2011-04-28

Introductory Microcontroller Programming

Peter J. Alley
Worcester Polytechnic Institute

Introductory Microcontroller Programming

by

Peter Alley

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Master of Science

in

Robotics Engineering

May 2011

Prof. William Michalson
Advisor

Prof. Taskin Padir
Committee member

Prof. Susan Jarvis
Committee member

Abstract

This text is a treatise on microcontroller programming. It introduces the major peripherals found on most microcontrollers, including the usage of them, focusing on the ATmega644p in the AVR family produced by Atmel. General information and background knowledge on several topics is also presented. These topics include information regarding the hardware of a microcontroller and assembly code as well as instructions regarding good program structure and coding practices. Examples with code and discussion are presented throughout. This is intended for hobbyists and students desiring knowledge on programming microcontrollers, and is written at a level that students entering the junior level core robotics classes would find useful.

Contents

List of Figures

List of Tables

Preface

Why this topic?

In the spring of 2009, Worcester Polytechnic Institute's junior level robotics courses were first offered, with me as one of the teacher's assistants for the courses. At the time, I had never worked with microcontrollers before, excepting the VEX system used in the lower level WPI robotics classes, which provides a layer of abstraction between the programmer and the actual hardware. This was the first time that many of the students and my self actually had to worry about special function registers, peripheral initialization and other similar tasks. Over the period of being the TA for these courses multiple times, I learned a great deal, but also noticed several recurring issues the students seemed to face. In addition to furthering my own knowledge on the care and feeding of microcontrollers, this thesis is an attempt to provide students with a resource to overcome some of their difficulties, to provide a better understanding of the hardware level operation of a microcontroller, and as instruction into improving their knowledge and code.

One problem I noticed, was that many students viewed a microcontroller as a little black box in which magic happens. By the end of the course the black box in their imagination may have shrunk slightly and grown a few more inputs and outputs, but they still did not know what was actually occurring in the hardware to generate the results they desired and saw. Therefore, one of my goals was to provide this understanding both by giving a general overview of the hardware as well as by examining the schematics for portions of the microcontroller at a component level and explaining their operation.

A second problem is that students appear to have a great aversion to proper commenting and documentation of their code. In many cases it is due to the laziness and overconfidence of the programmer, however in some cases it is due to the programmer having never been properly instructed on good practices and the reasoning behind them. This instruction is something I provide, along with several examples of well documented code to model what should be done.

Finally, I have noticed that students coding styles lack forethought and design. When handed a problem, they will jump to a conclusion on how to solve one section of it and begin work on that, which causes portions of the solution to be completed before others are even considered. This leads to disorganized,

hard to follow code which is difficult to maintain and debug. This text attempts to demonstrate a method of working through problems stem to stern, and show that forethought can make the solution simpler and more elegant.

Although this text was initially intended for students in the first junior level robotics engineering class at WPI (RBE3001), it should prove to be a valuable resource to anyone beginning work with microcontrollers, students and hobbyists alike. This text focuses on the use of the ATmega644p 8-bit microcontroller produced by Atmel, though the lessons taught can be applied to any microcontroller, and some lessons to programming in general.

Organization

This text works through the various peripherals on the ATmega644p in an order commensurate with their complexity and usefulness. Interspersed with these portions are sections devoted to solving the other problems I have noticed over the years.

- Chapter 1: What is a Microcontroller?
  This chapter explains the general components of a microcontroller, via comparisons with microprocessors and microcomputers. The next step was to introduce the common memory models in order to distinguish what most people are more familiar with: von Neumann architecture where programs and data are stored together such as in desktop computers, from the Harvard architecture commonly used in microcontrollers. The Harvard memory model separates the program and data memories into separate address spaces.

  The first chapter of the text finishes with an explanation of what the stack is and how it operates. This background knowledge provides a basic understanding of how microcontrollers operate, and is important for understanding assembly code.

- Chapter 2: Datasheets, SFRs and Libraries
  Chapter 2 is a collection of disparate topics that each deserve discussion prior to beginning to look at code. The section on datasheets disects an example datasheet and explains the various portions. The special function register section introduces what SFRs are and how to address them on the microcontroller. The final section in the chapter gives a basic understanding of what a library is, and which are most commonly used in programming the ATmega644p.

- Chapter 3: Hello World
  In this chapter the first peripheral, the digital input and output, is introduced and used in a basic task. This example is examined from a very basic level and demonstrates a good process for solving the problem without jumping to conclusions and checking every aspect.

- Chapter 4: Analog to Digital Converter
  Continuing the example from chapter 3, chapter 4 introduces the ADC. After explaining background information on ADCs in general, the example using the ADC on the ATmega644p is presented.

- Chapter 5: Code Styles As the programs in the examples, and those that would be required in using these peripherals, are becoming more complex, chapter 5 takes a break from peripherals and hardware in general to introduce good coding styles. These include how to use and form header files, when to use documentation and non-documentation comments as well as more information on functions and libraries, this time looking at creating libraries rather than using pre-existing ones. The chapter finishes up by reexamining the example from chapter 4, using it to demonstrate proper styles and comments.

- Chapter 6: C Data Structures Continuing the digression from microcontroller peripherals, chapter 6 presents several data structures that are useful in programming: arrays, structs and pointers. In addition to discussing what these structures are and the specifics of their usage, several uses for structs that are not immediately apparent are presented which aid in making programs cleaner and easier to create.

- Chapter 7: Serial Communications With this chapter, the text returns to discussing microcontroller peripherals. After providing background knowledge on serial communications in general, and on each peripheral in question, the operations of three types of serial communications are presented with examples. These communications methods are USART and the RS232 standard, the serial-peripheral interface (SPI) and the two-wire interface (TWI or I2C).

- Chapter 8: Interrupts and Timers Chapter 8 concludes the discussions on individual peripherals by introducing hardware interrupts, timers, and how they can be used in conjunction or separately.

- Chapter 9: External Devices Up to this point, the text has focused on the microcontroller and coding styles. This chapter breaks off and looks at the other devices that might need to be attached to a microcontroller: motors, servos, various sorts of sensors, and how to use them.

- Chapter 10: Real-Time Operating Systems This chapter discusses real-time operating systems (RTOSs) and their uses. It proceeds to present an example of a basic RTOS that might be used to control a mobile robot, which is implemented in the following chapter.

- Chapter 11: The Final Example This chapter is dedicated to a single mobile robot example. It begins with a simple problem statement and works through several chains of choices and decisions to be made, culminating in a functional program.

Hardware

This text focuses on the usage of the ATmega644p. This is an 8-bit microcontroller from the AVR family produced by Atmel [1]. There are a variety to choices for programmers, however the STK500 is a good choice [10]. It is a development board capable of handling a number of difference microcontrollers, provides the hardware for connecting the microcontroller to a computer using serial via the RS232 standard and provides eight switches and LEDs for use with basic programs. Figure 1 shows the 40 pins of the ATmega644p with the names of each pin. Most pins have multiple names as they are used for different tasks by different peripherals.

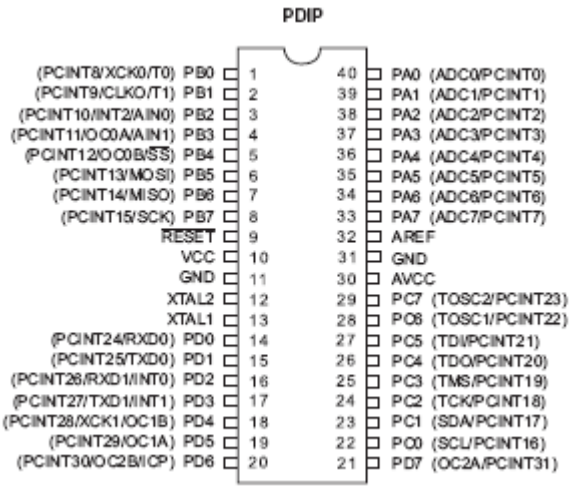


Figure 1: Pin configuration for ATmega644p.

A variety of additional hardware components are introduced, primarily in the later chapters. These are by no means required to understand and learn from this text, and for many readers it would be a better exercise to determine what devices best fit their needs, rather than blindly following the examples presented herein.

Software

There are several choices for the software required to develop programs for the ATmega644p and download them to the microcontroller.

- AVR Studio AVR Studio, in version 5 at the time of writing, is a development environment produced by Atmel specifically for the AVR family of microcontrollers [2]. As it is more dedicated than other IDEs, it has debugging and simulation tools that surpass those available elsewhere. Its

primary drawback is that it is specialized and cannot be used to develop programs for microcontrollers outside this family.

- Eclipse Eclipse is another good choice for IDEs. Although it requires more setup and configuration than AVR Studio, it is a very extensible environment and there are plugins available for most any programming language or environment one could need. Instructions for setting up Eclipse for programming the ATmega644p, as well as for creating projects within Eclipse are provided in appendix C.

- Others While there most certainly are other choices available, these two are the best. They provide a great number of features useful in development, and are both well supported in the community, with help easily available.

Read, learn and enjoy

As you read this text, keep this thought in the back of your mind: this document is not trying to turn you into an automaton by telling you how to do everything every time. It is attempting to demonstrate methods and teach you how to learn on your own, providing you with the tools to do so. Keep this in mind and the lessons learned within will be useful outside as well.

Chapter 1

What is a Microcontroller?

As time progresses, an increasing number of consumer goods contain microcontrollers. With the current cost of basic microcontrollers as low as 30 cents [6], they are being used as simple solutions to tasks that previously utilized transistor-transistor-logic. With the profusion of these devices being used in industry as well as in hobby electronics the ability to program them can be very useful for any project that may be attempted.

This text is written to be an introduction to microcontrollers as well as to take a new user from opening the datasheet for the first time through programming a microcontroller and using a variety of peripheral devices. For a more advanced user this text will provide suggestions on how to make code more readable and put forth good coding practices. While this will focus on a single microcontroller, the ATmega644p, the ideas presented are valid for most microcontrollers available today.

1.1 Micro-processors, -computers, -controllers

While microprocessors, microcomputers and microcontrollers all share certain characteristics and the terms are often used interchangeably, there are certain distinctions that are used to classify them into separate categories.

1.1.1 Microprocessor

The simplest of the three categories is the microprocessor. Also known as a CPU (Central Processing Unit), these devices are generally found at the heart of a much larger system such as a desktop computer and are primarily used as data processors. They generally consist of an arithmetic logic unit (ALU), an instruction decoder, a number of registers and digital input/output (DIO) lines (see figure 1.1). Some processors also include memory spaces such as a cache or stack which can be used for more rapid temporary storage and retrieval of data than having to access system memory. Additionally, the processor must connect

to some form of data bus to access the memory and input/output peripherals external to the processor itself.

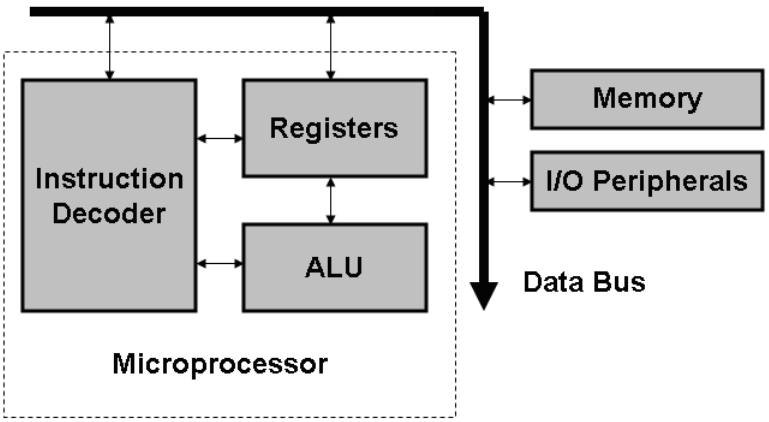


Figure 1.1: Basic components of a microprocessor.

Depending on the memory architecture the microprocessor may have only a handful of registers such as a program counter for keeping track of the address of the next instruction and an instruction register for loading and storing the next instruction; or there may be dozens of registers. These additional registers are known as general purpose registers and store data while it is being used.

1.1.2 Microcomputer

A microcomputer contains all the components of a computer in a small circuit, though not on a single chip. This term generally applies to laptops and desktop computers, however has fallen out of usage for these devices. The component devices of a microcomputer consist of a CPU (such as a microprocessor), memory and/or other storage devices, as well as IO devices (see figure 1.2). Several examples of I/O devices include a keyboard, display, network, etc.; but can be any device that the microcomputer uses to collect or distribute information.

1.1.3 Microcontroller

A microcontroller is, in some ways, a cross between a microprocessor and a microcomputer. Like microprocessors, the term microcontroller refers to a single device; however it contains the entire microcomputer on that single chip. Therefore a microcontroller will have a processor, on-board memory as well as a variety of IO devices. While using a microcontroller instead of a microcomputer simplifies the overall design, to accomplish this it sacrifices the flexibility. A microcomputer can be configured to have specific quantities of memory or devices attached. Microcontrollers are generally limited to the memory sizes and peripherals that the manufacturers dictate. There are a great many choices in

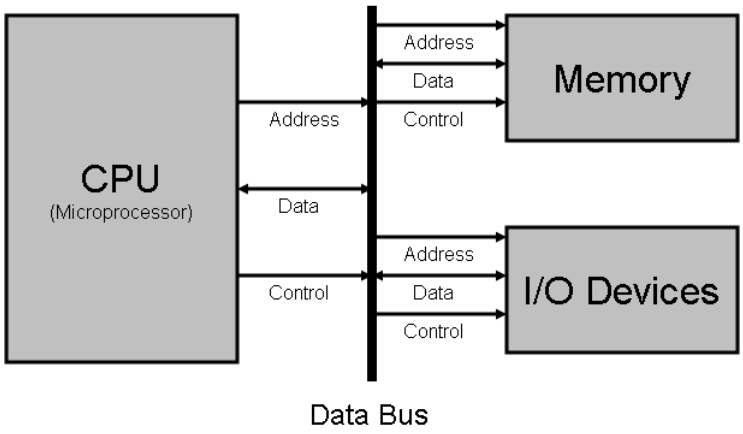


Figure 1.2: Basic components of a microcomputer

microcontrollers and their capabilities, however this still can be a limitation in some circumstances.

Because microcontrollers are designed more to be standalone data collection and control devices, rather than for the human interaction or networking tasks that microcomputers often handle, their standard IO devices differ. Analog-digital converters (ADCs), timers and external interrupts are common peripherals found on microcontrollers, while keyboards, monitors and other devices used daily to control a personal computer are not.

1.2 Memory Models

1.2.1 Von Neumann

The von Neumann architecture was named after a scientist involved in the Manhattan Project and, due to the computational requirements of that project, joined in the development of the EDVAC stored-program computer [12]. During this time he wrote First Draft of a report on the EDVAC, which became the source of the von Neumann architecture [21].

The first computers and computational devices had fixed programs. These programs were built into the machine in various ways and to change the program the machine often had to be rebuilt. This includes most of the early computers such as the ENIAC. These rebuilds could take weeks and used a high percentage of the machine's time.

The von Neumann architecture fixed this problem by storing the program in memory (hence stored-program). This memory block is shared between the program storage and data storage, which allows data to be treated as code and vice versa. Furthermore, it allows the use of self-modifying code, which was useful in the early days of the architecture to reduce memory use or improve

performance [21].

Figure 1.3 contains a block diagram of the von Neumann architecture. This shows the single memory block which both the control unit, the device that reads and interprets the program, and the ALU, where most operations are executed, are connected to. The necessity of communicating with memory external to the CPU leads to a throughput limit known as the von Neumann bottleneck [3]. This bottleneck is especially severe in this architecture compared to others due to the control unit and ALU both needing to read and write to the memory, therefore sharing the limiting resource in the system (memory access time).

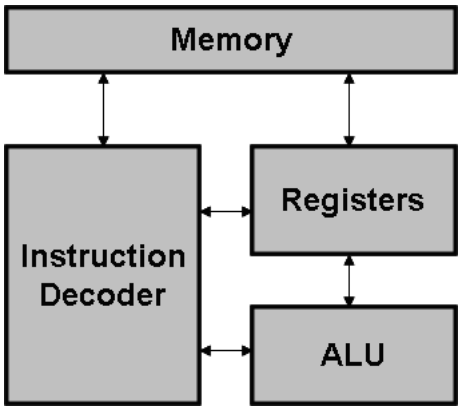


Figure 1.3: Block diagram of the von Neumann architecture.

1.2.2 Harvard Architecture

One solution to the von Neumann bottleneck is to separate the program memory from the data memory (see figure 1.4). This separation allows for several improvements over the von Neumann architecture.

The first and most obvious improvement is that both program memory and data memory can be accessed simultaneously. In the von Neumann architecture, in order to store a word from a register through the ALU to memory, the control unit must first load and interpret the instruction, then the ALU can transfer the data to memory, and finally the control unit can move on to the next instruction. This requires two separate read/write operations along the same path. In the Harvard architecture the write to data memory and read from program memory for the next operation can be performed simultaneously reducing the time required for any instruction that accesses data memory.

A somewhat less obvious change that can greatly increase operating speed is that lengths of words in the program memory no longer need to be integer numbers of bytes. This allows for longer instruction words that can contain both an instruction and a memory address in a single instruction, and therefore every read from program memory, and every clock cycle of the processor, can be an entire instruction. In the von Neumann architecture instructions are often

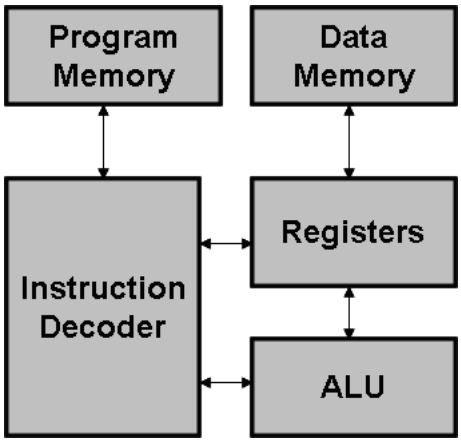


Figure 1.4: A block diagram of the Harvard architecture.

multiple words long to encompass both the op-code and any memory addresses required. The example given above would therefore require three read/write operations as opposed to the two previously mentioned or the single operation of the Harvard architecture.

The major disadvantage of the Harvard architecture is that it cannot modify the program memory, limiting its usefulness in general systems such as personal computers. This does not pose a problem for more dedicated processors such as those in embedded systems, though memory bandwidth must be high as the memory may be accessed by two operations per cycle. Both the AVR family of microcontrollers produced by Atmel and Microchip's PIC family are Harvard architectures, though they have been modified slightly to allow read/write operations to program memory. These operations are used primarily for bootloaders.

