog Timer Is On in the Configuration Bits

Once the watchdog timer is enabled, you can set any period you want and develop very robust applications.

Other Ways to Conserve Power

In the previous section, we looked at sleep mode as one of the ways we can conserve power in microcontrollers. There are other ways we can conserve power as well, which we will explore in this section. These methods are those that I used over the years in PIC microcontroller projects.

Reduce the Clock Frequency

Lowering the clock frequency of the microcontroller, such as operating at 31 kHz, directly reduces power consumption by decreasing the amount of energy used for each operation. This is effective for applications with minimal computational demands. The challenge lies in balancing the need for processing speed against power savings, as running at too low a frequency could lead to increased operational time and greater overall energy use.

Reduce the Operating Voltage

Operating the microcontroller at a lower voltage, such as shifting from 5 V to 3.3 V, significantly decreases power consumption. This method requires careful consideration of the entire system's voltage requirements, especially if certain components necessitate higher voltages, which might necessitate level shifters or logic-level converters. In modern designs, it's not much of an issue as many peripherals are only available in 3.3 V versions.

Power External Devices from I/O Lines

Utilizing I/O lines to power low-consumption external devices can lead to power savings by simplifying the power management scheme. This approach must be balanced with the current handling capabilities of the microcontroller's I/O pins to prevent damage.

Utilize Peripheral Shutdown Features

Many PIC microcontrollers offer the ability to disable unused peripherals, such as ADCs, UARTs, or SPI modules. Turning off these peripherals when they're not in use can significantly reduce power consumption.

Implement Interrupt-Driven Programming

Instead of using polling, which keeps the CPU active, use interrupt-driven programming. This approach allows the CPU to enter sleep mode and only wake up when necessary, drastically reducing power usage.

Optimize Firmware Algorithms

Efficient code can reduce CPU load and operational time. Optimization techniques include using efficient data types, minimizing loop iterations, and avoiding unnecessary peripheral accesses.

Take Advantage of Brown-Out Detect (BOD) Disable

For applications where precise voltage monitoring isn't critical, disabling the BOD can save power. The BOD consumes power by continuously monitoring the supply voltage, so turning it off when not needed can be beneficial.

Use Power-Saving Modes Wisely

Beyond sleep mode, PIC microcontrollers may offer other low-power states like idle mode. Understanding and utilizing these modes based on the application's requirements can lead to substantial power savings.

Minimize External Peripheral Power

Choose low-power external components and ensure they're only powered when necessary. For instance, sensors or communication modules can be powered down when not in active use.

Capacitive Touch Sensing with Low Power

For interfaces requiring user input, consider using capacitive touch sensing, which can be implemented to be highly energy efficient, avoiding the need for power-intensive mechanical switches or buttons.

Dynamic Clock Switching

For applications that vary in processing demand, dynamically switching the clock speed can save power. Operate at higher speeds only during computationally intensive tasks and reduce the clock speed for routine tasks. This works well for PIC microcontrollers because their clock speed can easily be configured on the fly.

Optimize ADC Usage

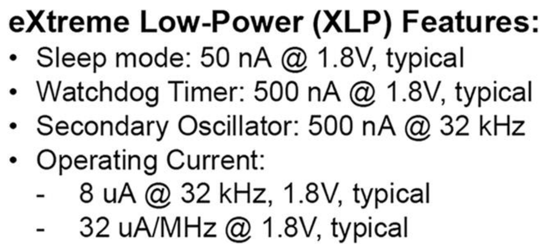
Analog-to-digital converters (ADCs) are power-hungry components. Use them at the lowest possible resolution and sampling rate that your application can tolerate, and turn them off when not in use.

These strategies, when applied thoughtfully, can lead to significant energy savings in PIC microcontroller–based projects, extending battery life and reducing energy costs in embedded systems. Balancing performance with power efficiency requires careful planning and testing but can result in highly efficient and effective applications.

---

eXtreme Low Power (XLP) Technology

A lot of the techniques for conserving power are integrated into a technology Microchip called "eXtreme Low Power (XLP) technology." This innovative technology is specifically engineered to meet the rigorous demands of modern embedded applications that require minimal power consumption without compromising performance. The nice thing about XLP technology is that many of its features are inherent to the architecture. In the PIC16F1719, the XLP features of the device are given in Figure 13-2.



eXtreme Low-Power (XLP) Features:

- Sleep mode: 50 nA @ 1.8V, typical
- Watchdog Timer: 500 nA @ 1.8V, typical
- Secondary Oscillator: 500 nA @ 32 kHz
- Operating Current:
  - 8 uA @ 32 kHz, 1.8V, typical
  - 32 uA/MHz @ 1.8V, typical

Figure 13-2 PIC16F1719 XLP features

XLP technology achieves exceptional energy efficiency through various advanced features, including multiple sleep modes for reducing power consumption to nanoamp levels, dynamic clock switching to adjust power usage according to operational requirements, and low-power peripherals designed to operate at reduced energy levels. These capabilities enable developers to design and implement applications ranging from portable medical devices to environmental monitoring sensors, where power efficiency is paramount, extending operational life and reducing the ecological footprint of electronic devices.