1.2.3 Modified Harvard Architecture

The modified Harvard architecture is a mix of von Neumann and Harvard architectures attempting to capture the benefits of each. Specifically, the data and program again share a memory space similar to how the von Neumann architecture works, however data and instructions do not share cache memories or paths between the CPU and memory. This allows for less restricted memory access than von Neumann, and yet the ability to treat code and data as each other that Harvard lacks. One of the most common examples of processors that use this architecture is the x86 processor found in most personal computers.

1.3 The Stack

The stack is a last-in-first-out (LIFO) section of memory used to store information related to procedure calls as well as data for certain operations. The stack can be in general memory with a fixed or variable size, or have a dedicated memory block of fixed size. In either case the processor will have a stack pointer register pointing to the most recently addressed location in the stack. In some processors the stack begins at the highest address in its memory block and grows downwards, in others it begins at the lowest address and grows up.

LIFO refers to the order in which data is removed from where it is stored. In LIFO the most recent piece of data that was added to storage is the first one removed when data is requested. This is the source of the stacks name, due to the similarity between LIFO and creating a stack of some form of object, when all that is available is the top item. The alternative to last-in-first-out is first-in-first-out or FIFO. In this paradigm the oldest piece of data in storage is removed whenever data in requested. This is often called a queue due to its similarity to a checkout line or the line of people waiting for a cashier at a bank.

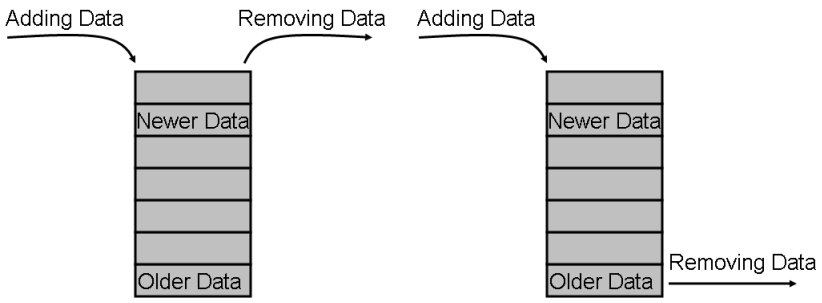


Figure 1.5: Origin of requested data in LIFO storage (left) and FIFO (right).

The basic stack supports only two operations; `push` and `pop`. The `push` operation adds data to the top of the stack, incrementing the stack pointer, while `pop` decrements the stack pointer and returns the data on top of the stack. Some environments that depend on the stack for many of its operations will occasionally have additionally operations such as `peek` (a `pop` operation without changing the stack pointer), `dup` (a `pop` followed by pushing the result twice), or `swap` (switching the order of the top two items on the stack).

In addition to the stack, any processor will be able to store several pieces of data without having to access the system's memory, though the amount of storage varies between processors. These storage locations are called registers, and while some may have specific purposes, most processors will have several for general usage called general purpose registers (GPRs). Among the special purpose registers, there will be one called the program counter. This register holds the memory address of the next instruction to be executed. Any time an instruction is executed, it increases, and without it the processor wouldn't know where to look for the next instruction. What happens then when a function call

occurs? Some form of return address has to be kept for the program to return from the called function's location in memory back to where the program counter was before, and that is where the stack steps in. Whenever a function is called, the current program counter, along with some other data, is pushed onto the stack, and the called functions memory address is entered into the program counter. At the end of the called function, the old program counter value is restored and the processor picks up where it left off.

It was mentioned that additional data is pushed to the stack as well during a function call. What this data is depends somewhat on the compiler as well as the instructions that the processor can execute, as not all processors have the same instruction set. When entering a new function, the local variables of the previous function must be saved for restoration, and space for the new variables created. One method some compilers and processors use is to push onto the stack the current contents of all the GPRs during a function call, and restore them afterwards. This is especially common in processors that have a "push all" instruction such as the x86 processors since the 80186 (these are processors common to desktop computers since the early days of home computing).

A second method of saving variables and the current state of the registers is for the called function to push to the stack only the values of the registers that it will actually use. This is common on processors with smaller memories, and therefore smaller stack sizes, such as most microcontrollers.

There are two types of errors relating to the stack; underflow and overflow. Of these, underflow is far less common and is most often found in software stacks, or as the result of malice. These are caused by attempting to pop a value off the stack when the stack is empty. Stack overflow errors, on the other hand, occur when an attempt is made to exceed the maximum size of the stack. Most often this error occurs when there have been too many function calls nested inside each other, such as with recursion. This error can be fixed by reducing the depth of function calls, or by reducing the memory requirements of each function.

1.4 Conclusion

The purpose of this chapter was to introduce some terminology and provide a very basic understanding of what a microcontroller actually is. From here on out each chapter will be covering some aspect of microcontrollers, their programming and their use in greater detail. This text is focused around the ATmega644p microcontroller produced by Atmel, so users of other companies' microcontrollers may need to go elsewhere for specifics on their devices. It is one of the goals of this text to provide enough background that the reader will be able to use any Atmel microcontroller at the very least, and figure out other companies' microcontrollers as well.

If the history or other subjects covered in this chapter are of interest, there are many sources available for further reading. A brief search on Google or at the local library should result in numerous sources of interest. Wikipedia

is a good initial source for additional information as well, however care must be taken with publicly editable sites such as that. Further resources should be available via links at the bottom of most wikipedia articles.

Chapter 2

Datasheets, SFRs and Libraries

Before beginning to examine a microcontroller in depth, there are a few important topics to learn about. These are topics which are closely related, or directly about microcontrollers that should provide a better understanding of what exactly is occurring in the subsequent chapters, and where the information presented is stemming from. This chapter is presented as three independent sections, each a mini-chapter in it's own right.

The first section, datasheets, dissects an example datasheet and explains the various portions thereof. The special function register section introduces what SFRs are and how to address them on a microcontroller. The final section in this chapter provides a basic understanding of what a library is and which are most commonly used in programming the ATmega644p.

2.1 Datasheets

Datasheets are the documents containing all the vital information that the manufacturer is supplying, which should answer most any question on the usage and operation of the component. Datasheets should be available, either directly from the manufacturer or most commonly on-line, for any electrical component one may wish to include in a project. This includes resistors, capacitors and other basic electrical components.

On first glance, datasheets can seem somewhat daunting with pages of graphs, tables and text. This is because the datasheet is presenting a wide variety of information about operation and usage. Generally, unless the component is to be used in extreme conditions, only a small percentage of the information is actually required, though tracking down the important information is where the skill of reading datasheets enters. Each section of the datasheet that is discussed will be accompanied by an example from STMicroelectronics' M74HCT08 Quad 2-Input AND Gate [18].

It is also important to note that there is no such thing as a standard datasheet. The following information is common, but by no means always present. Furthermore there may be multiple documents required to obtain the desired information. If the first data sheet found doesn't have the information, keep looking. There may be other documents covering different aspects of the device or generic aspects of an entire range of devices.

2.1.1 Part Name

The part name can often tell someone a lot of information. Besides stating what type of device it is (resistor, H-bridge driver, sensor etc.) The part name will also divulge further information. In the case of many ICs, the part name will reveal how many copies of the device exists on the chip. For example ICs containing the basic logic gates (AND, OR, NOT etc) regularly have four or even eight copies of the gate per chip. Other information that can be gleaned from the part name can involve the number of bits used for precision in devices such as analog-digital converters (ADC) or digital-analog converters (DAC), voltage ranges (op-amps, ADCs, DACs), or max power (resistors). The lesson here is to always read the full part name, just in case there's some extra piece of important information therein.

Example - Quad 2-Input AND Gate

The name of this device specifies a number of things. First, that his device contains AND logic gates. Secondly, that there are 4 independent gates on the device, and finally that each AND gate is limited to two inputs. AND gates require at least two, however it is possible to create AND gates with three or more inputs.

2.1.2 Description and Operation

The first few pages of a datasheet are often text. These pages contain a description of what exactly the device does, and often how it works, which can include the duration of delays of various operations, descriptions of what occurs and explanations of errors. This should be at minimum skimmed by anyone who is unfamiliar with the specific device, and those who are unfamiliar with the type in general should read the entire section.

Example

While the description is very short in this case, it does provide information about the sorts of other devices it can interact with (TTL and NMOS). This covers the majority of other logic components, and if in doubt the electrical characteristics can be examined as well.

DESCRIPTION

The M74HCT08 is an high speed CMOS QUAD 2-INPUT AND GATE fabricated with silicon gate C^2MOS technology.

The internal circuit is composed of 2 stages including buffer output, which enables high noise immunity and stable output.

The M74HCT08 is designed to directly interface HSC^2MOS systems with TTL and NMOS components.

All inputs are equipped with protection circuits against static discharge and transient excess voltage.

Figure 2.1: Description section of AND gate datasheet.[18]

2.1.3 Absolute Maximum Ratings

Most datasheets will include this section. It lists the maximum voltages, currents, and possibly temperatures at which the device can be used. Exceeding these limits can, and often will, damage or destroy the device in question.

Example

ABSOLUTE MAXIMUM RATINGS

Symbol	Parameter	Value	Unit
V_{CC}	Supply Voltage	-0.5 to +7	V
V_I	DC Input Voltage	-0.5 to V_{CC} + 0.5	V
V_O	DC Output Voltage	-0.5 to V_{CC} + 0.5	V
I_{IK}	DC Input Diode Current	± 20	mA
I_{OK}	DC Output Diode Current	± 20	mA
I_O	DC Output Current	± 25	mA
I_{CC} or I_{GND}	DC V_{CC} or Ground Current	± 50	mA
P_D	Power Dissipation	500(*)	mW
T_{stg}	Storage Temperature	-65 to +150	°C
T_L	Lead Temperature (10 sec)	300	°C

Absolute Maximum Ratings are those values beyond which damage to the device may occur. Functional operation under these conditions is not implied
(*) 500mW at 65 °C; derate to 300mW by 10mW/°C from 65°C to 85°C

Figure 2.2: Absolute Maximum Ratings section of AND gate datasheet.[18]

The far left column, symbol, is what is used elsewhere to refer to these parameters. An immediate example is the values for the input and output voltages. Both of these refer to V_{CC} which, according to the first line in the table, is the supply voltage. Since the supply voltage's maximum rating is 7V, the input and output voltages can be as high as 7.5V. Keep in mind, these are the maximum ratings. Designing a circuit to be pushing these ratings is more likely to cause failures, either due to occasional spikes over the maximum value or through added wear and tear. Most devices do not function as long when continually pushed to their limits.

Although not always present, the datasheet in question also provides a table of recommended operating conditions. These are somewhat lower voltages at which it is safe to continually run the device. These are the conditions that the design should not exceed. If the design indicates exceeding these limits, selected parts and voltage levels should be reassessed.

2.1.4 Electrical Characteristics

Aside from the maximum ratings, this section will contain the most important information when designing a circuit. This includes voltage limits, current limits, uncertainties and any additional information required for the successful operation of the device. While the maximum ratings supply the limits of what the device can safely accept, this section covers how the signals will be interpreted. This will often include three values per parameter, a minimum value, a typical value, and a maximum value. These indicate the range over which the device will interpret the signal in a specific way, as well as what value should be designed for. If the typical value cannot be exactly reached, it is not an issue, and generally not worth adjusting the signal assuming it falls within the minimum and maximum ratings. On occasion some of these values may not be present, and only a minimum or maximum may be given (with or without a typical value). If this is the case, it means that the missing value if often zero, or may be specified by a $\pm$ in another column. Many parameters will also have a test condition specified. The test condition specifies some aspect that may or may not be true in a given application (such as supply voltage or current), and means that any values given in that row of the table are only guaranteed if that test condition is true.

Example

DC SPECIFICATIONS

Symbol	Parameter	Test Condition		Value							Unit
		V_{CC} (V)		T_A = 25°C			-40 to 85°C		-55 to 125°C		
				Min.	Typ.	Max.	Min.	Max.	Min.	Max.	
V_{IH}	High Level Input Voltage	4.5 to 5.5		2.0			2.0		2.0		V
V_{IL}	Low Level Input Voltage	4.5 to 5.5				0.8		0.8		0.8	V
V_{OH}	High Level Output Voltage	4.5	I_O=-20 µA	4.4	4.5		4.4		4.4		V
			I_O=-4.0 mA	4.18	4.31		4.13		4.10		
V_{OL}	Low Level Output Voltage	4.5	I_O=20 µA		0.0	0.1		0.1		0.1	V
			I_O=4.0 mA		0.17	0.26		0.33		0.40	
I_I	Input Leakage Current	5.5	$V_I = V_{CC}$ or GND			± 0.1		± 1		± 1	µA
I_{CC}	Quiescent Supply Current	5.5	$V_I = V_{CC}$ or GND			1		10		20	µA
ΔI_{CC}	Additional Worst Case Supply Current	5.5	Per Input pin V_I = 0.5V or V_I = 2.4V Other Inputs at V_{CC} or GND I_O = 0			2.0		2.9		3.0	mA

Figure 2.3: Electrical Characteristics section of AND gate datasheet.[18]

As the M74HCT08 is a logic chip, it is necessary to know what constitutes

a logical low vs a logical high. Looking at the first two rows of the table, and referring to the recommended input voltages, any input between 2.0V and V_{CC} registers as high, and any input between 0V and 0.8V registers as low. This is only true if V_{CC} is between 4.5V and 5.5V. If V_{CC} is lower than 4.5V then the triggering voltages may also change. A voltage between the maximum low input (0.8V) and minimum high input (2.0V) should be avoided as the output will be unpredictable.

2.1.5 Physical Characteristics

There will be several tables covering a variety of characteristics of the device, including physical, electrical and thermal responses. These can appear in any order in the datasheet, but appear after the description of the device.

The physical characteristics table will often be accompanied by a diagram of the exterior of the device. The table will proceed to list dimensions, mounting requirements etc., everything someone would need to know to physically fit the component onto a PCB or whatever medium is supporting the circuit. While eventually vital for the final designs of the physical circuit, during the initial design and prototyping, this section can mostly be ignored. The most important information in this section at that point is determining whether there is a package available that will easily fit in the prototype, such as the classic through-hole dual-inline-package (DIP) design of ICs which fit nicely into breadboards.

Example

On the AND gate datasheet, the physical characteristics are three full-page descriptions of the packages in which the chip is available, with all the dimensions necessary to use them. As they are full page they are not copied here.

2.1.6 Other Information

The other available information will vary depending on the device. For devices which relate to temperature (thermocouples) or whose characteristics may change with temperature, there will often be graphs linking variable parameters to internal or external temperatures. Motors and other actuating devices will regularly have power/velocity curves of some variety. Whenever looking at a new datasheet, skim all the charts for anything mentioned in the description, or anything unknown and determine which of these are important to the task at hand, and remember, if the needed information can't be found, look for additional documents on the device or family of devices.

Complex devices, such as microcontrollers and other integrated circuits, will often have a schematic of the interior workings of the device. Examples of these schematics, and descriptions thereof, can be found in Chapter 3: Hello World and Chapter 4: Analog Digital Converter of this text.

2.2 Special Function Registers

The datasheets for microcontrollers are generally several times that of other devices, often running into the hundreds if not thousands of pages. This is due in part to the number of devices, called peripherals, that are internal to the microcontroller, but also due to the required listings of all the special function registers (SFRs).

SFRs are specific locations in the memory map that have predefined tasks. The memory map is the linking of a memory address, which is the value used to determine where the desired information is, to the physical location of that data. For the majority of memory addresses, these locations are within a block of RAM (random-access memory). For SFRs, however, these addresses point to buffers, latches or other devices in the hardware related to each peripheral, not to the RAM. Each SFR is connected to one of the various peripherals in the microcontroller (ADC, timer, input/output etc.) or to the processor itself. The SFRs serve as the connection between the software and hardware. The majority of these registers are for settings and status information, with every bit field meaning something different. A bit field is one or more bits that relate to the same very specific operation, such as a 2-bit field choosing one of four modes of operation. Each register can contain several bit fields relating to different options on the peripheral. The datasheets must therefore list not only each register, but each bit of each register and explain what they do.

2.2.1 SFRs in Datasheets

Although specifics change from manufacturer to manufacturer, the sections of the datasheets relating to SFRs are always similar. For each peripheral in the microcontroller, a list of related SFRs is given. Each SFR entry has a listing of each bit, and what they do, including whether the can be read or written. Elsewhere there will also be a large table of all the SFRs and their memory addresses, which may also include the bit names, though without the descriptions. Figure 2.4 presents an example from the ADC of Atmel's ATmega644p.

This is one of the control and status registers for the ADC. The first line gives the name of the register as used in the libraries and elsewhere in the datasheet, followed by a more human-readable name. Note that all the register and bit names are some form of abbreviation of their full name. The small table below the register name is a listing of the bits. Each bit has a unique name. The second to last row of the chart indicates whether the bit can be read and/or written. An 'R' indicates that the bit can be read, and a 'W' for written. All the bits in this register can be both read and written. The final row gives the default value of the bit when the microcontroller is first powered on. In most cases this is 0, but not always.

Following the table is a listing of each bit: its bit number, its code name, its full name and a description. Often times the description is just text, however on occasion there is a table as well. Tables most often occur when there is a bit field of more than one bit, such as bits 0 through 2 in this register. Wider bit

20.9.2 ADCSRA – ADC Control and Status Register A

Bit	7	6	5	4	3	2	1	0	
(0x7A)	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7 – ADEN: ADC Enable**

Writing this bit to one enables the ADC. By writing it to zero, the ADC is turned off. Turning the ADC off while a conversion is in progress, will terminate this conversion.

- **Bit 6 – ADSC: ADC Start Conversion**

In Single Conversion mode, write this bit to one to start each conversion. In Free Running Mode, write this bit to one to start the first conversion. The first conversion after ADSC has been written after the ADC has been enabled, or if ADSC is written at the same time as the ADC is enabled, will take 25 ADC clock cycles instead of the normal 13. This first conversion performs initialization of the ADC.

ADSC will read as one as long as a conversion is in progress. When the conversion is complete, it returns to zero. Writing zero to this bit has no effect.

Figure 2.4: ADCSRA Register Excerpt from ATmega644p datasheet.[11]

fields can be recognized because their names are identical except for a number at the end. When referring to the bit field as a whole either the number can be left off (`ADPS`) or the size of it can be specified by giving a range of bits (`ADPS2:0`).

- **Bits 2:0 – ADPS2:0: ADC Prescaler Select Bits**

These bits determine the division factor between the XTAL frequency and the input clock to the ADC.

Table 20-5. ADC Prescaler Selections

ADPS2	ADPS1	ADPS0	Division Factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8

Figure 2.5: ADPS2:0 Bit field from ADCSRA Register from ATmega644p datasheet.[11]

2.2.2 Addressing SFRs

As SFRs exist in the memory map, each one has a unique memory address. While they can appear anywhere in data memory, they are generally either at the bottom (near address 0x00) or near the top of the memory map. There may also be a number of general purpose registers (GPRs) which are used as temporary

storage and do not relate to the peripherals. Both PIC microcontrollers and Atmel's AVR microcontrollers place the SFRs at the bottom of the data memory map, along with a number of GPRs.