Furthermore, XLP technology enhances the PIC microcontroller's versatility in power-sensitive applications through its ability to maintain system responsiveness and reliability under stringent power constraints. It facilitates the creation of devices that can operate for years on a single battery charge, a critical advantage in remote sensing applications, wearable technology, and IoT devices where frequent battery replacement is impractical or impossible. By integrating XLP technology, PIC microcontrollers offer an optimal solution for developers looking to balance power efficiency with computational needs, ensuring that devices remain active and functional only as necessary, thereby conserving precious energy resources. This approach not only addresses the immediate needs of power-sensitive applications but also aligns with broader environmental sustainability goals, marking a significant step forward in the evolution of microcontroller technology.

Conclusion

In this chapter, we looked at using the watchdog timer and using the sleep mode on the microcontroller. We also looked at specific ways to reduce power consumption. We also discussed XLP technology.

A. Subero, *Programming PIC Microcontrollers with XC8*, Maker Innovations Series
https://doi.org/10.1007/979-8-8688-0467-0_14

14. PIC Microcontroller Projects

Armstrong Subero[1]

(1) Moruga, Trinidad and Tobago

In this chapter, we explore the practical application of PIC microcontrollers through the development of two engaging projects. The initial project we tackle is a staple in the microcontroller realm—a temperature-controlled fan. This project not only serves as an excellent introduction to the fundamentals of microcontroller programming and circuit design but also provides a hands-on experience with sensors and actuators. The second project we undertake is more communication focused, where we demonstrate how to transmit data directly to a web browser using the serial port of the microcontroller. This project highlights the capabilities of microcontrollers in networked applications and offers a fascinating glimpse into the interface between hardware and web technology. Together, these projects will enhance your understanding of how microcontrollers can be applied in both environmental control and data communication scenarios.

Project: Temperature-Controlled Fan

The first project involves building a temperature-controlled fan. The LM34 temperature sensor output is converted to Celsius values, and when a particular threshold voltage is reached, the microcontroller turns on the fan. Once the temperature is within the normal values, the string "Temp OK" is displayed on the OLED. When the temperature crosses a certain value, the string "Warning!!" is displayed on the OLED. The way we can do this project can either be on-off control or PWM-based control for the fan.

On-Off Control

On-off control is one of the simplest forms of feedback control used in various applications to maintain a desired setpoint within a system. At its core, the on-off control strategy operates by simply toggling the output device—such as a heater, fan, or light—between its on and off states. This decision is based on the comparison between the measured variable, like temperature, and a predefined threshold or setpoint. When the process variable exceeds the setpoint, the controller turns the output device off, and when it falls below the setpoint, the controller turns it on. This method is widely employed due to its simplicity and ease of implementation in both mechanical and electronic systems.

Despite its simplicity, on-off control is remarkably effective for many applications where precise control over a process variable is unnecessary. It is commonly found in household appliances such as thermostats, refrigerators, and ovens, where it regulates temperature, and in sump pumps, where it controls water levels. The main advantage of on-off control is its straightforwardness, making it not only easy to understand and implement but also cost-effective. However, to mitigate the issue of excessive switching near the setpoint—a phenomenon known as chattering—systems often incorporate a hysteresis or deadband. This introduces a small range around the setpoint within which no action is taken, reducing wear and tear on the control device and preventing rapid oscillations.

However, on-off control systems are not without their limitations, particularly when it comes to applications requiring high precision or where the inertia of the system is significant. In such scenarios, the simplicity of on-off control can lead to oscillations around the setpoint and potentially unstable operation if the system cannot respond quickly enough. Additionally, the use of hysteresis to prevent chattering can result in a steady-state error from the setpoint, as the process variable stabilizes within the deadband rather than at the precise setpoint. Despite these drawbacks, the widespread use of on-off control in both industrial and domestic applications underscores its utility in scenarios where its limitations are outweighed by its advantages in terms of simplicity, reliability, and cost.