Within the code, there are a few methods of accessing the contents of the SFRs, depending on how the code libraries are set up. PIC microcontrollers, for example, use prebuilt data structures called `struct`s to access the individual bit fields of the SFRs, while AVR microcontrollers use bitmasking on predefined variables. As this text focuses on the ATMega644p microcontroller from Atmel's AVR line of microcontrollers, bitmasking is the method that will be examined here. For additional information on using `struct`s, both in general and for accessing SFRs, refer to chapter 6.

In and amongst the numerous files included in WinAVR (the development kit for AVR microcontrollers for Windows users), is a file called `iomxx4.h`. This is the file that maps all the SFR register addresses to names for the developer to use for the family of microcontrollers the ATmega644p belongs to. In it, for every SFR on the microcontroller, is a line similar to this:

```
#define      ADCSRA       _SFR_MEM8(0x68)
```

This statement tells the compiler that any time the name `ADCSRA` appears in the code, to replace it with `_SFR_MEM8(0x68)`. This function is defined elsewhere in the assorted libraries and accesses the given location in the memory map, in this case `0x68`. After each register definition is a listing of each bit in the register, defining each name to be the bit's location in the byte.

```
#define    ADIE    3
```

The above line of code causes any appearance of `ADIE` to be replaced with `3`, as `ADIE` is the fourth bit in the `ADCSRA` register. Bits are zero indexed in a byte, which means that the first bit is called bit 0.

With all the registers and individual bits named, it is possible to read and write to the SFRs. Technically its possible to not use the names, however it becomes much more difficult both on the initial programmer and on anyone who needs to look at the code later to understand what is going on. To this end, there are several methods of reading and writing to the registers.

Reading an SFR

Retrieving data from an SFR can have several purposes. It can be used to retrieve data, inquire about the specific status of the peripheral, or be used as part of writing a specific bit (see below). The simplest case is when the contents of the entire SFR need to be read, such as when accessing the results of an analog to digital conversion:

```
char ADCResult = ADCH;
```

This reads the entire contents of the `ADCH` register and saves it to the single byte variable `ADCResult`. If only specific bits are desired, a bitmasking process must be used.

```
char ADCEnabled = ADCSRA & (1<<ADEN);
```

This retrieves the value of only the `ADEN` (ADC Enable) bit of the register using a bitwise AND operation. A bitwise and compares each bit of the left argument (`ADCSRA`) with the corresponding bit of the right hand argument (`1 <<ADEN`) and saves the result of each of these comparisons in the corresponding bit in `ADCEnabled`. If the ADC were enabled, the result would be `0b10000000` else it would be 0, as the `ADEN` bit is bit 7. Another option is to replace `(1 <<ADEN)` with `_BV(ADEN)` which accomplishes the same purpose. The function `_BV()` is defined in the libraries supplied with the compilers to have the same result as the left shifting operation performed in the code example above.

Writing to an SFR

While writing a specific value to an SFR is easy, it is not commonly done. Far more often only a single bit or two need to be changed, which becomes more complicated as then entire byte must always be written simultaneously. To avoid changing the other bits, they must first be read and then masked to prevent them from changing. To set a specific bit, use a bitwise OR operation such as:

```
ADCSRA = ADCSRA | (1<<ADEN);
```

After this executes, the ADC will be enabled whether or not it was before. In order to clear a bit however, the register must be bitwise ANDed with a bitwise NOT of the left-shifted bit.

```
ADCSRA = ADCSRA & ~(1<<ADEN);
```

After being executed, this line will ensure that the ADC is disabled without changing any other bit. This works because any bit that was enabled, excepting the `ADEN` bit, will result in an enabled bit in the final result as it is being ANDed with another high bit. The `ADEN` bit is the only bit in the right hand argument that is not high after the bitwise not.

While the concept of SFRs, their names and the logic required to access them may seem daunting at the moment, they will become second nature through use and the logic will become a matter of recognizing a pattern rather than working out the steps every time.

2.3 Libraries

Libraries are vital components to any form of programming. They extend the capabilities of a programming language by supplying functions which can be reused from project to project without the need to rewrite the functions every time. If they did not exist, programming anything would be a far longer and more labor intensive task.

A library is a collection of pre-defined, and often pre-compiled, functions, definitions, types or classes (in a language which uses classes). In short, a library can contain any part of a program except the `main` function and interrupt service

routines (ISRs, see chapter 8). The contents of each library are in some manor related, and often call upon the contents of other libraries.

Libraries sole purpose is to increase code re-usability. Once written, libraries can be used in any program to provide the same capabilities as the previous program it was used in. An example of this would be the common math functions in `math.h`, which is a library containing a variety of commonly used math functions, such as to find a square-root or the various trigonometric functions. These functions are commonly used, however can be difficult to implement. The trigonometric functions are often implemented via a look-up table rather than actually calculating the precise value as it is more efficient. If every programmer had to re-implement these functions for every program they wrote, the time lost would be immense.

While the majority of necessary libraries are supplied with the compiler, or available from other sources, libraries can also be user created. Any time a function is written that is likely to be needed again elsewhere, it should be added to a library. These user-created libraries are often stored somewhat differently. Libraries obtained from other sources are generally pre-compiled and are header files and object files (.h and .o respectively in C). User-created libraries are often left uncompiled, and are compiled into any program they are used in, which allows the user to more freely modify and add to the contents of the library.

Libraries are extremely useful and time-saving constructs. Some care needs to be taken in their creation and use, however. Firstly, user-created libraries should be stored in some central repository rather than copied into each program separately. By referencing every program to the same file, rather than copies thereof, correctly errors and adding functionality need only occur once. Otherwise keeping track of what program uses which version of a library can become a nightmare. Secondly, once a function is created and used in a program, its name, arguments and return type cannot be changed without changing every call to the function in every program. The functions should therefore be written to be as generic as possible to avoid having to add to it or create multiple similar functions later. One method to help keep it general is to use a pointer to a `struct` as the sole argument. This allows the function to be modified by changing the `struct`, which is less likely to cause issues than attempting to change the function call itself. For more information on pointers and `struct`s and their uses, refer to chapter 6.

2.3.1 Including libraries

Whenever a library is to be used in a program, the compiler must be told to include it using a `#include` line at the top of the source file or header file (see chapter 5 for more information on header files). This `#include` statement tells the compiler what file to look for, and where it can be located. There are two syntaxes for including libraries. The first uses angle brackets (`<`,`>`) and tells the compiler to look for the library somewhere within its defined library structure. These are the libraries that were specifically installed with the compiler, as well as any other folders it has been instructed to look in via the /I compiler option.

If the library exists within a sub-folder of one of these locations, the entire path from the base point the compiler looks at must be included. Examples of these paths are the `avr\io.h` and `utils\interrupt.h` libraries mentioned below.

The second syntax is for giving a relative system path and uses double quote marks (`"`). When looking for these libraries, the compiler begins in the folder containing the program files and browses the system file structure from there. For example, to reach a library called `ADClib.h` in the folder Libraries, which is contained in the same parent folder as the project, the command would be: `#include "..\Libraries\ADClib.h"`. No semi-colon is included after either of these syntaxes.

For examples of files using `#include` instructions, refer to appendix B.

2.3.2 Commonly used libraries

These libraries are specifically those commonly used on the ATmega644p microcontroller, a member of the AVR family of microcontrollers made by Atmel, though the ones created and distributed by Atmel are usable on many Atmel microcontrollers. Logic internal to the libraries handles the microcontrollers specific needs. Paths will also be given for the libraries that require them.

- `<avr\io.h>`: This library, and the libraries it calls, includes defined names for all of the SFRs on the microcontroller. These names allow access to the registers without needing to know their memory addresses. This library is used in almost every single program written for the AVR family of microcontrollers.

- `<utils\delay.h>`: This library contains two functions for causing the program to wait for certain periods of time. These functions are `_delay_ms` and `_delay_us`: which cause the program to wait for a given number of mili- or micro-seconds respectively. This library only needs to be included if one or both these functions are required. See chapter 8 for more information on timing events.

- `<utils\interrupt.h>`: This library includes several functions and defined values for use with interrupts, and is required if interrupts are to be used. See chapter 8 for more information on interrupts.

- `<math.h>`: This library is a generic library available for most any platform. It contains a variety of commonly used math functions, including exponential and trigonometric functions.

- Peripheral libraries: Individual libraries for the various peripherals on the ATmega644p are not supplied with the compiler. It is highly advantageous to create libraries to handle common tasks for the various peripherals such as initialization or any reading/writing tasks that may be required.

There are numerous other libraries available for use. Check the documentation or explore the libraries themselves to find out more.

2.4 Conclusion

Data sheets and their contents, as well as the names and contents of SFRs, are incredibly diverse, with each company doing things their own way. Practice and experience will help a great deal, so don't feel bad if they seem daunting at first. Also, remember that if the information does not seem to be available there may be another document published about the device's family or series as a whole.

Chapter 3

Hello World

The first task for most any new programming environment, language or device is some version of a 'Hello world' program. This is one of the simplest programs possible because its goal is to test that the environment is setup properly and to display the basic syntax of the programming language and specific environment. In the case of microcontrollers this generally consists of turning an LED on or off.

In any sort of controlled system, there are four basic parts: the controller, the control interface, the plant, and the feedback interface. Generally the controller is a microcontroller. The microcontroller may be dedicated to this single task, or handle a few related subsystems. Due to the abundance and low cost of microcontrollers, it is common to use multiple microcontrollers in a single system, each one handling a different task with a final microcontroller sending commands to each of the low-level microcontrollers.

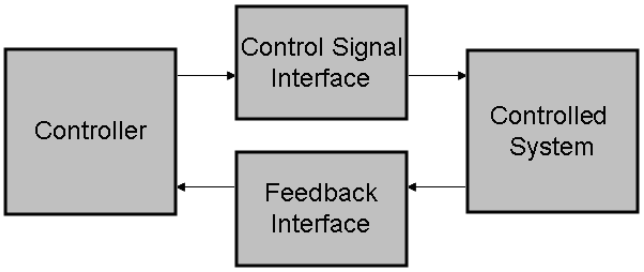


Figure 3.1: Diagram of a basic controlled system

The plant is the device to be controlled. This can be a motor, a sensor, or the controller for a further subsystem. This is most often the first part of the system to be defined, and may be immutable in the case of using commercially available products.

In nearly every conceivable system, merely connecting the controlled system to the microcontroller directly to the controller will either destroy one of the

components or nothing will happen. For example, attempting to drive a motor with the power that a microcontroller can supply will likely destroy the microcontroller before the motor moves. Therefore there must be some component or subsystem that converts the signal that the controller can safely provide into a signal that the controlled system can actually use. This could be as simple as a binary on/off signal being converted into the current necessary to light an LED, or more complicated such as the information a computer sends to a printer to print out a document, and can be in any number of formats or standards. The feedback interface performs the same role in reverse. Any information the plant needs to return to the microcontroller must be translated into something the microcontroller will understand. In addition to simply translating data, both these interfaces may need to adjust the voltage or current of the signals to prevent damage or supply the necessary power to make the plant operate. Finally, not all systems require both interfaces. Some operate with just one or the other.

In the case of controlling an LED, the system components are the microcontroller, the connection to the LED including any necessary support circuit, and the LED itself. In this example there is no feedback.

3.1 Example: Lighting an LED

The next two sections will cover the process for using a microcontroller to turn on and off an LED. This section will cover the electronics involved and section 3.2 will cover the programming required. While many students or hobbyists would simply throw a few components together and decide the results are either adequate, disastrous or in need of refinement, the proper procedure is far more in depth, and has the benefits of being scalable to any system being designed, and also of preventing the accidental destruction of any of the electrical components involved.

It is important to note that this is an iterative process. If each step is completed without thought for the subsequent steps, and without going back to make changes to previous steps, systems will often end up being far more complicated, inefficient and expensive than they need to be. While this example may seem to work first try, it is because of careful consideration and multiple iterations to create as simple a system as possible.

Step 1: Determining requirements and design constraints

Before doing anything else, it is vital to decide what the final results of the system are. Whether it is to lift a five-ton crucible of molten steel or to let a user know a machine is on, these requirements will lay the basis for the entire system. Additionally, defining a few design constraints or goals can help in choosing between several possibilities that fit the requirements. These goals often relate to reliability, cost, and simplicity. For this example the requirements are quite simple: turn on and off an LED which is bright enough that the on/off state is easily identifiable in a well-lit room. Meanwhile the goal is to have the

control interface as simple as possible, which means avoiding the use of any form of buffer or level-shifting.

Step 2: Selecting the LED

There are hundreds of varieties of LEDs on the market, varying in color, brightness, package size, voltage and current requirements, and view angle among others. All these factors need to be considered and determined what values are desired and how important it is to match that value. For this example, the most important requirement is that the brightness of the LED be high enough to be visible. Two additional characteristics, the forward voltage and current, must be considered due to the design constraints, namely attempting to keep the LED's requirements within the ranges that the microcontroller can supply, which will be addressed more later. These limits, along with several arbitrary decisions on color and package, lead to the choice of a red LED in a round 5mm package from Lite-On Electronics part number LTL-2F3VEKNT; for which the datasheet is available through an on-line search.

Figure 3.2 contains an excerpt from this datasheet, and contains the three pieces of information required: luminous intensity, forward voltage and the forward current test condition. As stated in the requirements, the brightness of the LED is paramount. According to the datasheet, the selected LED emits a minimum of 4200 mcd (milli-candela) at a test condition of 20mA. One candela (1000 mcd) is defined such as to be approximately equal to the brightness of a common candle, which is bright enough to be readily seen in a well-lit room. This LED has over four times as luminous intensity as is necessary, and therefore meets that requirement.

Parameter	Symbol	Part No. (LTL)	Min.	Typ.	Max.	Unit	Test Condition
Luminous Intensity	Iv	2F3VRKNT	3200	5500		mcd	I_F = 20mA Note 1 Note 2
		2F3VEKNT	4200	7200			
		2F3VHKNT	4200	7800			
		2F3VAKNT	4200	7800			
		2F3VFKNT	4200	7800			
		2F3VYKNT	4200	7800			
		2F3VSKNT	4200	7200			
Forward Voltage	V_F	2F3VRKNT		1.9	2.3	V	I_F = 20mA
		2F3VEKNT		2.0	2.4		
		2F3VHKNT		2.0	2.4		
		2F3VAKNT		2.0	2.4		
		2F3VFKNT		2.0	2.4		
		2F3VYKNT		2.0	2.4		
		2F3VSKNT		2.0	2.4		

Figure 3.2: Excerpt from data sheet for LED part LTL-2F3VEKNT.[16]

The forward voltage is the voltage required to turn on the LED. If the forward voltage level is not met, the LED will not turn on and will act like an open circuit. For the LED chosen, this level is typically 2.0V, but can be as high as 2.4V. The final piece of information is the test condition forward current. This is the maximum recommended current to be allowed to flow through the

LED, and is also the current which will provide the rated brightness. Lower currents will provide less light.

Step 3: Determining microcontroller capabilities

In general, the microcontroller used for a given application will be highly dependent on the requirements of that application. There is no need, for example, to use a high-end 32-bit microcontroller to control an LED. For the purposes of this example, the ATmega644p produced by Atmel will be used. Virtually any information that one could ever want to know about the technical specifications of a microcontroller can be found in the data sheet. Starting on page 325 of the datasheet for this microcontroller, all the electrical characteristics of the microcontroller are available, including the absolute max ratings and the test conditions. These test conditions are the generally accepted safe limits for long-term applications, so should be used as the design limits of the system.

Figure 3.3 contains a few important pieces of information. First it provides the current limits for both sinking (input) and sourcing (output) at multiple voltages. Additionally, notes three and four specify maximum currents for a group of pins. This is a limit that must be examined if the microcontroller is performing several tasks, such as controlling whole banks of LEDs. In the case of this example, a maximum of 20mA at 5V is available for powering the LED.

V_{OL}	Output Low Voltage[3],	I_{OL} = 20 mA, V_{CC} = 5V I_{OL} = 10 mA, V_{CC} = 3V			0.9 0.6	V
V_{OH}	Output High Voltage[4],	I_{OH} = -20 mA, V_{CC} = 5V I_{OH} = -10 mA, V_{CC} = 3V	4.2 2.3			V

Notes:

3. Although each I/O port can sink more than the test conditions (20 mA at VCC = 5V, 10 mA at VCC = 3V) under steady state conditions (non-transient), the following must be observed:
   1.)The sum of all IOL, for ports PB0-PB7, XTAL2, PD0-PD7 should not exceed 100 mA.
   2.)The sum of all IOL, for ports PA0-PA3, PC0-PC7 should not exceed 100 mA.
   If IOL exceeds the test condition, VOL may exceed the related specification. Pins are not guaranteed to sink current greater than the listed test condition.
4. Although each I/O port can source more than the test conditions (20 mA at VCC = 5V, 10 mA at VCC = 3V) under steady state conditions (non-transient), the following must be observed:
   1.)The sum of all IOH, for ports PB0-PB7, XTAL2, PD0-PD7 should not exceed 100 mA.
   2.)The sum of all IOH, for ports PA0-PA3, PC0-PC7 should not exceed 100 mA.
   If IOH exceeds the test condition, VOH may exceed the related specification. Pins are not guaranteed to source current greater than the listed test condition.

Figure 3.3: Excerpt from pp325-326 of datasheet for ATmega644p regarding maximum sink/source capabilities.[11]

Designing the control interface

Step 4: Comparing system requirements to controller abilities

Now that both the system and controller blocks of figure 3.1 have been defined, it is time to determine what is necessary for the control interface. The first step of which is to compare what is available from the controller to what is required by the system. In this case the maximum current is the same for both (20mA), and the microcontroller can provide the necessary 2.0 to 2.4V. As the

controller can provide everything necessary, the control interface can be kept fairly simple, not requiring any form of level shifting, amplification, or external current source.

Sink or source

There are two possible ways to setup the LED circuit. The first method is to have the microcontroller be the power source for the LED, and run the digital output of the microcontroller through the LED and a resistor to ground as shown in figure 3.4.