On-Off vs. PWM-Based Control

When comparing on-off fan control to PWM (Pulse Width Modulation)-based control, each method presents distinct advantages and operational characteristics that make them suitable for different applications.

On-off control is the simpler of the two methods, functioning by switching the fan on when the temperature exceeds a certain threshold and off when the temperature drops below this threshold. This approach is straightforward to implement and understand, making it a popular choice for basic cooling needs where precise temperature management is not critical. Its simplicity also contributes to lower cost and reliability since fewer components can fail. However, the downside of on-off control includes the potential for temperature fluctuations around the setpoint due to the binary nature of the control method. Additionally, frequent switching can lead to mechanical wear and tear over time, potentially reducing the fan's lifespan.

In contrast, PWM-based control offers a more sophisticated means of managing fan speed by varying the duty cycle of an electrical signal, effectively continuously controlling the fan speed. This method allows for finer temperature control, as the fan can operate at various speeds rather than simply being on or off. Such granularity reduces temperature oscillations around the setpoint and can improve the system's overall energy efficiency, as the fan can run at lower speeds when full power is unnecessary. Moreover, PWM control can contribute to quieter operation and longer fan lifespan, as it avoids the constant on-off cycling that can strain mechanical components. However, PWM control systems are generally more complex and costly to implement than on-off controls, requiring additional components and programming to manage the PWM signals.

Ultimately, the choice between on-off and PWM fan control depends on the specific requirements of the application. On-off control is well suited to simpler, cost-sensitive applications where the drawbacks of temperature fluctuation and the potential for increased mechanical wear are acceptable trade-offs. On the other hand, PWM control is preferred in scenarios where precise temperature control, efficiency, and reduced mechanical wear are priorities, justifying the higher complexity and cost of the control system.

In our own temperature-controlled fan project, we use PWM-based control. Figure 14-1 shows the schematic of the temperature-controlled fan project.