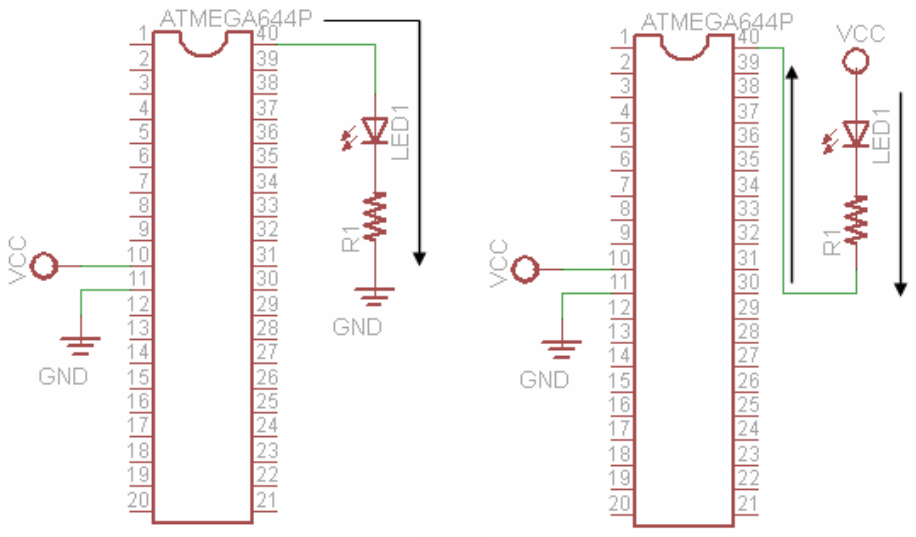


Figure 3.4: Circuit diagram for using the microcontroller as the current source (left) or as a current sink (right).

The first method, using the microcontroller as the current source, is the left hand schematic in figure 3.4. This connects the LED through a resistor to ground. When the pin on the microcontroller it is connected to is set to a logical high, power flows in the direction shown by the arrow to ground, lighting the LED. The second method, using the microcontroller as a sink, is the right hand schematic. In this method the LED is permanently connected to a voltage source. When the microcontroller's digital output is set high, the pin is set to the same voltage causing a voltage difference of 0V. As this difference is the important value, no current flows. When the pin is set to low, it is connected to ground. This allows current to flow in the direction of the arrows into the microcontroller, lighting the LED. In this case the logic for turning on the LED is reversed from the previous method, in that a logical high will turn off the LED as there is no voltage difference between V_{CC} and Chip DO. This is known as active low, as opposed to the active high method above.

As previously determined from the datasheet, the sink and source capabilities of the ATmega644p are the same at 20mA per pin. This is not always the case, however, as many microcontrollers can sink more current than they can source. The second method, sinking the current and active low logic, is therefore more commonly used, and the method that will be used here.

Step 5: Resistor selection

By itself, an LED has an insignificant resistance when turned on. This would cause a very high current to flow through it, likely burning out both the LED and the microcontroller in the process without the inclusion of the current limiting resistor seen in figures 3.3 and 3.4.

Determining the value of the resistor needed depends on several variables: the supply voltage, the diode forward voltage, diode forward current, and sink current of the microcontroller.

As determined earlier, the selected LED has a forward voltage of 2.0V and forward current of 20mA. The resistor therefore must be sized to allow a maximum of 20mA through with a voltage drop of the remaining 3V of the 5V supply. Using Ohms Law:

$$V = IR$$
$$R = V/I$$
$$R = 3V/20mA$$
$$R = 150\Omega$$

This is the minimum resistor size necessary, so it is important to always round up to the next standard resistor value. This is true even if the calculation results in a standard value. All resistors have a tolerance, which means that the actual resistance will fall within a percentage error often 5 or 10% for inexpensive, commonly available resistors. Were a 10% tolerance resistor to be used in this application, choosing a 150Ω resistor could result in actually using a 135Ω resistor, causing the actual current to be higher than the rated 20mA. By rounding up to 220Ω, the next standard value, a current below 20mA is guaranteed.

In many cases it may be advantageous to reduce the current drawn from or carried to the microcontroller further. It is often preferred to reduce the current as low as possible while still achieving the minimum requirements, and the reasons are numerous.

- Power: According to Joule's law, power is equal to current multiplied by voltage. As that power has to come from somewhere, reducing the current flow will reduce the power consumed. If that power happens to stem from a battery, the batteries life will be improved.

- Heat: Whenever power is absorbed by something, such as a resistor, it will eventually result in heat. This heat is a waste of power, can adversely

affect long term performance, and can be dangerous to users if the heat generation is too high.

- Reliability: Increased current, especially when nearing the maximum, will cause increased wear and tear on components, reducing its reliability and lifespan. Excess heat can exacerbate the problem.

- Global limits: As was mentioned before in section 3.1 in step 3, there is a limit to the current that groups of pins can sink or source, which is far lower than the sum of the limits of the individual pins. If, therefore, eight LEDs were to be controlled, one per pin on port B, each pin would be limited to 12.5mA to stay within the 100mA limit; and this still assumes that port D and `XTAL2` are unused. By reducing the current to the minimum necessary instead of the maximum allowable, more devices can be run simultaneously.

In prior steps there had been an assumption about a current of 20mA. At this point the current will be lower than that, which means that while no components will fail the brightness needs to be rechecked. From the last page of the data sheet for the LED, it can be determined that the luminous intensity is directly proportional to the current. Calculating the current using the 220Ω resistor, 13.6 mA, and using this in the linear relationship, the LED is found to have a luminous intensity of 2856 mcd. While significantly lower than the initial 4200 mcd, it is still well above the level of a candle (1000 mcd). Using the concept of calculating for the minimum required, the current must be at least 4.76mA, so the maximum usable resistor value would be 630Ω.

3.2 Programming "Hello World"

As with most "Hello World" programs; this one is very short. It will be presented in its entirety and then broken down piece by piece. A basic understanding of the C programming language will be helpful, though not necessary for this. Before jumping in and using peripherals in a microcontroller, one should first look at the datasheet, read about the operation of the device and learn what each of the associated registers are for. On the ATmega644p, digital I/O utilizes three types of registers, with each of the four 8-bit ports requiring one of each type, plus an additional register relating to pull-up resisters for a total of 13 registers. Port designations have been generalized to 'x' here. Section 4 of this chapter contains further information on how these registers operate and interact in the hardware.

- `PORTx` - Port x Data Register
  The four `PORTx` registers control the signals being generated for digital output. Each bit corresponds to a separate pin on the microcontroller. If the port is set for any or all pins to be input, a 1 in the corresponding bit(s) of this register will enable a pull-up resistor on that pin. For more

information about controlling the pull-up resistors and their purpose, refer to section 3.4 later in this chapter.

- `DDRx` - Portx Data Direction Register
  These registers control whether a given pin on the port is an input or an output. If the bit in the register is high, the corresponding pin is configured as an output. The default when the ATmega644p is powered on is that all pins are inputs.

- `PINx` - Port x Input Pins Address
  This register reads the current value of the pin, regardless of the data direction. When a pin is set to input, the pin is not connected to anything else and the pull-up resistor is off, the value of the corresponding bit may randomly change. If the pull-up resistor is turned on, the pin will read high when it is not connected. If the data direction is set to output, this register will match `PORTx`.

- `MCUCR` - MCU Control Register
  Only bit four of this register matters to the digital I/O. This pin, `PUD` is a global disable for the pull-up resistor. When set to 1, the pull-ups are disabled even if `DDRx` and `PORTx` are configured to use them.

Now, knowing what each of the register do, they can be used in an application.

```
#include <avr/io.h>

int main(void) {
    DDRA = 0xff;
    PORTA = 0xfe;
    return 0;
}
```

This program will turn on an LED connected to pin 40 of the ATmega644p and then ends (pin configuration is shown in figure 1 in the Preface). Due to the nature of microcontrollers this LED will remain on as long as power is supplied to both the microcontroller and the LED. In order to better understand how this program works, each line is explained below.

```
#include <avr/io.h>
```

This line tells the compiler to include the specified header file (`avr/io.h`). Almost every program written for any AVR series processor will include this line, as it in turn includes additional files that define much of the memory structure such as the names of all the registers. Most microcontroller manufacturers or compilers will include such a header file to define names for all the registers in their microcontrollers Did this not exist, the programmer would be forced to address the registers by memory address, rather than by name.

```
int main(void) {
```

This is the main function. Every program must have exactly one main function and it is where program execution starts. This line also specifies that the program does not take any information as input (`void`), but it does return an integer upon completion (`int`).

```
DDRA = 0xff;
```

This sets the contents of the register `DDRA` to the hex (`0x`) value `ff`, which turns all 8 pins to output. Only one of the pins is actually used, but setting them all does not cause any issues in this case.

Whenever there are several registers serving the same purpose for multiple devices, the actual register names are often generalized by replacing the distinguishing character, in this case A, with a generic character indicating that it could be any of the possibilities. This generic character is generally a lower case `'n'` for numerical distinguishers or `'x'` for alphabetic. Therefore, when some reference is made to `DDRx`, it is referring to any or all of the data direction registers rather than a specific one. This 'x' means something different when preceded by a '0' and followed by a string of numbers or letters a-e. In this case the '0x' is a designator that the following value is in hexadecimal (base 16).

```
PORTA = 0xfe;
```

This sets all of the bits in the data register to high except for bit 0. As the circuit is using active-low logic, with the microcontroller acting as a sink, this turns the LED on. As a value of 0 is default here this line could be left out, however including this line will cause any LEDs attached to the other pins on the port to be off. Additionally, the value `0xfe` can be replaced by any syntax that gives the same value. This syntax is for hexadecimal, other options include binary (`0b11111110`) or decimal (`254`). Hexadecimal is commonly used due to its compactness and easy conversion to and from binary (each character in hexadecimal corresponds to a block of four bits).

```
return 0;
```

The final line of code supplies the integer requested as the return value in the function declaration. The `return` statement has the added effect of exiting the function immediately. While not necessary in this application, many compilers will throw errors or warnings if the main function does not have a return type of int or if execution of the function can reach the end of the function without encountering the return command.

```
}
```

While the `return` statement tells the program to exit the function, this bracket signifies that the definition of the function is complete, and any further lines belong to some other aspect of the code.

3.3 Digital Input

Detecting a dead LED

The ability to control a system if useful, however some form of determining the result of the control signal is also generally required. Feedback is necessary to account for vagaries in the system and unforeseen circumstances. Continuing the example of lighting an LED from before, the feedback desired would be whether the LED is properly functioning. While there are several methods that can be used, including various forms of optical sensors, the method used here will take advantage of the fact that LEDs generally become an open circuit (allowing no current to flow) or a short circuit (providing no resistance to the current) when they fail.

Step 1: Determining feedback requirements

Before any steps toward designing the feedback interface can be taken, requirements must be laid down as they were for designing the control interface.

- Feedback signal must differentiate between an LED working under expected parameters (2V forward voltage, under 20mA) and a broken LED
- Feedback signal must be interpretable by the digital inputs on the microcontroller
- Broken LEDs can be either a short or open circuit, it is not important to distinguish between the modes of failure
- Easily implemented on a solder-less breadboard

The second requirement necessitates a return to the microcontroller datasheet. Figure 3.5 shows the relevant information from the Electrical Characteristics chapter. The first row in the table states that for a signal to be guaranteed to be read as low, the signal can be no greater that 0.3 times V_{CC} or 1.5V at a 5V V_{CC} as is being used here. Similarly, the second line says that a high signal must be at least 3V at a 5V V_{CC}.

Symbol	Parameter	Condition	Min.	Typ.	Max.	Units
V_{IL}	Input Low Voltage,Except XTAL1 and Reset pin	V_{CC} = 1.8V - 2.4V V_{CC} = 2.4V - 5.5V	-0.5 -0.5		$0.2V_{CC}$ (1) $0.3V_{CC}$ (1)	V
V_{IH}	Input High Voltage, Except XTAL1 and RESET pins	V_{CC} = 1.8V - 2.4V V_{CC} = 2.4V - 5.5V	$0.7V_{CC}$ (2) $0.6V_{CC}$ (2)		V_{CC} + 0.5 V_{CC} + 0.5	V

Notes: 1. "Max" means the highest value where the pin is guaranteed to be read as low
2. "Min" means the lowest value where the pin is guaranteed to be read as high

Figure 3.5: Excerpt from pp325-326 of datasheet for ATmega644p regarding digital input voltage levels.[11]

Step 2: Adjusting control interface

There are two reasons the previously designed control interface is no longer sufficient. The first is for safety's sake in event that the LED is burnt out. Should the LED become a short circuit, the 220Ω resistor would then be absorbing the entire 5V instead of merely 3V. This would cause the current to increase to 22.7mA, which is above the safe limit of the microcontroller as determined in step 3 of the first example. The second reason to redesign the control interface is to simplify the feedback interface in the long run. As the method of feedback hasn't been fully determined yet, this step may be returned to later for further modification to the control interface.

New resistor value

As mentioned above, the resistance in the control interface must be increased to prevent a shorted LED from damaging the microcontroller. Therefore, instead of designing for a 3V drop across the resistor, designing for a 5V drop will protect the microcontroller.

The next higher standard value for resistors is 270Ω, and this is a perfectly acceptable value, however it is possible to have a 260Ω resistance by using a 110Ω and a 150Ω resistor in series. This is what will be used for reasons described later.

Recheck design requirements

Due to the change in the resistance, the current through the LED has dropped to 11.5mA while everything is working correctly. This will reduce the luminous intensity, so the design requirements must be rechecked to ensure the modifications do not violate any of them. As luminous intensity increases linearly with current, this new current will result in a luminous intensity just over half the test condition; specifically 2415mcd. As this is still over the 1000mcd minimum, the design requirements have not been violated.

Step 3: Available feedback signals

As the feedback is an attempt to determine, in essence, if the voltage drop across the LED is what is expected, perhaps the easiest method for determining this drop is to determine if the voltage drop across the resistors is the expected value. This voltage drop can then be compared to various break points to determine what the LED is doing using a comparator circuit.

Due to the resistor choice made above, there are now several options for where to obtain the voltage level for the feedback: just after the LED (nominally 3V), after the LED and the 110 resistor, or, swapping the order of the resistors, after the LED and the 150 resistor (see figure 3.6).

Table 3.1 contains information on the voltage levels for each of the three options for normal operations (on), as well as the two modes of LED failure

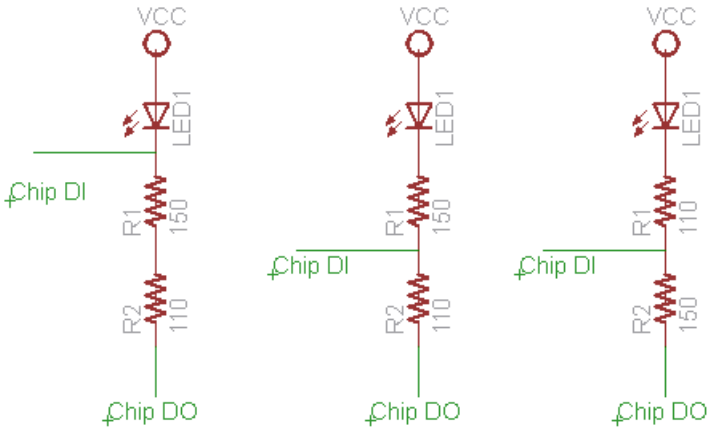


Figure 3.6: Three options for feedback signal.

(short, open). The first column denotes the resistor or resistors that the voltage is being measured across.

R value	On (V)	Short (V)	Open (V)
260	3	5	0
110	1.26	2.11	0
150	1.73	2.88	0

Table 3.1: Voltage values for the various feedback possibilities.

Step 4: Feedback interface

After inspecting table 3.1, it should be apparent that none of the three possible signals will fulfill the requirements without some form of interface to the microcontroller. As the desired result is a the digital result of whether the feedback voltage is at a desired value, a pair of comparator circuits can be used to compare the feedback voltage to set points between the operating voltage and the voltages of each of the failure modes.

The way a comparator circuit such as the one in figure 3.7 works is as follows. If VREF is at a higher voltage than VIN, the output is tied to the negative power supply for the comparator (often -V_{CC} or ground), otherwise the output is left 'floating' with a high impedance to ground. Please note that though V_{REF} is labeled as a reference voltage, either input can be used as reference depending on the desired bias of the output. The 3k resistor in figure 3.7 acts as a pull-up resistor causing V_O to be approximately V_{CC}.

The pullup resistor is necessary because the output of the comparator is something known as "open collector". An open collector is an internal NPN transistor connected by its base to the output of the comparator, with the

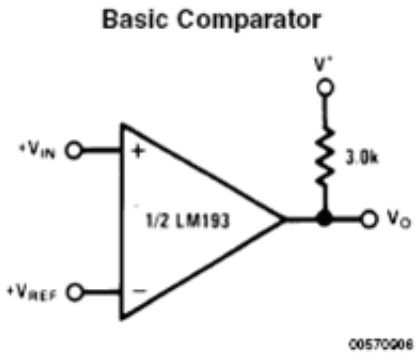
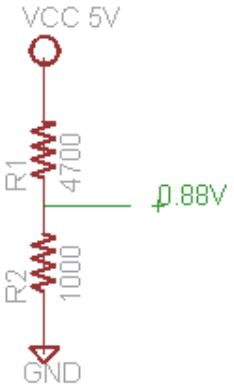


Figure 3.7: Comparator circuit design from LM193 datasheet.[20]

emitter connected to ground and the collector connected to the output pin of the IC. When the comparator sends a high signal to the transistor, current is allowed to flow from the collector (IC output) to the emitter (ground). This ties the output to ground and occurs in the case where V_{REF} is greater than VIN. When the comparator sends the transistor a low signal, the transistor essentially becomes an open circuit causing the IC output to float. The pull-up resistor then causes the output to be approximately V_{CC}.

Reference Voltages

The simplest method for obtaining the necessary reference voltages is to use a voltage divider. This consists of two or more resistors between VCC and ground, with the reference voltage being pulled from between two of the resistors. Figure 3.8 contains both a schematic of a voltage divider as well as the calculation for the output voltage.



$$V_{divider} = V_{CC}\frac{R_1}{R_1 + R_2}$$

$$0.88V = 5V\frac{1000\Omega}{1000\Omega + 4700\Omega}$$

Figure 3.8: Schematic and calculations for a voltage divider

While any of the three possible feedback signals are usable, the third option (measuring across the 150 resistor) provides for simple calculations and resistor requirements, so that is the chosen signal in this example. Having made that decision, the necessary reference voltages are some value between 0V and 1.73V, and between 1.73V and 2.88V. The second voltage is easy to obtain. A voltage divider between two equal valued resistors, such as 1k for both R1 and R2, between a V_{CC} of 5V and ground would result in a 2.5V reference voltage, which fits neatly in the required slot. For the lower reference voltage R1 again being 1k and R2 being 4.7k would provide an acceptable reference voltage of 0.88V.

Putting the parts together

Now that the general design and the reference voltages have been determined, its time to once again compare what the interface provides to what the microcontroller needs. At this point, using two comparator circuits, one for each mode of failure, the feedback interface provides two digital signals of undecided bias. As the design requirements do not necessitate the ability to discriminate between the two failure modes, ideally two signals would be reduced to a single signal binary signal indicating whether the LED works or is broken. Fortunately, due to the floating nature of the comparator's output, it is a simple matter to build a logical OR between the output of the two comparators (see figure 3.9), leaving only the matter of ensuring the correct biases on the comparators.

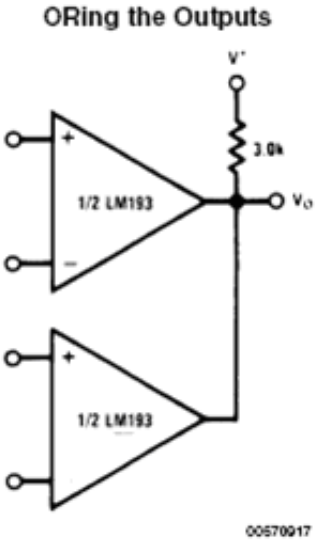


Figure 3.9: Logical OR of the output of 2 comparator circuits. From LM193 datasheet.[20]

While it is called a logical OR gate in the documentation for the comparators,

care must be used in utilizing the design. The way it actually works is that if either of the inverting inputs (the inputs marked by a - sign) are at a higher voltage than the non-inverting input (marked by a + sign), then V_O will be tied to ground. This means that to receive a logical high on V_O, both non-inverting inputs must be at a higher voltage than their corresponding inverting inputs. In this way it works more like an AND gate.

The final result

Figure 3.10 below contains the complete circuit diagram for the system. Starting with a digital low from the microcontroller to turn on the LED, passing through both the control and feedback interfaces, the microcontroller can read a digital input, seeing a digital low for a functional LED or a digital high for a broken LED.

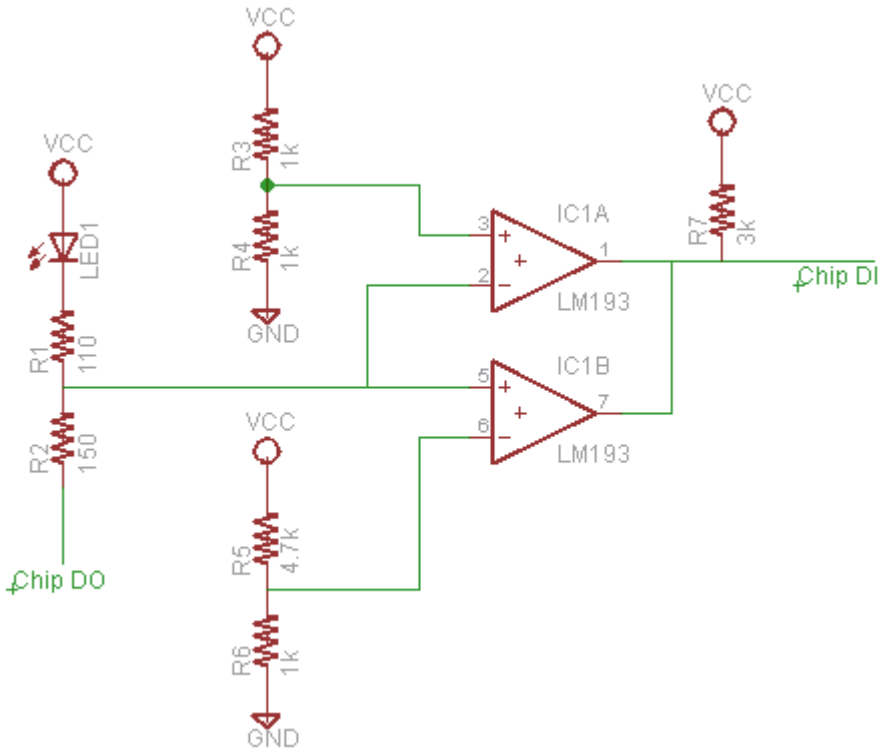


Figure 3.10: Circuit diagram of the entire LED control and feedback system.

Reading the result on the microcontroller

```
#include <avr/io.h>
```

```
#include <util/delay.h>

int main(void) {
  unsigned int LEDstatus;
  DDRA = 0xff;
  PORTA = 0xfe;
  _delay_us(1);
  LEDstatus = PINB & 0x01;
  return 0;
}
```

This is the same program that appeared in section 3.2 with four additional lines.

```
#include <util/delay.h>
```

This tells the compiler to include the delay library, which includes two functions causing the microcontroller to idle for a certain period of time.

```
unsigned int LEDstatus;
```

Here the program initializes a variable in which the program will store the information garnered from the feedback

```
_delay_us(1);
```

This is a call to a function in the delay library, and causes the microcontroller to idle for 1μs. While not necessary, it allows the system to fully stabilize before the chip checks the feedback.

```
LEDstatus = PINB & 0x01;
```

As previously mentioned, there are three registers for each of the 8-bit ports on the Atmega644p: PINx, PORTx, DDRx. Here the program is reading the inputs on port B. the rest of the line performs a bitwise AND operation on the read value and stores it in the previously initialized variable LEDstatus. A bitwise AND compares each bit in the two bytes (PINB and 0x01), and results in a new value that has a 1 in each bit in the byte where both the input bytes have 1s. For example, ANDing 0x56 and 0x24 would result in 0x04. `DDRB` was not set anywhere in the code as it defaults to all pins being input.

```
  0b01010110
& 0b00100100
  0b00000100
```

In the program this operation causes only the one bit of data corresponding to port B pin 0 (pin 1 on the ATmega644p) to be saved and ignores the other seven bits of the port.

3.4 DIO Operation

Being able to read the schematics of a given device or peripheral will aid in understanding how the device actually operates. As such, this section will take

the schematic for the digital I/O from the ATmega644p datasheet (figure 3.11) and explain the various components. This schematic covers the DIO for a single pin. Each pin has has these components, though some of the signals are shared between pins. Any other capabilities the pin may have (other peripherals) are not covered here. If the term "pin" or "port" is lower case such as these, they are referring to a single external connector on the microcontroller and its associated hardware or a collection of 8 of these pins respectively. If on the other hand these terms in all caps, especially if they are followed by "x" or "xn" they are referring to the registers which are components in each pin/port.

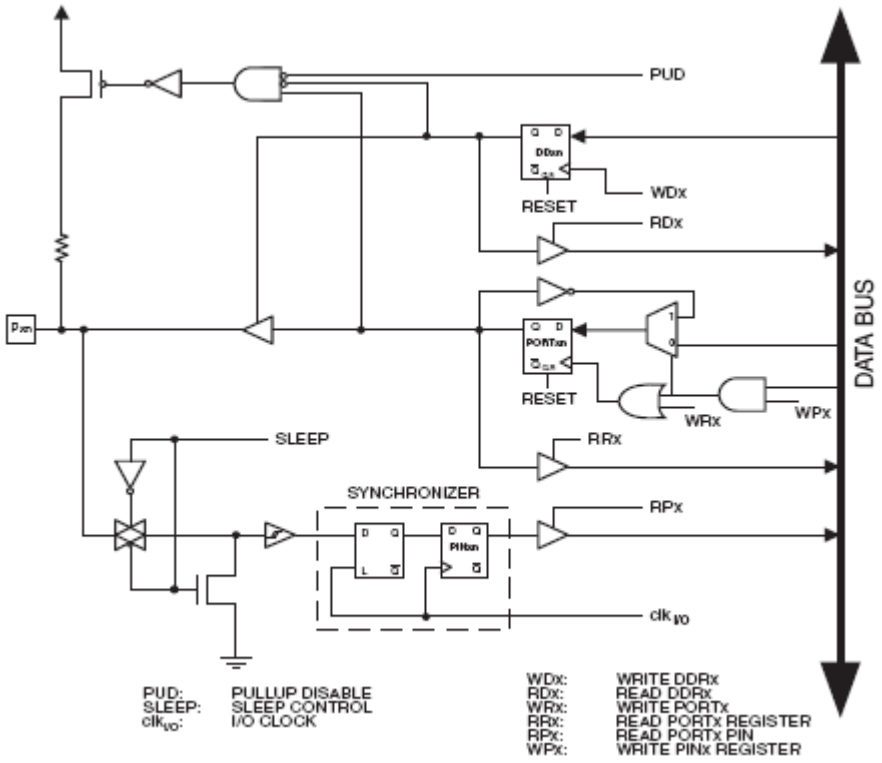


Figure 3.11: General Digital I/O schematic from ATmega644p datasheet.[11]

The first major component of this peripheral is the data bus on the right side of the schematic. This bus is what links all the peripherals and memory to the processor. Any read/write operation sends its data through this bus. The control signals which control the actual operation of the circuit are the labeled, unconnected lines. A legend for these signals is found at the bottom.

PORTx

The latch (the square component) in figure 3.12 is the component that is specifically PORTxn. Its output signal (Q) is what is connected to the pin when the pin is set as an output. The other components are involved in the logic of setting the value of this latch. The input to the latch marked < is the clock. On the rising edge of this signal, the latch samples the D input at stores that value. This can occur either when a 1 is being written to the corresponding bit in the PINx register, or when the PORTx register is being written to. In the former case the multiplexer (the trapezoidal component) changes to using its 1 input, which is the inverse of the latches output, thus causing the PORT's output to toggle. If, on the other hand, it is the PORTx write signal, the multiplexer uses its 0 input drawing the value from the data bus. The final signal in this segment is the PORTx read signal. This signal enables the buffer (triangular component at the bottom) and causes the latches output to be connected to the data bus. When not enabled a buffer has no effect on the line connected to its output. This is called a high-impedance or tri-state (see chapter 7). Table 3.2 is a truth table consisting of the various possible combinations of signals and conditions, as well as what results from them. The Q column is the new value for the latch and the PINxn and PORTxn columns refer to what is being written to a particular pin via the databus. A Q appearing within a column refers to the old value stored in the latch, while an x means that the value does not matter to the conditions in the remainder of the row. A line above Q means "not Q", which is the other logical value. If Q is high, then $\overline{Q}$ is low and vice-versa.

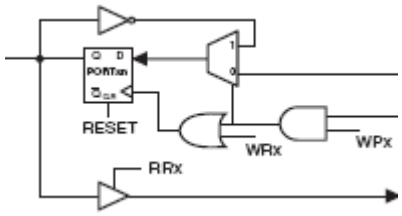


Figure 3.12: Segment of DIO schematic relating to PORTx register.[11]

WPx	WRx	PINxn	PORTxn	Q
0	0	x	x	Q
1	0	0	x	Q
1	0	1	x	$\overline{Q}$
0	1	x	0	0
0	1	x	1	1

Table 3.2: Truth table for the PORTxn latch.

DDRx

Figure 3.13 contains the portion of the DIO schematic relating to the DDRx register. Similarly to the PORTx portion, the actual value is contained in a latch which samples from the data bus when it receives the write signal on its clock input. Additionally it has them same type of buffer enabled by a read signal to return the currently stored value. The output of this latch is used as the enable for a buffer on the output of the PORTx latch. This buffer is what connects the latch to the actual output when the DDRx latch value is high.

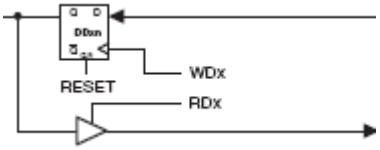


Figure 3.13: Segment of DIO schematic relating to DDRx register.[11]

PINx

The schematic for the PINx register is in figure 3.14. The segment to the left of the portion marked SYNCHRONIZER relates to power-saving (the SLEEP signal and signal conditioning. This ensures that the input to the SYNCHRONIZER will be an acceptable signal. The SYNCHRONIZER is a pair of latches which prevents pin value changes near clock pulses from causing glitches, but introduces a delay of $\frac{1}{2}$ to $1\frac{1}{2}$ clock cycles. The read signal enabled buffer is the same as was on the previous two sections.

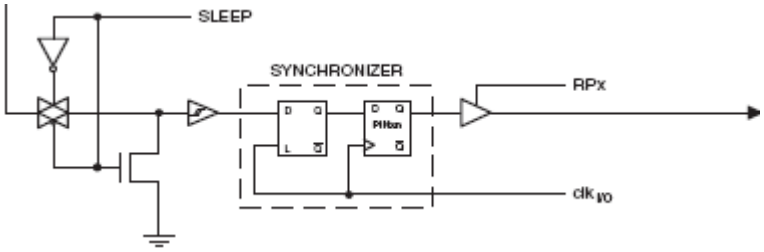


Figure 3.14: Segment of DIO schematic relating to PINx register.[11]

Pull-up resistor

The final portion of the schematic relates to the pull-up resistor as seen in figure 3.15. If enabled, the pull-up resistor connects the pin to V_{CC} through a large resistance. This has the effect of causing a floating value at the pin to be pulled up to V_{CC} rather than being indeterminant. A floating value at a pin is caused

by the pin not being connected, either directly or indirectly, to a power source or ground. The actual voltage potential on a floating pin cannot be predicted, and it can change over time based on background electrical noise. The switch enables the resistor unless one or more of the following statements is true: the `PUD` (pull-up disable) bit in the `MCUCR` register is set, the `DDRxn` bit for the pin is set high causing the pin to be an output or if the `PORTxn` bit for the pin is low. If one or more of those conditions is true, the switch is opened and the pull-up resistor is disabled.

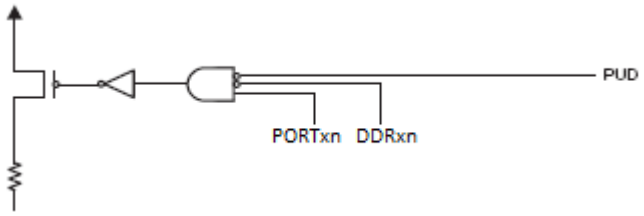


Figure 3.15: Segment of DIO schematic relating to pull-up resistor.

Reading the registers

It can be incredibly frustrating, and sometimes difficult to debug, if a port is not behaving as one believes it should. This section will attempt to reinforce how the registers interact by giving several cases.

- If `DDRxn` is set to output, reading `PINxn` and `PORTxn` will both result in the value stored in `PORTxn`.
- If `DDRxn` is set to input, `PINxn` and `PORTxn` are disconnected from each other by the disabled tri-state buffer. Reading `PORTxn` will read result in the value stored in the register, while `PINxn` will read the value that any external device is setting the pin to. If left floating and the pull-up resistor is enabled, this will read as high.
- Writing to `PINxn` will always toggle the value stored and read from `PORTxn`, whether the pin be set to input or output.
- Writing to `PINxn` does not directly affect the `PINx` register. If reading the `PINx` register after writing to it, do not expect to see the value just written.
- After writing to `PORTx`, if reading `PINx` does not give the same value, it is likely that the data direction is set to input and not output.

To further demonstrate this point figure 3.16 displays the path the `PORTxn` signal travels to reach the output pin as well as back to the data buffer along both the `PINxn` and `PORTxn` read paths when the pin is set as an output, while

figure 3.17 shows the separation between `PORTxn` and `PINxn` caused by setting the pin as an input. In both figures a heavy solid line indicates the path of `PORTxn` while a dashed line is the signal generated by an external device.

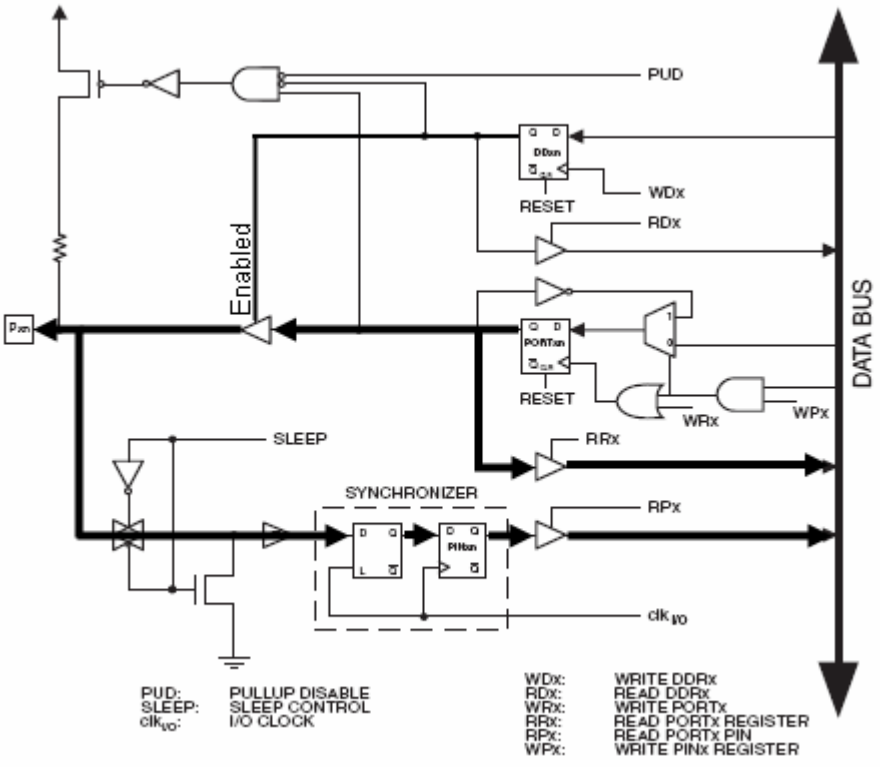


Figure 3.16: Path of `PORTxn` signal when a pin is set as an output.

3.5 Conclusion

Digital I/O is the most basic method a microcontroller has to interact with other devices. Though simple, it forms the back bone for most everything else and teaches a great deal about actually operating a microcontroller. Just like beginning to learn a programming language by writing a "Hello World" program, digital I/O is the first step for any new microcontroller.

If the schematics shown in this chapter don't make sense, or more information about the components is desired, there are numerous introductory or hobbyist level books available on electronics. Check the local library, as they will likely have a variety of them.

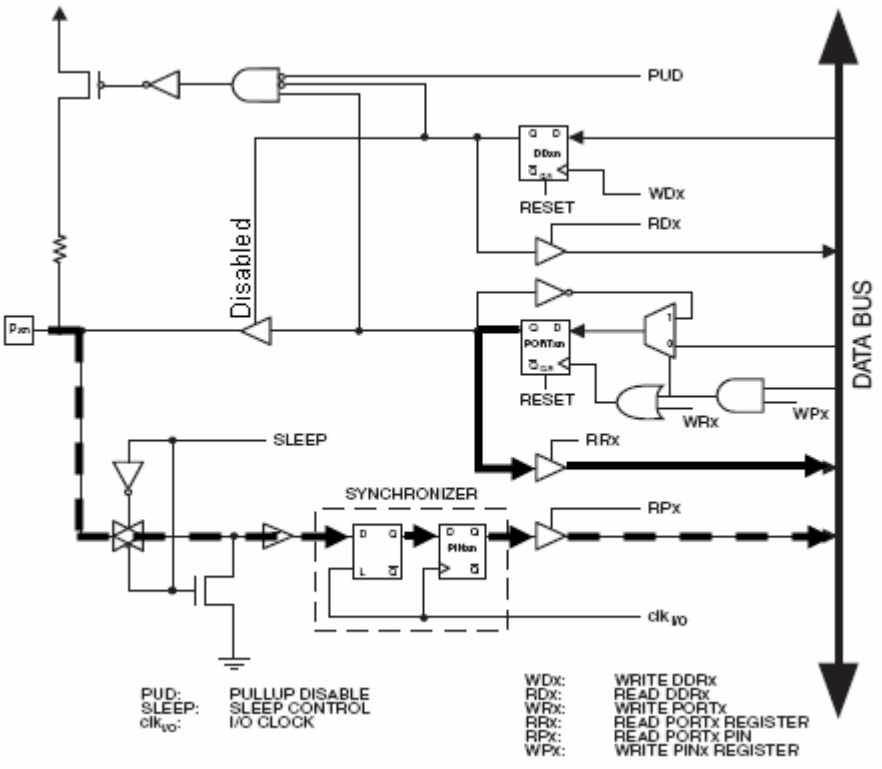


Figure 3.17: Path of `PORTxn` (heavy, solid line) and input (heavy, dashed line) signals when a pin is set as an input.