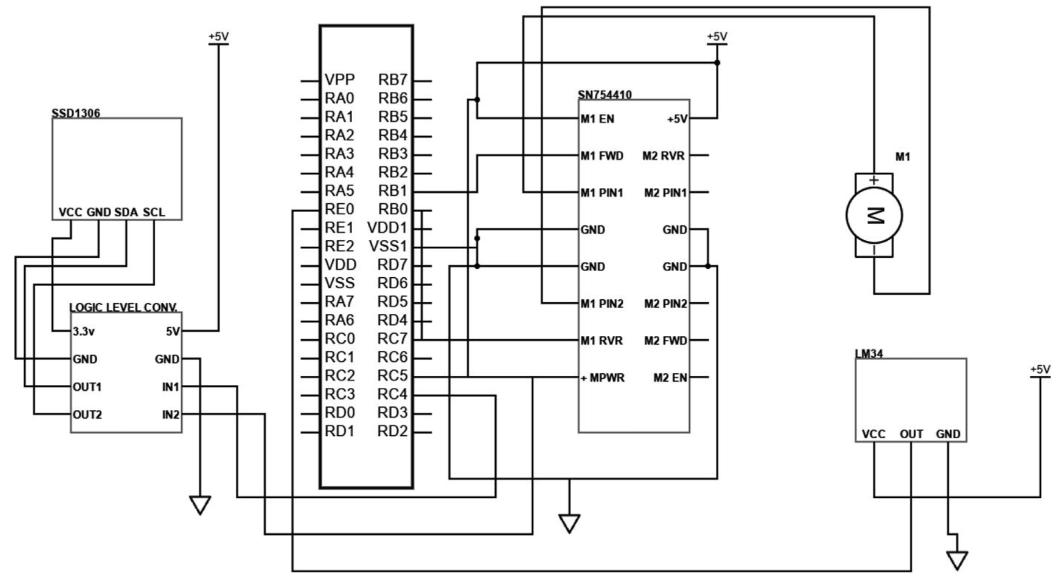


Figure 14-1 Temperature-controlled fan project

Once we have our hardware set up, we can write our program. The code is given in Listing 14-1.

```
/*
 * File: Main.c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/Internal OSC @ 16MHz, 5v
 * Program: 50_Temperature_Controlled_Fan
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 * Program Version: 1.0
 *
 * Program Description: This project builds a temperature controlled
fan. When the temperature rises above 35 Celsius a fan turns on until
the temperature drops to 35 Celsius or below.
 *
 * Hardware Description: A generic brushed hobby DC motor is
connected to the SN754410 as per standard connections. The PWM
signals are emanating from RB0 and RB1. The LM34 temperature sensor
is connected to PIN RE0 and an SSD1306 based OLED is connected as per
header file.
 *
 * Created April 18th, 2017, 4:36 PM
 * Updated March 27th, 2024, 3:10 PM
 */

/*************************************************************
 *Includes and defines
 *************************************************************/

#include "PIC16F1719_Internal.h"
#include "I2C.h"
#include "oled.h"
```

```
/************************************************************
 * Function: void initMain()
 *
 * Returns: Nothing
 *
 * Description: Contains initializations for main
 *
 * Usage: initMain()
 ************************************************************/

void initMain(){
    // Run at 16 MHz
    internal_16();

    //////////////////////
    // Configure PWM Ports
    //////////////////////

    // Set PIN B0 as output
    TRISBbits.TRISB0 = 0;

    // Set PIN B1 as output
    TRISBbits.TRISB1 = 0;

    // Turn off analog on PORTB
    ANSELB = 0;

    ////////////////////
    // Configure Timer6
    ////////////////////

    // Select PWM timer as Timer6 for CCP1 and CCP2
    CCPTMRSbits.C1TSEL = 0b10;
    CCPTMRSbits.C2TSEL = 0b10;

    // Enable timer Increments every 250 ns (16MHz clock) 1000/(16/4)
    // Period = 256 x 0.25 us = 64 us

    //                          Crystal Frequency
    //    PWM Freq  = ----------------------------------------
    //                  (PRX + 1) * (TimerX Prescaler) * 4

    //    PWM Frequency = 16 000 000 / 256 * 1 * 4
    //    PWM Frequency = 15.625 kHz

    // Prescale = 1
    T6CONbits.T6CKPS = 0b00;

    // Enable Timer6
    T6CONbits.TMR6ON = 1;

    // Set timer period
```

```
PR6 = 255;

//////////////////////////
// Configure PWM
/////////////////////////

// Configure CCP1

// LSB's of PWM duty cycle = 00
CCP1CONbits.DC1B = 00;

// Select PWM mode
CCP1CONbits.CCP1M = 0b1100;

// Configure CCP2

// LSB's of PWM duty cycle = 00
CCP2CONbits.DC2B = 00;

// Select PWM mode
CCP2CONbits.CCP2M = 0b1100;

/////////////////////
// Setup I2C
//////////////////

// Setup pins for I2C
ANSELCbits.ANSC4 = 0;
ANSELCbits.ANSC5 = 0;

TRISCbits.TRISC4 = 1;
TRISCbits.TRISC5 = 1;

//////////////////////////////
// Configure PPS
/////////////////////////////

PPSLOCK = 0x55;
PPSLOCK = 0xAA;
PPSLOCKbits.PPSLOCKED = 0x00; // unlock PPS

// Set RB0 to PWM1
RB0PPSbits.RB0PPS = 0b01100;

// Set RB1 to PWM2
RB1PPSbits.RB1PPS = 0b01101;

RC4PPSbits.RC4PPS = 0x0011;   //RC4->MSSP:SDA;
SSPDATPPSbits.SSPDATPPS = 0x0014;   //RC4->MSSP:SDA;
SSPCLKPPSbits.SSPCLKPPS = 0x0015;   //RC5->MSSP:SCL;
RC5PPSbits.RC5PPS = 0x0010;   //RC5->MSSP:SCL;

PPSLOCK = 0x55;
```

```
    PPSLOCK = 0xAA;
    PPSLOCKbits.PPSLOCKED = 0x01; // lock PPS

     ////////////////////
    // Configure ADC
    ///////////////////

    // Fosc/32 ADC conversion time is 2.0 us
    ADCON1bits.ADCS = 0b010;

    // Right justified
    ADCON1bits.ADFM = 1;

    // Vref- is Vss
    ADCON1bits.ADNREF = 0;

    // Vref+ is Vdd
    ADCON1bits.ADPREF = 0b00;

    // Set input channel to AN0
    ADCON0bits.CHS = 0x05;

    // Zero ADRESL and ADRESH
    ADRESL = 0;
    ADRESH = 0;

    // ADC Input channel PIN E0
    ANSELEbits.ANSE0 = 1;
}

/**************************************************************
 * Function: Main
 *
 * Returns: Nothing
 *
 * Description: Program entry point
 **************************************************************/

void main(void) {
    initMain();

     // Initialize I2C
    I2C_Init();

    __delay_ms(500);

    // Initialize OLED
    OLED_Init();

    __delay_ms(1000);

    // clear OLED
```

```
OLED_Clear();

// result to store ADC conversion
float result;

// variables for conversion
float conversion10;
float farenheit;
float celsius;

// PWM Off
CCPR1L = 0;
CCPR2L = 0;

OLED_YX(0, 0);
OLED_Write_String("Init");

while(1){

    // Turn ADC on
    ADCON0bits.ADON = 1;

    // Sample CH0
    __delay_us(10);
    ADCON0bits.GO = 1;
    while (ADCON0bits.GO_nDONE);

    // Store ADC result
    result = ((ADRESH<<8)+ADRESL);

    // 10 bit conversion
    conversion10 = (result * 5000)/1024 ;

    // to Fahrenheit
    farenheit = conversion10 / 10;

    // to Celsius
    celsius = (farenheit - 32) * 5/9;

    // Display temperature

    OLED_YX(1, 0);
    OLED_Write_Integer((int)celsius);

    // Update every second
    __delay_ms(2000);

    // If temperature is more than 35C turn on fan
    if ((int)celsius > 35){

        // Forward
        CCPR1L = 127;
        CCPR2L = 0;
```

```
                // clear OLED
                OLED_Clear();

                OLED_YX(0, 0);
                OLED_Write_String("Warning!!");
        }

        // If less turn off fan
        else{
                OLED_YX(0, 0);
                OLED_Write_String("Temp OK");
                CCPR1L = 0;
                CCPR2L = 0;
        }

      }
       return;
}
```

Listing 14-1 Temperature-Controlled Fan Project

The core of the program lies in its continuous monitoring and response loop, where it reads the temperature, performs necessary conversions to Celsius, and then decides on the fan's operation based on the current temperature. This is achieved through a mix of hardware configuration for precise control (PWM for fan speed modulation and ADC for temperature measurement) and software logic to interpret the temperature data and adjust the fan speed accordingly. The inclusion of an OLED display enriches user interaction by providing instant feedback on the system's status, making the project not only an excellent example of embedded systems design but also user-friendly. Through this, the project showcases the integration of electronic components and programming to solve practical problems, such as temperature regulation, with efficiency and precision.

Project: UART to Browser Bridge

In our second project, we delve into the creation of a UART to browser bridge, an innovative application that leverages the microcontroller's UART (Universal Asynchronous Receiver/Transmitter) interface to facilitate direct communication between hardware devices and web technologies. This bridge serves as a conduit, allowing data from the microcontroller to be displayed in real time on a web browser interface. The project not only demonstrates the capability of microcontrollers to interact with software layers but also emphasizes the practicality of integrating simple hardware protocols with complex web-based systems. By utilizing serial communication to send data to a browser, this project provides a hands-on experience with both electronic communication standards and web programming, thus bridging the gap between embedded systems and user-facing applications.