Chapter 4

Analog Digital Converter

The feedback interface designed previously for the LED example is all well and good; however it is relatively complicated, especially for such a simple task and basic control interface. It would be helpful if the microcontroller itself could accept any voltage and then would calculate the logic itself. This has the benefit of being a much simpler system and a more flexible system as the logic could be changed in software rather than needing to redesign the feedback interface.

Thankfully, most microcontrollers have a peripheral called an Analog to Digital Converter (ADC). ADCs accept an analog signal and using one of a variety of methods provide a digital representation of that signal for use in a program. Before returning to the LED example and using an ADC, there is some information that should be understood.

4.1 Terminology

There is a great deal of terminology integral to the use of ADCs. Here are the definitions of many of these terms, followed by an example of their use and values:

- ADC (Analog Digital Converter): A device found often in microcontrollers or as a dedicated chip which converts an analog voltage into a representative digital value.
- ADC Resolution: Generally measured in bits, the ADC resolution indicates the number of discrete digital values the ADC can output. For example an 8-bit ADC can produce 256 (2^8) .
- Accuracy: There are several sources of error inherent in the use of ADCs. Accuracy is an overall measure of these errors and is measured in a unit called an LSB (see Least Significant Bit).
- Least Significant Bit (LSB): This is a unit for measuring, among other things, the error. It is equal to the change in input signal voltage necessary

to increase the digital output signal to the next consecutive value (i.e. the difference between receiving an output of 114 vs 115).

- Full scale: The full scale measurement of an ADC is the range that covers the lowest measurable value to the highest measurable value. These are often based on ground or V_{CC}, but can also use separate reference voltages.
- Voltage Resolution: This is the change in input voltage corresponding to a single step change in the output signal. It is also equal to 1 LSB.
- Conversion time: Each analog to digital conversion takes a finite amount of time. The conversion time is the longest time a conversion can take.

Example:

A 10-bit resolution ADC is attempting to measure a 0-5V analog signal. In this case the full scale measurement is 0-5V and the resolution is 10-bit. The voltage resolution is equal to the full scale measurement range divided by the number of discrete intervals.

$$Q = \frac{E_{FSR}}{2^n - 1} = \frac{5V - 0V}{2^{10} - 1} = \frac{5V}{1023} = 0.0049V/step$$

where Q is equal to both the voltage resolution and 1 LSB.

4.2 Converting methods

There are several methods for converting an analog signal to a digital value. This section will cover some of them [15].

- Direct Comparison: This method uses a large bank of comparators, each one corresponding to a specific and distinct value. This method is very fast as the conversion time is just the settling time of the comparators plus the time required to sample them, however is generally limited in precision as for each additional bit the number of required comparators is doubles. This also leads to a large die size and high power dissipation.
- Successive-approximation: This is the method that is employed on the ATmega644p. It uses an internal digital-analog converter (DAC) which provides reference voltages to a single comparator. A binary search algorithm is then used to successively narrow down the possible values. This leads to a somewhat longer conversion time, requiring a comparison for each bit of precision, but requires far less space and power than a direct comparison ADC.

Example:

A 3-bit ADC with a range of 0-10V is measuring an 7V signal.

Clock cycle 1 - The DAC produces a 5V (0b100) signal for the comparator, which reports that the sampled signal is higher. A binary 1 is stored in the first bit of a successive approximation register (SAR)

Clock cycle 2 - The DAC produces a 7.5V (0b110) signal. This time the comparator reports that the DAC's output is higher, so a 0 is stored in the SAR (which now reads 0b100).

Clock cycle 3 - The DAC produces a 6.25V (0b101) signal. Again the sampled signal is higher so a 1 is stored in the SAR (0b101).

The end result is that the ADC reports that the input signal is 6.25V. Due to the low resolution this is the best it can do. It is important to note that in this example the reported voltage is strictly lower than the actual voltage by as much as 1 LSB. This error is called quantization error and will be covered more fully later. It is possible to shift the interpreted value in software by 0.5 LSB which will shift the error from +1/-0 LSB to ±0.5 LSB. If this is implemented the above example would still report the same digital signal, but this would now be interpreted as 6.875V, which is closer to the actual value. This method of shifting the interpreted value will not always reduce the error, but will reduce the standard deviation of the error.

- Ramp-compare: Similar to the successive-approximation method, the ramp-compare ADC utilizes a single comparator, but with a difference in the input method. Instead of using several compare and adjust cycles, a slowly increasing voltage is applied to the reference voltage of the comparator and a clock is used to determine when the reference voltage increases above the sampled voltage. There are two methods to produce this ramped voltage. Either a DAC using a clock to constantly increase its output can be used, or a calibrated timed ramp. A simple, though non-linear, possibility for the calibrated ramp uses a single resistor and capacitor. One benefit of this ADC design is that additional comparators require only an additional comparator and storage register. The clock and ramp generation can be shared between all the comparators in use.

- Integrating: This method applies the unknown sampled voltage to the input of an integrator circuit. The integrator is allowed to ramp up for a known duration (the run-up period), at which point the input of the integrator is switched to a known voltage of opposite polarity. The time it takes for the output of the integrator to reach zero (the run-down period) corresponds to the sampled voltage.

4.3 Sources of Error

This section will address and quantize some of the sources of error inherent in ADCs. The error values are given in units of LSB as discussed previously.

- Quantization: Mentioned earlier, quantization error is the error inherent in any attempt to represent an analog value as digital. Depending on the type of ADC in question, this error will either be +1/-0 LSB or ±0.5 LSB. In the former case it is adjustable to ±0.5 LSB simply by adding 0.5 LSB to the result.

- Aliasing: ADCs are often used to regularly sample a changing signal to determine the signals characteristics. Because an ADC works with discrete points in time, it is working with an incomplete picture of what's going on. Aliasing is an error caused by attempting to sample a signal at a frequency over half of the sampling frequency, and will cause the signal to appear to be a different frequency than it actually is. This maximum observable frequency of half the sampling frequency is called the Nyquist rate. When the sampled signal is at a frequency greater than the Nyquist rate it will appear to be at a frequency mirrored across the Nyquist rate.

  $$f_{obs} = f_{Nyquist} - (f_{actual} - fNyquist)$$

  Take the example of a 10Hz ADC was attempting to observe a 7.5 Hz signal. Here the Nyquist rate is 5 Hz, so the observed signal will be at 2.5 Hz.

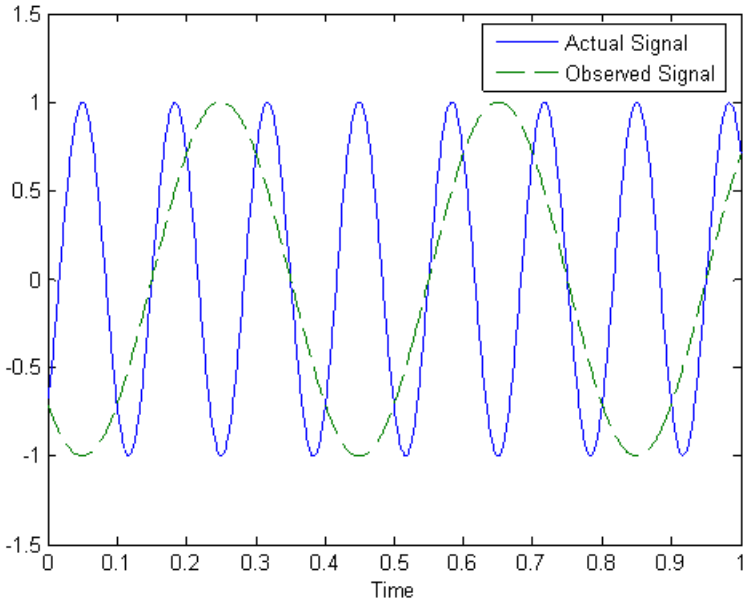


Figure 4.1: Aliasing caused by sampling a 7.5Hz signal which is above the Nyquist rate of 5Hz. Sampling rate is 10Hz.

- Aperture: No clock is perfectly accurate. There is always some jitter, which is the clock's cycle being slightly longer or shorter than the adjacent cycles. Overall this doesn't change the clock frequency, but can cause individual cycles to happen at a time other than what is expected. When the sampled signal is a non-constant value, this slight offset in sampling time will cause the wrong value to be sampled.

4.4 Return to the LED example

With the inclusion of the ADC, the feedback interface for the LED example becomes far simpler. No longer needed are the comparators, reference voltage dividers or pull-up resister. In fact, the only required feedback interface is a wire running from either side of the 110Ω resistor to the ADC on the microcontroller. The changes in the voltage reading will be larger if the connection between the LED and the 110Ω resistor is sampled, so that is where the ADC should be connected.

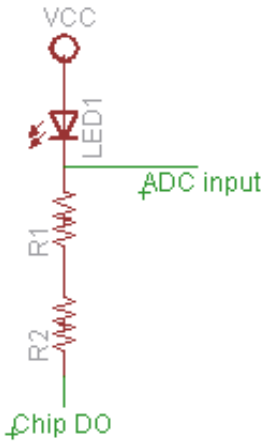


Figure 4.2: Complete circuit for LED example using the ADC

4.4.1 The ATmega644p ADC

The ADC on the ATmega644p is a 10-bit successive approximation ADC. The ADC is connected to the pins of Port A via an 8-channel analog multiplexer. This allows any and all pins on Port A to be used as an analog input, however as the ADC has only a single converter, only one of these channels can be read at any given time.

The ADC contains a sample-and-hold circuit, which is used to prevent the sampled value from changing during a conversion. This is important when measuring a non-constant signal, especially when the signal is changing rapidly in comparison to the conversion time.

There are several choices for the reference voltage on the ATmega644p. The reference voltage can be selected between AVCC, which is a separate supply voltage pin, but which also must be within ±0.3V of V_{CC}, internal 1.1V or 2.56V reference voltages, or an external `AREF` (analog reference) pin. The selection of the reference voltage is stored in the SFR `ADMUX` (ADC Muliplexer Selection).

The ADC also supports a variety of differential voltage measurements, some of which with programmable amplification of 1x, 10x or 200x. Using these amplifications is limited to only two pairs of the differential inputs (`ADC1,0` and `ADC3,2`) and also reduces the resolution. More information on the differential voltage capabilities of the ATmega644p can be found in its data sheet with the other ADC information in chapter 20.

4.4.2 Understanding the ADC

The simplicity of the physical circuit has its cost in code complexity. Using this peripheral requires initialization and configuration. On the ATmega644p, there are 5 registers involved in the use of the ADC. Here is a brief description of each register. Further, more detailed information can be found in section 20.9 Register Description of the data sheet [11].

- ADMUX - ADC Multiplexer Selection Register
  Within the ADC are several multiplexers which are controlled by the bit fields in this register. These include the selection of the reference voltage (REFS1:0) as well as the various channel and gain selections (MUX4:0). This register also contains the justification of the result within the ADCW register (ADLAR).

- ADCSRA - ADC Control and Status Register A
  This is the first of two registers responsible for controlling the ADC and making available to the programmer various bits of status information. This includes, but is not limited to, enabling the ADC, triggering conversions and controlling interrupts. Additionally, this register controls the selection of a clock prescaler for the ADC. This prescaler causes the ADC clock to run several times slower than the system clock, and is necessary because in order to obtain the full 10-bit precision the ADC must run at a clock frequency between 50kHz and 200kHz.

- ADCW - The ADC Data Register
  This register is a 16-bit register made by the combination of ADCL and ADCH. While the datasheet only lists the component registers, ADCW is defined in the header files for the ATmega644p, and is very useful in that it precludes the necessity of manually joining the registers to obtain the full value. Additionally, it is required that when using ADCL and ADCH that ADCL be read first. Reading ADCW takes care of this requirement as well.

- ADCSRB - ADC Control and Status Register B
  At the time of writing, there is only a single 3 bit field in this second control

register. This bit field controls the source for auto triggered comparisons. Auto triggering allows for performing analog conversions at times specified by certain events rather than at specific points in the code. These events could be based on time or an external event. It is also possible to have the ADC begin conversions as soon as the previous one finishes.

- DIDR0 - Digital Input Disable Register 0
  The use of this register is completely optional and is used to reduce power consumption. Each bit of this register corresponds to one of the eight ADC input pins. When a given bit is set, the corresponding pin's digital input capabilities are disabled and the PINA register will always read a 0 for that bit.

Each of these registers corresponds to a physical device inside the ADC. Four of them (`DIDR0` is left out as it appears in the DIO logic) appear across the top of figure 4.3, which is a schematic of the internal workings of the ADC. The values of each bit in the registers are transmitted in parallel to the other devices in the ADC. These are the lines exiting and entering the bottom of the register boxes in the schematic. These signals control the other devices, causing the ADC to behave as desired. `ADCSRB` only controls auto triggering, so its logic is included in figure 20-2 of the data sheet, rather than this one.

There are two sections of this schematic that deserve additional attention. These are the reference selection and the signal selection.

Reference selection

When configuring a progressive-approximation ADC such as the one in the ATmega644p, the ADC must have some reference value to compare to. This voltage is supplied by the portion of the ADC found in figure 4.4. There are four options for the voltage source for the comparator which are selected between using the `REFS1:0` bits of the `ADMUX` register. These options are `AVCC`, which is an external pin that should be connected to the voltage that the microcontroller is running at, `AREF`, which is an externally supplied voltage, and the two internally supplied voltages of 1.1V and 2.56V. The first three of these options are selected through a multiplexer. The final choice on whether it is the output of the multiplexer or the voltage supplied at `AREF` is used is determined by the switch. If the switch is engaged the multiplexer output is connected to the `AREF` pin and supplied the digital-analog converted (DAC), otherwise only `AREF` is connected to the DAC. If one of the three sources chosen by the multiplexer is used, it is best to connect `AREF` to ground using a capacitor to reduce electrical noise, which may result in inconsistencies.

The selected voltage is used to power a DAC. This DAC can then produce voltages in the range of 0V to the selected reference in 2^{10} increments. The output of the DAC is used in the comparator to determine the analog voltage that the ADC is working on.

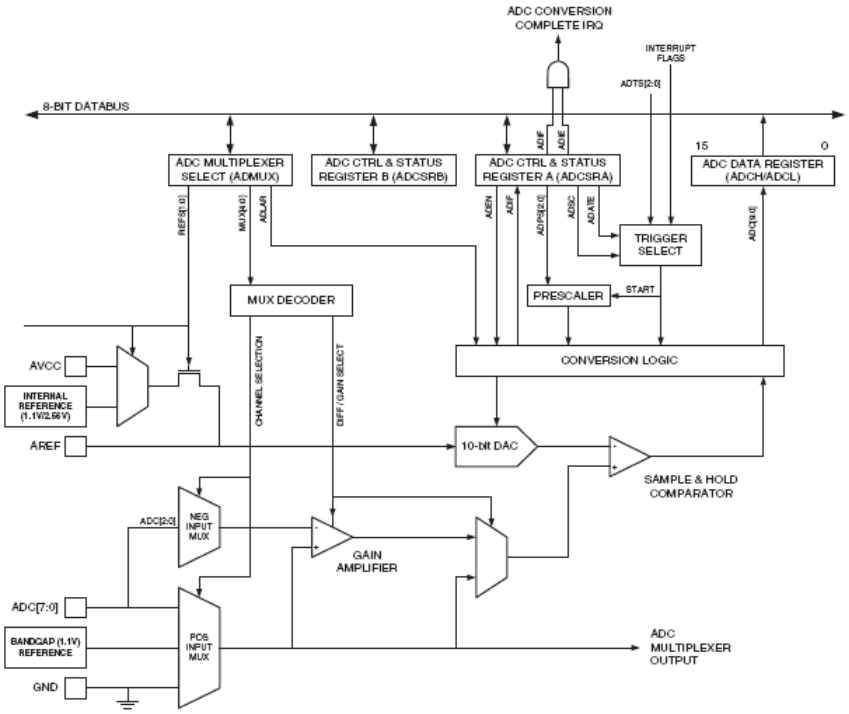


Figure 4.3: Schematic of the ATmega644p ADC. Found in section 20.2 of the datasheet.[11]

Signal selection

The other section of the ADC that deserves attention is the signal selection. This portion determines what signal is actually converted into a digital value. This value can either stem from one of the eight external ADC pins, which are the pins of port A, ground, or an internally created 1.1V reference. The ground and 1.1V reference voltages are generally used to calibrate the ADC and ensure that it gives the expected value. All these options are fed into the positive input multiplexer, and one of them is selected depending on the status of the `MUX4:0` bits in the `ADMUX` register. This voltage is used both as the positive input to the gain amplifier, and as one of the inputs to another multiplexer discussed below.

The first three external ADC pins (`ADC2:0`) are also connected to another multiplexer. This is the negative input multiplexer which allows for differential conversions. In this case the signal selected by the negative input multiplexer is subtracted from the signal selected by the positive input multiplexer by the gain amplifier. This amplifier can optionally multiply this difference by 10 or by 200. The result of this subtraction and optional multiplication is then used as the second input to the afore mentioned final multiplexer.

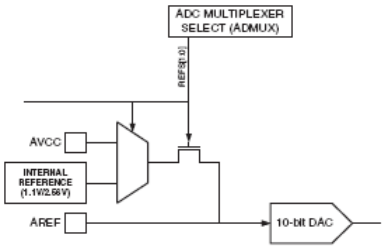


Figure 4.4: Reference voltage selection schematic. Excerpt from figure 4.3.

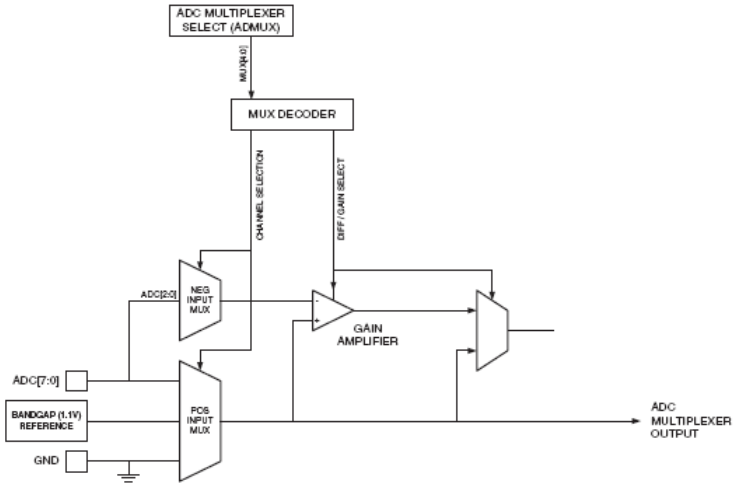


Figure 4.5: Signal selection schematic. Excerpt from figure 4.3.