Essentially it's a simple way to monitor your microcontroller projects without having to use a Bluetooth or Wi-Fi module. The microcontroller project is given in Listing 14-2.

```
/*
 * File: main.c
 * Author: Armstrong Subero
 * PIC: 16F1719 w/int OSC @ 32MHz, 5v
 * Program: 51_UART_To_Browser_Bridge
 * Compiler: XC8 (v2.45, MPLAX X v6.15)
 * Program Version: 1.2
 *                  * Updated from PIC16F1717 to PIC16F1719
 *                  * Updated clock speed to 32 MHz
 *                  * Added PLL stabilization
 *
 * Program Description: This program allows the PIC16F1719 to
communicate with a server to send data via the serial port and turn
on an LED based on data received from the server
 *
 * Hardware Description: PIN RB2 of a the PIC16F1719 MCU is connected
to a PL2303XX USB to UART converter cable and a 10k potentiometer is
connected to pin RA0. An LED is connected to PORTD1
 *
 * Created November 7th, 2016, 7:05 PM
 * Last Updated: March 24th, 2024, 4:58 AM
 */

/*******************************************************************
 *Includes and defines
 *******************************************************************/

#include "PIC16F1719_Internal.h"
#include "EUSART.h"

/*******************************************************************
 * Function: void initMain()
 *
 * Returns: Nothing
 *
 * Description: Contains initializations for main
 *
 * Usage: initMain()
 *******************************************************************/

void initMain(){

    // Run at 32 MHz
    internal_32();

    // Allow PLL startup time ~2 ms
    __delay_ms(10);

    // Set PIN D1 as output
    TRISDbits.TRISD1 = 0;

    // Turn off LED
    LATDbits.LATD1 = 0;

    // Setup PORTD
    TRISD = 0;
    ANSELD = 0;
```

```
TRISBbits.TRISB2 = 0;
ANSELBbits.ANSB2 = 0;

/////////////////////
// Setup EUSART
////////////////////
PPSLOCK = 0x55;
PPSLOCK = 0xAA;
PPSLOCKbits.PPSLOCKED = 0x00; // unlock PPS

RB2PPSbits.RB2PPS = 0x14;   //RB2->EUSART:TX;
RXPPSbits.RXPPS = 0x0B;   //RB3->EUSART:RX;

PPSLOCK = 0x55;
PPSLOCK = 0xAA;
PPSLOCKbits.PPSLOCKED = 0x01; // lock PPS

////////////////////
// Configure ADC
///////////////////

// Set A0 as input
TRISAbits.TRISA0 = 1;

// Set A0 as analog
ANSELAbits.ANSA0 = 1;

// Fosc/32 ADC conversion time is 1.0 us
ADCON1bits.ADCS = 0b010;

// Right justified
ADCON1bits.ADFM = 1;

// Vref- is Vss
ADCON1bits.ADNREF = 0;

// Vref+ is Vdd
ADCON1bits.ADPREF = 0b00;

// Set input channel to AN0
ADCON0bits.CHS = 0b00000;

// Zero ADRESL and ADRESH
ADRESL = 0;
ADRESH = 0;

//////////////////////
// Configure Timer1
/////////////////////
// prescale = 1
T1CONbits.T1CKPS = 0b00;
```

```
    // Set TMR1 to 0
    TMR1 = 0;

    // enable timer1
    T1CONbits.TMR1ON = 1;

    // enable timer1 interrupt
    PIE1bits.TMR1IE = 1;

    // enable peripheral interrupt
    INTCONbits.PEIE = 1;

    // enable global interrupts
    ei();
}

/*************************************************************
 * Function: Main
 *
 * Returns: Nothing
 *
 * Description: Program entry point
 *************************************************************/

void main(void) {
    initMain();

    // Initialize EUSART module with 19200 baud
    EUSART_Initialize(19200);

    while(1)
    {

      if (PIR1bits.RCIF)
        {   // Check if data received via USART
         char command = EUSART_Read();  // Read incoming command from
serial port using EUSART
         if (command == '1') {
            LATDbits.LATD1 = 1; // Turn on the LED if '1' is received
            EUSART_Write_Text("LED turned on\n");
         } else if (command == '0') {
            LATDbits.LATD1 = 0; // Turn off the LED if '0' is
received
            EUSART_Write_Text("LED turned off\n");
        }
      }
    }

    return;
}

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

```
 * Function: void interrupt isr(void)
 *
 * Returns: Nothing
 *
 * Description: Timer1 interrupt
 *************************************************************/

void __interrupt() isr(void){
    static uint8_t count = 0;

    PIR1bits.TMR1IF = 0;

    count++;

    // Send sensor data every second
    if (count == 122){
        count = 0;

        // Turn ADC on
        ADCON0bits.ADON = 1;

        // Sample CH0
        __delay_us(10);
        ADCON0bits.GO = 1;
        while (ADCON0bits.GO_nDONE);

        // Store ADC result
        int result = ((ADRESH<<8)+ADRESL);

        EUSART_Write_Integer(result);
        EUSART_Write_Text("\n");

    }

}
```

Listing 14-2 PIC Serial to Browser Program

The main function of the program continuously checks for incoming data via the EUSART module. If data is received, it processes commands to turn an LED on or off based on the commands ‘1’ or ‘0’, respectively, while sending a confirmation message back through EUSART. This demonstrates a responsive system that interacts with external inputs to control hardware (LED) and provide feedback.

The interrupt service routine (ISR) is triggered by Timer1 overflows, which are set to occur at intervals that simulate a one-second timer, although the exact duration depends on the timer configuration, which needs validation against the system clock. When the ISR fires, it increments a counter, and upon reaching a specific count (here assumed to be every second at count 122), it activates the ADC to sample the analog value.

The first part of our project involves taking data from our serial port and putting it on a local server that we can read. All our code is written in Python, which should be very easy for an embedded C developer like yourself to understand. The Serial to Server Program is given in Listing 14-3.