The unlabeled multiplexer on the right side of figure 4.5 determines whether the signal directly from the positive input multiplexer is used (called a single-ended input) or if the result of the subtraction and multiplication is used (called a differential input). The result of this selection is what is passed on to the progressive-approximation circuitry. For additional information regarding the values stored in `ADMUX` for each of these options, as well as the limits and capabilities of the peripheral refer to the ADC portion of the microcontroller's datasheet.

4.4.3 Programming with the ADC

With the information on the various ADC registers now in hand, it is possible to utilize the ADC for checking the status of the LED. As before, the entire program will be presented first followed by a breakdown of its various components. Since it is much longer and more complicated, this will not include a line by line breakdown, rather certain aspects will be emphasized and other parts merely

summarized.

```
#include <avr/io.h>
#include <util/delay.h>

int main(void) {
  unsigned int LEDstatus;

  ADMUX = 0x00 | (0b01 << REFS0) | (0b0 << ADLAR);
  ADCSRA = 0x00 | (1 << ADEN) | (0b110 << ADPS0);
  ADCSRB = 0x00;
  ADCSRA |= (1 << ADSC);        //begin first conversion.
                          //Done for timing purposes
  while (ADCSRA & ~(1 <<ADIF)) {}
          //wait until conversion complete
  DDRA = 0x01;

  while(1) {
    PORTA = ~PORTA;
    _delay_ms(1);

    while (ADCSRA & (1 << ADSC)) {}
    //wait for any previous conversion to complete
    ADMUX = (ADMUX & 0xe0) | 1; //set ADC channel to 1
    ADCSRA |= (1 << ADSC);      //begin conversion
    while (ADCSRA & ~(1 <<ADIF)) {}
            //wait until conversion complete
    LEDstatus = ADCW;

    if (~PORTA && ((LEDstatus>820) || (LEDstatus<410))) {
      // This will be reached only if the LED is bad
      break;
    }
    _delay_ms(998);
  }
  return 0;
}
```

Before examining the code in great depth, a few of Atmel's standards for accessing the SFRs must be understood. This information can be determined through examination of the header files, especially, iomxx4.h. This header file contains a list of several hundred #define statements used to access the various registers and bit fields. Each register name from the data sheet is defined to the specific memory address of the SFR and can therefore be used directly in the code as was done previously with the PINx, PORTx and DDRx registers. Additionally, for each register there are #define statements for each bit. These bit field definitions are not directly usable however. Rather than directly accessing the bit in question, the bit field names are defined as the bit number in the register. It is therefore necessary to left shift the desired value of a given bit by the bit field name to position the desired value correctly. For example, in

order to start a conversion a value of 1 must be written to the ADSC bit of the ADCSRA register. In order to do that the 1 must be left shifted (<< operator) ADSC times

```
ADCSRA |= (1 << ADSC);
```

where ADSC is defined as

```
#define ADSC  6
```

The |= operator is 'or equals'. This tells the compiler to store in the variable to the operator's left the results of a bitwise OR operation between the left and right sides of the operator. It is necessary to use the OR operation to maintain the other bits as they were prior to this instruction because the entire register must be written at once. It is not possible to actually write a single bit of a register without writing the entire register. This is not the only method for performing this sort of shifting operation, but it is the one that will be used throughout this text.

Initializing the ADC

There are several settings required to properly run the ADC.

```
ADMUX = 0x00 | (0b01 << REFS0) | (0b0 << ADLAR);
ADCSRA = 0x00 | (1 << ADEN) | (0b110 << ADPS0);
ADCSRB = 0x00;
ADCSRA |= (1 << ADSC);         //begin first conversion.
                          //Done for timing purposes
while (ADCSRA & ~(1 <<ADIF)) {}
          //wait until conversion complete
```

This section of code is responsible for initializing and setting up the ADC for the use it will be put in this example.

```
ADMUX = 0x00 | (0b01 << REFS0) | (0b0 << ADLAR);
```

This line sets the ADMUX register to specify that the reference voltage for the conversions will be the AVCC voltage and that the result will be right justified. Technically it also specifies that the ADC will perform single-ended (non-differential) conversions on channel 0, however that does not matter at this point. This line could be simplified to

```
ADMUX = 1 << REFS0;
```

However the extra code makes it easier to change specific settings in the future, as well as make it more clear what settings are being used to someone reading the code. There is no loss of efficiency either as the extra operations are optimized out by the compiler.

```
ADCSRA = 0x00 | (1 << ADEN) | (0b110 << ADPS0);
```

These settings for the ADCSRA register specify that the ADC is to be enabled (ADEN) and that the prescaler is to be 64 (ADPS). This required prescaler was calculated by dividing the clock speed of the microcontroller (8MHz) by the maximum speed of the ADC clock to ensure full precision (200kHz). This resulted in a required prescaler of at least 40. The smallest possible prescaler above that value was 64 resulting in an ADC clock speed of 125kHz.

The final lines of the section should, at this point, be readily comprehensible. They, in order, clear all the bits of the ADCSRB register as auto triggering is not being used, begin an ADC conversion, and wait for the ADC Interrupt Flag to be set, signaling the completion of the conversion. While the results of this first conversion do not matter, the fact that it is performed at this point rather than later is important for timing. For this ADC, all conversions but the very first require 13 ADC clock cycles (104μs with current settings). Due to the necessity of initializing the analog circuitry, the first conversion after enabling the ADC requires 25 ADC clock cycles (200μs). This difference in time can cause issues in applications where precise timing is required from the beginning, so this longer conversion can be performed during initialization when timing tends to be less important.

Reading the ADC

At this point it should be a fairly simple task of determining the individual tasks of each line of code based on the initialization code, so only a brief overview should be required. This section takes as an input the channel to perform a conversion on. It then ensures that any previously requested conversions are completed, and waits for them to finish if necessary. It then sets the channel selection multiplexer to the desired channel and begins a conversion. After the conversion is completed, it saves the results of the conversion to `LEDstatus`.

```
while (ADCSRA & (1 << ADSC)) {}
//wait for any previous conversion to complete
ADMUX = (ADMUX & 0xe0) | 1;    //set ADC channel to 1
ADCSRA |= (1 << ADSC);         //begin conversion
while (ADCSRA & ~(1 <<ADIF)) {}
            //wait until conversion complete
LEDstatus = ADCW;
```

Putting it together

After initializing required variables, setting the DDRA register to have pin 0 be a digital output and initializing the ADC, the main loop of the program can begin.

```
while(1) {
  PORTA = ~PORTA;
  _delay_us(1);

  while (ADCSRA & (1 << ADSC)) {}
```

```
    //wait for any previous conversion to complete
    ADMUX = (ADMUX & 0xe0) | 1; //set ADC channel to 1
    ADCSRA |= (1 << ADSC);      //begin conversion
    while (ADCSRA & ~(1 <<ADIF)) {}
            //wait until conversion complete
    LEDstatus = ADCW;

    if (~PORTA && ((LEDstatus>820) || (LEDstatus<410))) {
      // This will be reached only if the LED is bad
      break;
    }
    _delay_ms(999);
  }
```

This loop does several tasks, in order, ad infinitum unless a problem is detected. The first thing it does is change the entire Port A from outputting a high signal to low, or from low to high. The first execution is a low to high transition. While only the first pin of the port is actually required, the entire port can be changed with no adverse effects because all the other pins are set to input in the data direction register.

After a one microsecond delay to ensure stabilization of the output, the ADC is read and the result stored. The logic in the if statement checks for a failure in the LED. If the output pin is low (which turns on the LED) and the voltage is above 4V (a reading of 820) or below 2V (a reading of 410) the LED must be bad, as the voltage reading of a correctly operating circuit would read at 3V (an ADC reading of 615). In this case the program leaves the loop via the break command and the program exits. If the LED is functioning properly the program waits the remainder of a second before repeating the loop.

It should be noted here that the entire loop is not exactly one second. It is in fact slightly longer than one second due to the time involved in performing the analog conversion, the logic test to determine the state of the LED as well as the bookkeeping for the loop itself. This will be covered in greater detail later, it is sufficient to say for now that the loop is less than half a millisecond longer than desired.

4.5 Conclusion

ADCs are very commonly used as many sensors provide analog signals. It is also a device that can cause problems that may be difficult to notice if incorrectly set. As such, always ensure the reference voltage is stable and correct, and that the ADC clock is an appropriate frequency. There is a great deal more information, and numerous more methods than what is presented in this chapter. A good place to start for further information would be *The Data Conversion Handbook* available at Analog Device's website [15], and certainly the ATmega644p datasheet's chapter on the ADC (chapter 20) for more information on the ADC on this microcontroller [11].

Chapter 5

Code Styles

As can be seen even within the short programs presented so far, without good organization code can quickly become an unmanageable mass, especially for people other than the author attempting to discern how it works. This chapter will break away from specifics on using any particular hardware and delve into some stylistic and organizational aspects of coding.

5.1 Headers and Header Files

This section of the code consists of instructions to the compiler as well as variable and function declarations. These instructions are necessary to tell the compiler what to do with certain things found throughout the code, and are not ultimately executed in the compiled code. Often times these instructions are contained within a header file. These header files are used to separate the compiler instructions from the executed code for organization as well as for improving the possibility for code re-use. When written to correspond to a single, specific file containing code, the header file will often use the same name as the code file, only using a .h (for header) extension in place of the .c extension (e.g. my_program.h for my_program.c).

5.1.1 `#ifndef MY_PROGRAM_H`

All header files should start with a `#ifndef` statement. This is a pre-processor instruction (as signified by the `#` symbol) which tells the pre-processor to check to see if `MY_PROGRAM_H` has been defined previously. If it has, then the compiler skips to the corresponding `#endif` statement, which should be at the bottom of the header file. The line after the `#ifndef` should be

```
#define MY_PROGRAM_H}
```

which will create the definition of `MY_PROGRAM_H` and cause the compiler to skip the interior of the `#ifndef #endif` section should the compiler attempt to compile the file a second time within the same compilation of the program, thus

avoiding illegal multiple declarations of anything contained within. Should this combination of lines not be included and the file appears in multiple `#include` lines in other files, then the compiler may throw errors about multiple definitions of the same function or variable. The standards for the definition is to use the file name in all capital letters, with the period for the extension turned into an underscore.

5.1.2 Additional Inclusions

In order to better organize the header files, similar commands are grouped together. The first of these commands is `#include`. This command tells the compiler to look at another file and compile it as well, and is used to bring code in from other files such as libraries (see the section on Libraries later in this chapter). Even the header files must be included by the source files in this manor. As long as all the header files are properly constructed with the `#ifndef` commands as covered above, there is little need to be concerned over including the same file multiple times in a single project. There are two slightly different types of `#include` statements depending on where the library is located. If the library is located in a folder in specific paths known by the compiler, such as where the built in standard libraries were installed to, then angle brackets are used around the library name e.g.

```
#include <math.h>
```

whereas if the file is located in the directory where the current source file is contained, then double-quotes replace the angle brackets. The standard relative path syntax is used to navigate between folders and can be used to access higher level folders than the current directory. The command

```
#include "..\Libraries\myFile.h"
```

would look for `myFile.h` inside the `Libraries` folder which resides in the same parent folder as the current directory (see figure 5.1 for the file structure in question).

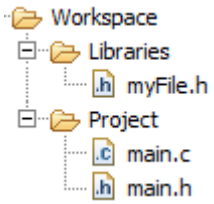


Figure 5.1: File structure for relative path include.

5.1.3 Macro Definitions

Compiler macros are created using the #define statement using the structure

`#define MACRO_NAME macro`

It is standard to use all capital letters for macro names. The compilers then use macros by directly replacing any occurrence of the macro name in the code with the macro itself via direct substitution. This is most commonly used for removing numbers from calculations in the code whose purpose may not be obvious, and replacing them with names. This can improve the readability of the code, and also makes it simpler to change the value at a later date. This is especially true if the number appears in multiple locations.

For example, there are two values that are found by experimental means that are needed as a multiplier and an offset in a control function. The header file could then contain the lines

```
#define CONTROL_MULT 0.23897
#define CONTROL_OFFSET 2.348
```

and the values would be used in the code in the line

```
output = (input * CONTROL_MULT) + CONTROL_OFFSET;
```

which is more readable and, when new data is available and the multiplier and offset change, the macros will be easier to change than finding that line again.

While this is one common use, these macros can also be used to replace calculations, function calls and even entire blocks of code. Here's an example used for calculating the value to store in a specific register on the ATmega644p which determines the baud rate for serial communications (see the chapter on the USART peripheral).

```
#define UBRR_VAL F_CPU/16/BAUD-1
UBRR0 = UBRR\_VAL;
```

This calculates the required value for the `UBRR` register based on the clock speed (`F_CPU`) and the desired baud rate.

Important note: macros should NOT end in a semi-colon unless one is desired where the macro is used. This example

```
#define DEF\_NUM 4;
sum = DEF\_NUM + 2;
```

throws an error of "`expected identifier or '(' before '+' token`" as the macro will result in

```
sum = 4; + 2;
```

and the compiler will interpret `+ 2;` as its own command rather than as part of the sum as intended.

5.1.4 Global Variable Declaration

The next section of statements in a header file are the global variable declarations. These global variables are just like local c variables, except that multiple functions can access them through the use of the `extern` keyword. It is also important to note that initializing the global variables in the header file can cause errors that may not be easily traceable back to the actual source (the initialization).

5.1.5 Function Prototypes

Finally there are the function prototypes. These are promises that the functions exist and can be called, but are implemented in some source file not at the location where the prototype appears. The prototypes are formatted the same way as the standard function declaration, however instead of there being code contained within {}'s, there is only a semicolon.

```
int foo(char bar);
```

The purpose of these prototypes is to increase the scope of the functions and allow other source files to call them. Additionally it allows the implementations to occur in any order in the source file. Take, for example, a program with a main function which calls the function `foo` and no function prototypes. If the `main` function were implemented before `foo` (earlier in the file), then the error `'foo' undeclared (first use in this function)` would be thrown. The error would not be thrown if `foo` was implemented first, or if function prototypes were included.

5.2 Commenting

Comments are a vital part of a program, not for changing how the code is run on a machine, but for maintenance and use by other people. Some people propose that every line that contains code should be commented, others are proponents of fewer comments, but still more lines with comments than lines without. The number and style of comments needed will vary depending on the desires of the person in charge of the project (often times a manager; the boss of the person writing the code) and the complexity of the code. Well written code will require fewer comments as the code is more self explanatory, however a good rule of thumb is that a reader should be able to understand what the code is doing, as well as how it is doing it by reading the comments alone.

5.2.1 Documentation Comments

Documentation comments are blocks of comments whose purpose is to create the documentation for the program. When properly formated programs such as Doxygen will automatically pull these comments out and form them into a nice set of documentation.

Doxygen standards are very similar to Javadoc, both in methods to declare a comment block as documentation and in the tag system used to mark certain pieces of information. Many of the pieces of information important to include in the documentation have keywords called tags that precede the information in the comment block. These tags include `brief`, `return`, `param` and `file`, and are preceded by either a \or an @ symbol. Either will work, but it is best to choose one of the two and stick with it within any program for consistency's sake.

There are several variations on how to mark comments as documentation, using both the comment block and comment line styles. The most common method for comment blocks, because it is also the method used by Javadoc, is to begin the comment block with a second asterisk. Often times each line will also begin with an asterisk to improve visibility in the code.

```
/**
 * This is a documentation comment
 */

/*
  This is not
*/
```

Another method, utilizing comment lines, is to create a block of comments at least two lines long, and beginning with an extra /.

```
 ///
 /// This is a documentation comment
 ///
```

File Documentation Comments

Each file should begin with a documentation comment block. This block should contain a description of what is included within the file, the date of creation or modification and the authors' name(s). Often times there will also be a change log or version history included in the file comments as well. A full listing of the usable tags can be found at www.doxygen.com [4]. Here is an example of the file documentation for the source file `mainfile.c`.

```
/**
 * \file mainfile.c
 * \author Peter Alley
 * \date 7 JUL 2009
 * \version 2.3
 *
 * \Section
 * Here we have a description of the file.  As this is
 * just a demo and the file doesn't contain anything,
 * this is just a dummy description.
 *
```

```
 * \Section
 * By the way, there can be multiple sections.
 */
```

Macro and Global Variable Comments

Each defined macro and global variable needs to be commented. This will help keep track of which symbols have been used as well has what they have been used for. The tags for these comments tend to be fairly self explanatory, in that the `def` tag is for defined macros, `struct` is for structures, `typedef` is for defined variable types and `var` is for variable declarations. Again, a full list of these tags can be found at www.doxygen.com [4].

```
/**
 * \def ADC_CHANNEL
 * Channel to use on ADC
 */
#define ADC_CHANNEL 1
```

Function Documentation Comments

Perhaps the most important and most useful documentation comments are those for individual functions. These can appear either with the function prototype in a header file or with the implementation in the source file, or with both; and contain descriptions of the functions as well as what each of the input parameters should be and what the function returns. In a multiple-author file, there may also be an author tag.

These are some of the most common tags for function, and how they are used.

- brief: Following this should be a brief, one to two sentence explanation of the function. Should a more in depth explanation be desired or required, put the verbose explanation at the end of the comment block, separated from the tags above it by an empty line.
- param: This tag has two arguments separated only by spaces. The first argument is the name of the parameter in question, and the second is a description of what information the function is expecting to be in it.
- return: This tag indicates a brief explanation of what the return value for a function is. If the return type for the function is `void` then there is no need to include this tag.

```
/**
 * \brief This is a function that subtracts the first
 *    argument from the second
 * \param param1 The number to be subtracted from
 * \param param2 The number to subtract
```

```
 * \return The result of the subtraction problem
 *
 * This is the place to include a longer description of
 * the function as necessary, though in this case it is
 * hardly required due to the simplicity of the function.
 */
float subtract(float param1, float param2) {
    return param1-param2;
}
```

5.2.2 Non-Documentation Comments

While documentation comments purpose is to explain *what* the code does, the comments not intended for documentation cover *how* the code works. These comments are most useful to future programmers attempting to maintain the code. Every comment will give them more information and make their job easier. These are several situations where comments are highly useful.