```
import asyncio
import serial
import aiohttp
import socketio

sio = socketio.Client()
ser = serial.Serial('COM3', 9600, timeout=0)

@sio.on('turn_on_led')
def turn_on_led():
    print("Received command to turn on LED")
    ser.write(b'1')

@sio.on('turn_off_led')
def turn_off_led():
    print("Received command to turn off LED")
    ser.write(b'0')

sio.connect('http://localhost:5000')

async def read_sensor(mailbox):
    sensor_value = ""

    while True:
        if ser.in_waiting > 0:
            data = ser.read(ser.in_waiting).decode()
            sensor_value += data

        lines = sensor_value.split('\n')
        if len(lines) > 1:
            complete_values = lines[:1]
            sensor_value = lines[-1]

            for line in complete_values:
                sensor_val = line.strip()
                await mailbox.put(sensor_val)

async def send_to_server(mailbox):
    url = 'http://localhost:5000/recieved_sensor_data'
    async with aiohttp.ClientSession() as session:
        while True:
            sensor_val = await mailbox.get()
            if sensor_val:
                data = {'sensor_data': sensor_val}
                async with session.post(url, json=data) as response:
                    if response.ok:
                        print("Sensor data sent sucessfully")
                    else:
                        print("Failed to send sensor data")

async def main():
    mailbox = asyncio.Queue(maxsize = 1)
    tasks = [
        asyncio.create_task(read_sensor(mailbox)),
        asyncio.create_task(send_to_server(mailbox)),
    ]

    await asyncio.gather(*tasks)

# Run the main event loop
asyncio.run(main())
```

Listing 14-3 Serial to Server Program

This Python script combines asynchronous programming with serial communication and WebSocket connections to interact with IoT devices. The script integrates libraries such as "asyncio", "serial", "aiohttp", and "socketio" to facilitate real-time operations involving a microcontroller connected through a serial port.

The script initializes a WebSocket client using "socketio.Client()" to connect to a local server at "http://localhost:5000". This client listens for specific events ("'turn_on_led'" and "'turn_off_led'"), handling them by sending commands through a serial connection established on COM3 with a baud rate of 9600. When the respective events are received, the script writes the corresponding byte ("b'1'" to turn on, "b'0'" to turn off) to the serial port controlling an LED.

In parallel, the script continuously reads sensor data from the same serial port. This is handled by the "read_sensor" coroutine, which accumulates data from the serial buffer. If complete lines of data are detected (terminated by newline characters), these are enqueued into an asyncio queue ("mailbox"), potentially for further processing or immediate dispatch.

The "send_to_server" coroutine then takes these sensor values from the queue and sends them as JSON payloads to a specified endpoint ("/recieved_sensor_data") on the server using the "aiohttp" library for asynchronous HTTP requests. Successful or failed data transmission is logged accordingly based on the HTTP response status.

The "main" function sets up and runs these tasks concurrently. It creates an asyncio queue ("mailbox") to serve as an intermediary for sensor data, schedules both the sensor reading and server communication tasks, and awaits their completion collectively, illustrating the script's capacity to perform I/O operations concurrently, which is essential for responsive IoT applications.

Finally, the script's execution kicks off by running the main event loop with "asyncio.run(main())", orchestrating all asynchronous activities in a nonblocking fashion. This makes our application very responsive.

Setting Up a Flask Server

Our next step is to set up a Flask server. In case you're not familiar with Flask, Flask is a lightweight and versatile web framework for Python, known for its simplicity and fine-grained control over components. It is particularly favored for small to medium web applications and microservices and makes a great server for building web-based PIC microcontroller projects.

Flask operates with minimal overhead, thanks to its "no batteries included" philosophy, yet it is extensible enough to support complex applications with additional libraries as needed. The framework uses a simple yet powerful routing system where URLs can be bound to Python functions, making web application development both intuitive and straightforward. Developers can quickly create a web server using Flask that handles web requests and serves data to clients, making it an ideal choice for projects that require a quick setup and a clean, maintainable code base.

This Flask server will communicate with our serial server program and will be responsible for updating the web page. The listing for the Flask server is given in Listing 14-4.

```
from flask import Flask, render_template, request
from flask_socketio import SocketIO, emit

app = Flask(__name__)
app.config['SECRET_KEY'] = 'secret'
socketio = SocketIO(app)

@app.route('/')
def index():
    return render_template('index.html')

@socketio.on('message')
def handle_message(message):
    print(f"Received message: {message}")
    if message == 'turn_on_led':
        socketio.emit('turn_on_led')
    elif message == 'turn_off_led':
        socketio.emit('turn_off_led')

@app.route('/receive_sensor_data', methods=['POST'])
def receive_sensor_data():
    data = request.json.get('sensor_data')
    print(f'Received sensor data from webpage: {data}')
    socketio.emit('update_sensor', data)  # Emit the sensor data to
all connected clients
    return 'Data received successfully'

if __name__ == '__main__':
    socketio.run(app, debug=True)
```

Listing 14-4 Setting Up the Flask Server

Our code outlines a web server application using Flask along with Flask-SocketIO for handling real-time communication. The application is configured with a secret key to manage sessions and other security features.

At the heart of the application, there's a Flask "app" instance. It serves a web page from the root URL ("/"), which likely contains the user interface for interacting with IoT devices, as the function "index" returns a rendered "index.html" template.

The code also incorporates SocketIO to manage WebSocket connections, which allow real-time, bidirectional communication between the server and connected clients. Within this setup, there's an event handler "handle_message" that listens for incoming messages from clients. When messages such as 'turn_on_led' or 'turn_off_led' are received, the server reacts by emitting corresponding events back to the clients, facilitating actions like turning an LED on or off.

Additionally, there's a route "'/receive_sensor_data'" that handles POST requests, which is likely used to receive sensor data sent by IoT devices. The received data is extracted from the JSON payload of the request, logged, and then broadcast to all connected clients using the "socketio.emit" function. This would update clients with the latest sensor readings, maintaining real-time data flow across the system.

Finally, the application includes a check to see if the script is the main one being run, and if so, it starts the Flask application with SocketIO integrated, enabling debugging features to aid in development and troubleshooting.

Overall, this server-side application acts as the communication hub for an IoT system, handling user inputs from the web interface and sensor data from IoT devices and updating all clients connected in real time with current device statuses and sensor readings.

Static Page

Once you have your Flask server set up, you can create a web page that will be updated by the server. This web page displays all the data from our serial port. The code for our web page is given in Listing 14-5.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <title>IoT Control Panel</title>
</head>
<body>
  <div>
    <h2>IoT Dashboard</h2>
    <div>
      <button type="button" id="turnOnButton">Turn On LED</button>
      <button type="button" id="turnOffButton">Turn Off LED</button>
    </div>
    <div>
      <h4>Sensor Value:</h4>
      <p id="sensorValue">Awaiting data...</p>
    </div>
  </div>

  <!-- jQuery -->
  <script src="{{ url_for('static', filename='js/jquery.min.js') }}">
</script>
  <script src="{{ url_for('static',
filename='js/bootstrap.bundle.min.js') }}"></script>

  <!-- Socket.IO -->
  <script src="{{ url_for('static', filename='js/socket.io.js') }}">
</script>

  <script>
    const socket = io.connect('http://' + document.domain + ':' +
location.port);

    document.getElementById('turnOnButton').addEventListener('click',
function() {
      socket.send('turn_on_led');
    });

    document.getElementById('turnOffButton').addEventListener('click',
function() {
      socket.send('turn_off_led');
    });
    // Receive sensor data from the server and update the UI
    socket.on('update_sensor', function(data) {
      document.getElementById('sensorValue').innerText = data;
    });
  </script>
</body>
</html>
```

Listing 14-5 The Web Page

The provided HTML document defines a basic user interface for an Internet of Things (IoT) Control Panel, specifically tailored for operations such as turning on/off an LED and displaying sensor data. The document begins with standard HTML and metadata settings, including character encoding and viewport settings for responsive design, ensuring it renders well on devices of varying sizes.

The title of the page is set to “IoT Control Panel”, which likely appears in the browser tab. Within the body of the document, there’s a main division containing an IoT Dashboard header. This section includes two buttons labeled “Turn On LED” and “Turn Off LED”, each with specific identifiers to facilitate interaction through scripting.

Below the buttons, there’s another division dedicated to displaying sensor values. Initially, it shows a placeholder text “Awaiting data...”, which suggests that it will later display live data received from sensors.

The document also includes references to several JavaScript libraries and frameworks: jQuery, Bootstrap for styling and front-end components, and Socket.IO for real-time bidirectional communication between web clients and servers. The scripts at the end of the document set up a connection to the server using Socket.IO, based on the current domain and port, indicating dynamic retrieval of these values.

Event listeners are attached to both buttons, sending commands (‘turn_on_led’ or ‘turn_off_led’) to the server when clicked, leveraging the Socket.IO connection. Additionally, a listener for ‘update_sensor’ events is set up to receive sensor data from the server, updating the content of the paragraph with ID “sensorValue” to reflect the latest sensor readings. This setup allows for real-time interaction and data display, making it suitable for controlling and monitoring IoT devices remotely.

Conclusion

In this chapter, we looked at two PIC microcontroller projects. One of our projects is a classical embedded systems project: a temperature-controlled fan. We then looked at a UART to browser bridge providing a fun way for us to monitor our embedded systems project without using a Bluetooth or a Wi-Fi module.

Index