- Lines with complicated logic. Whenever there is a logic check that includes more than one or two operations it is advisable to include a comment specifying, in English, what cases will result in a true result.
- Beginnings of conditional blocks. Whenever a section of code may or may not be executed, a comment explaining the block of code and when it will run should be included.
- Beginnings of loops. When entering into a loop (while, do-while etc.), it is useful to have a description of what the loop accomplishes, and how long it will loop for. If the loop is set to loop a set number of times, a reason why that number should be included.
- Any line with a non-obvious result or reason. If a single line, or even a collection of lines, has a result that is not easily understood, be it the reason, process, or result, a comment explaining the line(s) should be included.
- Logically separated blocks of code. Every so often a given task will be completed, and a new task started in the code. Each of these tasks should be commented at the beginning, briefly stating what the section does. This could be something such as “initialization” or “compute motor torque”. When there is a problem discovered in a certain portion of the program, these are the comments that are the first inspected, to find the correct section of code.

These are by no means the only places comments can be. Any time the programmer wants a comment, be it to write down his thought process, get an idea across, demonstrate pseudo-code or even to temporarily remove code from execution, a comment can be included. If in doubt about the usefulness or necessity of a comment, include it: someone, someday, may find it useful.

5.3 Code reuse: Functions and Libraries

One of the best ways to increase both the speed and accuracy of coding is to reuse code. By taking code from previously written programs or sections of the same program, correct operation is easier to ensure, and time is saved in not having to rewrite and debug the code. This section covers using functions and function libraries to this end.

5.3.1 When to use functions

The primary purpose of a function is to allow the same piece of code to be used in multiple locations in a program. While physically possible to write an entire program within the main function, it would be extremely difficult to follow and would likely have numerous bugs it would be difficult to remove. Functions allow for better organized, easier to maintain, and often result in smaller resultant programs.

The question then becomes, when are functions appropriate? Whenever the answer to any of the following questions is yes, then the code in question should probably be its own function.

- Does the section of code exist in more than one location? Repetition of code leads to a nightmare for maintenance as one location may be changed or updated and another location left as original.
- Will the code be useful for other/future projects? This comes into play more with the code use and libraries discussed later in this chapter.
- Is it a complex section of code with a relatively simple result? This can be complex calculations to result in a single value, or a series of initialization commands. In either case, even if the function would only ever be called once, separating the code out into a function can improve readability and maintainability.

5.3.2 Variable Scope and Arguments

The instant a function is entered, all previous local variables are no longer available. The only variables usable (in scope) are the global variables and the arguments passed to the function in the call. It is therefore important to pass the function any information it will need to produce its result.

There are, however, a few problems with simply adding another argument for ever piece of information the function requires. First, the function call could become unmanageably large as the number of arguments reaches 10 and beyond; and secondly if the function is being called from multiple locations or even multiple programs (see *Libraries* below) then every function call has to be updated when another argument is added.

5.3.3 `structs`

An easy solution to both problems is to use `structs`. Structures, or `structs` as they're commonly called, are blocks of memory that can hold several pieces of information in key-value pairs. This allows many pieces of information to be stored and passed using a single symbol, especially when the included information is relevant to a single theme (the dimensions of an object for example). When returning from a function, `structs` also have the ability to carry several results back to the calling function, as opposed to the single result that is normally possible. Finally, the addition of another key-value pair to a `struct` does not change its appearance in function calls. The same function call can be used without any modification, and the new value is available inside the function as long as it has been declared somewhere else.

Structs are used in a manor very similar to accessing fields of objects in object-oriented programming languages such as C++ or Java. The command to access a specific value in the `struct` is the name of the `struct` followed by a period and then the field name. E.g.

```
myStruct.myField = 4;
```

will cause the value 4 to be saved to the field `myField` in the `struct` `myStruct`, and afterwards calling

```
myStruct.myField
```

will return a result of 4 (the value stored in `myField`).

For more information on `structs`, refer to chapter 6.

5.3.4 Libraries

A library is a group of functions related to a similar purpose that is collected in its own file. Libraries aid in the organization of large programs, but their primary purpose is in code reuse.

For every commonly used programming language there are dozens or hundreds of libraries already in existence. Many of these are included when the compiler is installed, but on occasion some of the more esoteric libraries may have to be installed separately. These libraries include functions for most basic tasks which most programmers will need at some time or another. These include all the various math functions (square roots, trigonometry etc.), time functions (system clock, delays), file and system IO etc. The list goes on and on. It is also very common for programmers to write their own libraries, either for use within a single project where their purpose is mostly organization, to use in multiple projects to prevent having to re-write the same functions many times.

Take the example of a microcontroller such as the ATmega644p. There are only a limited number of ways to setup peripherals such as the ADC, and once setup once a given programmer is likely to keep with the same setup every time he uses it. Why should the programmer then have to rewrite the initialization and reading functions every time a new program is written. Instead, by putting

the functions into a library the first time, that library can be included in the next project and the functions used saving time and effort.

These libraries are no different than most other source files in execution. There should generally be a header file (see Headers and Header Files above) and will contain functions in the exact same format as a non-library source file. The only differences is that there should never be a `main` function in a library, and that libraries are specifically intended to be used across multiple projects.

5.4 ADC program revisited

With the ideas put forth in this chapter, it should be a simple matter to take the program found at the end of the previous chapter on ADCs and improve upon it by separating the ADC related sections of the code into separate functions and placing them within a library that shall be called ADC for simplicity.

Main.c

Here is what the primary source file should look like.

```
/**
 * /file Main.c
 * /author Peter Alley
 * /version 1.0
 * /date July 2010
 *
 * This file contains the main function for a program to
 * slowly flash an LED and exit the program if a fault is
 * detected in the system using the ADC on the ATmega644p
 */

#include "Main.h"

/** \fn int main(void)
 * \brief Main function that continually reads the ADC
 * and exits if it detects a fault.
 * \return Returns 0 and exits upon fault
 */
int main(void) {
  unsigned int LEDstatus;
  ADC_init();
  DDRA = 0x01;
  while(1) {
    PORTA = ~PORTA;
    _delay_ms(1);
    LEDstatus = ADC_read(ADC_CHANNEL);
    if (~PORTA && ((LEDstatus > LED_UPPER) ||
            (LEDstatus < LED_LOWER))) {
      // This will be reached only if the LED is bad
```

```
      break;
    }
    _delay_ms(998);
  }
  return 0;
}
```

This should be far more understandable than it was previously. Note, however, that there is only a single `#include` command pointing to a local file, and that there are several undefined symbols (such as `ADC_CHANNEL`). These symbols are defined in the header file.

Main.h

```
/**
 * /file Main.h
 * /author Peter Alley
 * /version 1.0
 * /date July 2010
 *
 * Header file for a program to slowly flash an LED and
 * exit the program if a fault is detected in the system
 * using the ADC on the ATmega644p.
 */

#ifndef MAIN_H_
#define MAIN_H_

#include <avr/io.h>
#include <util/delay.h>
#include "ADC.h"

/**
 * \def ADC_CHANNEL
 * Channel to use on ADC
 *
 * \def LED_UPPER
 * Upper limit for reading a 'good' LED (4V)
 *
 * \def LED_LOWER
 * Lower limit for reading a 'good' LED (2V)
 */
#define ADC_CHANNEL 1
#define LED_UPPER       820
#define LED_LOWER       410

/** \fn int main(void)
 * \brief Main function that continually reads the ADC
 * and exits if it detects a fault.
```

```
 * \return Returns 0 and exits upon fault
 */
int main(void);

#endif
```

This file includes the standard pre-processor check against multiple includes of the file, instructions to include the built-in libraries required for the ATmega644p as well as a local library `ADC`, and defines the previously magic numbers to understandable symbols. Finally is the function prototype for the single function found in the source file. While not strictly required to be in both the header and the source file, the documentation comments for the `main` function is included in both locations here. Especially for libraries having the documentation in the header file is more important than in the source file, however it can sometimes help to have the documentation in both locations.

ADC.h

```
/**
 * /file ADC.h
 * /author Peter Alley
 * /version 1.0
 * /date July 2010
 *
 * Header file for library containing functions for using
 * the ADC on the ATmega644p.
 */

#ifndef ADC_H_
#define ADC_H_

#include <avr/io.h>

/** \fn ADC_init
 * \brief Initializes the ATmega644p and sets it to use
 *   ADC1 (PA1)
 *
 * ADMUX:
 * Use AVCC as reference
 * Right adjust result
 *
 * ADCSRA:
 * Enable ADC
 * Select prescaler of 64
 */
void ADC_init(void);

/** \fn ADC_read
 * \brief Performs conversion on selected channel and
 *     waits for completion
```

```
 * \param Channel upon which to perform conversion
 * \return Converted value
 */
unsigned int ADC_read(unsigned char channel);

#endif
```

By this point this file should be easy to follow and understand.

ADC.c

```
/**
 * /file ADC.c
 * /author Peter Alley
 * /version 1.0
 * /date July 2010
 *
 * Source file for library containing functions for using
 * the ADC on the ATmega644p.
 */

#include "ADC.h"

/** \fn ADC_init
 * \brief Initializes the ATmega644p and sets it to use
 *     ADC1 (PA1)
 *
 * ADMUX:
 * Use AVCC as reference
 * Right adjust result
 *
 * ADCSRA:
 * Enable ADC
 * Select prescaler of 64
 */
void ADC_init(void) {
  ADMUX = 0x00 | (0b01 << REFS0) | (0b0 << ADLAR);
  ADCSRA = 0x00 | (1 << ADEN) | (0b110 << ADPS0);
  ADCSRB = 0x00;
  ADCSRA |= (1 << ADSC);        //begin first conversion.
                          //Done for timing purposes
  while (ADCSRA & ~(1 <<ADIF)) {}
          //wait until conversion complete
}

/** \fn ADC_read
 * \brief Performs conversion on selected channel and
 *     waits for completion
 * \param Channel upon which to perform conversion
 * \return Converted value
 */
```

```
unsigned int ADC_read(unsigned char channel) {
  while (ADCSRA & (1 << ADSC)) {}
  //wait for any previous conversion to complete
  ADMUX = (ADMUX & 0xe0) | channel;
  ADCSRA |= (1 << ADSC);         //begin conversion
  while (ADCSRA & ~(1 <<ADIF)) {}
          //wait until conversion complete
  return ADCW;
}
```

Finally the program is complete. These two functions are called by the main function to operate the ADC. Additionally, this library can be used in future projects without modification assuming the task is similar. The symbols that are still not defined in the code in this or the previous chapter (such as `ADMUX`) are the registers and locations of individual bits within the register which are defined within the header files installed with the compiler, and are included via the `#include <avr/io.h>` command in the header file.

5.5 Conclusion

Nothing presented in this chapter is required by the compiler. Everything in the header files can be included in the code file, and the program certainly doesn't care about whether it's commented. Nonetheless, it is important to spend the time and effort to do all this, otherwise in six months when the code needs modifying it will be difficult to determine what was going on. In the long run, proper documentation and organization will save time. For further information, or other opinions on the proper documentation and design of code, look for instructional books on C programming such as *C for Dummies* [14].

Chapter 6

C Data Structures

At this juncture there are a couple aspects about the C programming language that should be brought up. While they are covered in more depth in most any C programming course or text, they are very useful in microcontrollers so at least a brief look at them and their uses are warranted. Specifically, this chapter will cover arrays (called matrices in some languages) and structs.

6.1 Arrays

To compare arrays to mathematics, arrays are Cartesian collections of data with at least one dimension. In other words, a single data point is just that, a point with zero dimensions. If additional data points are added next to the first, it becomes analogous to a line, or a shape with one dimension. This series of data is a one-dimensional array. More dimensions can be added on, however the frequency of finding multi-dimensional arrays decreases drastically as the number of dimensions increases. This is both due to the lack of a need for those arrays as well as the complexity and space requirements of storing arrays with large numbers of dimensions. The Cartesian reference indicates that each piece of data has a unique set of coordinates to reference it, similar to how a Cartesian graph operates.

6.1.1 Declaring Arrays

There are two ways to declare an array in C, depending on if the array will be initialized or not.

```
char firstArray[12];
```

The first method, seen above, is used when the array is not being initialized at the same time as its creation. This causes an appropriate block of memory, in this case 10 /textttchars and store the address of the first `char` to the name firstArray. This method is used when declaring global variables, or when the data does not exist yet to store in the array, such as when taking sensor readings.

```
char array1[] = "Hello World";
char array2[] =
    {'H','e','l','l','o',' ','W','o','r','l','d','\0'};
```

These two lines are equivalent, and merely differentiate the storage of a string vs a series of values. Functionally they are identical in this case, however when storing numerical data, the second method is what will be used. The '\0' character is a null character indicating the end of a string, and was appended automatically in `array1`. In this case the length specification is left out of the declaration and the array becomes the required length for the initialization, in this case 11 `char`s.

It is also permissible to specify the array length even when initializing at the same time. This can have one of four results. First, the specified size can be the same as what would be necessary, in which case there is no difference to what occurred above. Second, the specified size could be smaller than the length of the initialization, in which case an error will occur and the compiler will be most displeased. Third, the specified size could be larger than necessary. In this case the remaining space in the array will be filled with zeros. The final case is a special case that occurs when a string is used to initialize an array and there is only enough space for the letters and not for the terminating null character. In this case the null character will not be appended and the array will merely be filled with the characters in the string.

6.1.2 Accessing Arrays

Accessing the data within an array for either reading or writing is a fairly simple process. Below is an example of both writing to and reading from an array. There are two important things to note here. The first is that all access to the array after its initialization is performed on a per datum basis. It is not possible to read or write multiple pieces of data at once. The second piece of important information is that arrays in C are zero indexed. That means that the first piece of data is stored at location 0.

```
int array[] = {1,2,3,4};
int x = 2+array[2];
array[3] = 15;
```

At this point `x` is equal to 5 (2+3) and `array` is {`1,2,3,15`}.

6.1.3 Multi-Dimensional Arrays

Up to now all the examples have been using arrays with only a single dimension. To add a second dimension an extra pair of square brackets is added for each dimension. In a two dimensional array the order of the coordinates is row followed by column. For example

```
char array1[4][2] = {0,1,2,3,4,5,6,0};

char array2[4][2] = {{0,1}{2,3}{4,5}{6,0}};
```

```
char array3[4][2];
array3[0][0] = 0;
array3[0][1] = 1;
array3[1][0] = 2;
array3[1][1] = 3;
array3[2][0] = 4;
array3[2][1] = 5;
array3[3][0] = 6;
array3[3][1] = 0;
```

all produce the same result with four rows and two columns. Additional dimensions can be added in similar fashion. It is also possible to only define the values for part of the array. If there are not enough values to fill the array (such as if the zero was left off `array1` or `array2`) there will still be a value in that memory location, though it is not known. Any such unknown values are likely to be zero, however if that memory location had previously been used to store something else, that value will remain. It is therefore important to explicitly set each value of an array before it is read.

These are only the basics of working with arrays. There is a wealth of other sources available for those who want or need more than is covered here.

6.2 `structs`

In the C programming language the `struct` is a specific data structure that is somewhat analogous to objects from languages such as C++ or Java. A `struct` in C is a collection of one or more pieces of data stored in named fields within a single named object. Unlike arrays the pieces of data do not all have to be the same type. One piece could be an `int`, another a `float`, and a third be a pointer to a function. In this manor it is actually possible to almost directly convert C++ into C by turning all the objects in C++ into `structs`.

The similarities to objects do not end there. `structs` are also accessed via dot notation. This means that to access a specific field or piece of data in the `struct`, one uses the `struct`'s name, followed by a period, followed by the field name. For example, to store the value 5 into a field in `exampleStruct` one would enter:

```
exampleStruct.exampleField = 4;
```

This method of storing and accessing data requires a little more effort than simply storing the value in an ordinary variable, however it becomes far more useful when passing data from function to function (see below).

6.2.1 Declaring `structs`

There are three methods available for declaring `structs`. The first method is to follow `struct` keyword with the type name of the `struct`. Note that this is the general name for all `structs` using the set of fields. Following the type name is

a list of all the fields that will be contained in the `struct` within a set of curly brackets. If an instance of the struct is to be immediately created, the instance name or names follow the closing curly bracket, though this is not required.

```
struct exampleStructType { int field1;
                           float field2;
                           char[5] string;
} structInstance1, structInstance2;
```

In this example two separate `struct`s, `structInstance1` and `structInstance2`, are created. Each can store the same types of data (an `int`, a `float` and an array of five `char`s), and the two `struct`s are both of type `exampleStructType`. Note the semicolon after each field listing of data type and field name, as well as the semicolon at the end of the instance list. To create instances of a predefined `struct` later, the list of fields and accompanying curly brackets are not included. For example:

```
struct exampleStructType structInstance3;
```

The second method for declaring a `struct` is to precede the `struct` keyword with they `typedef` keyword and include before the list of instances to create, a name for the `struct` that will be they keyword for the type. Due to the similarity of usage to the name already included (`exampleStructType` in the example above), it is common to just append `_t` for the keyword.

```
typedef struct keywordExStruct { int field1;
                                 float field2;
                                 char[5] string;
} keywordExStruct_t structInstance1, structInstance2;
```

Note the lack of a comma after the new keyword. This is important as it distinguishes the keyword from the instance names. Using the `typedef` keyword in this fashion will turn `keywordExStruct_t` into a keyword akin to `int` or `float`, and remove the need of including the `struct` keyword in the future. Therefore future instantiations of the `keywordExStruct_t` is simply

```
keywordExStruct_t structInstance3;
```

The final method is an extension of using `typedef`. In this case the initial type name is left out. The drawback of this method is that instances cannot be created immediately. This drawback is minimal however, as shall be seen below.

```
typedef struct { int field1;
                 float field2;
                 char[5] string;
} keywordEx2_t;

keywordEx2_t structInstance1, structInstance2;
```

In any of the cases, whether using `typedef` or not, filling a `struct` with data can be tedious. Once the `struct` has been instantiated, it is only possible

to access a single field at a time. For large `structs` this can turn into a mess. There is an exception to this however. When the instance is first created, it can be set equal to a list of the values for each field in this manor:

```
keywordExStruct_t structInstance4 = {8, 3.2, "foo"};
```

The order of the data must match the order of the fields, but this is the simplest way of filling large `structs` if the data is known at the time of creation. `keywordExStruct` was used in this example, however the method of filling a `struct` applies to either method of declaration. This then minimizes the drawback of the third method of declaration, as it is most advantageous to fill `structs` as they are instantiated. A better method is to use a list format as above, but combine it with the dot notation. This allows for setting specific fields rather than relying on the order being correct. The whitespace in the example below is for readability and is ignored by the compiler.

```
keywordExStruct_t structInstance5 = {   .field1 = 8,
                                        .field2 = 3.2,
                                        .string = "foo"};
```

6.2.2 Uses for `structs`

The uses for `structs` are many and varied. Already mentioned was the possibility of emulating C++ or Java objects and storing references to functions, however these are not the purpose of introducing `structs` herein. Instead this text will introduce their use as another method of accessing SFRs (see chapter 2) and using them as the sole argument to functions, especially those in libraries referenced by different projects.

SFR access

Thus far access to individual bits of SFRs has been a clunky troublesome prospect requiring bitwise logic, left-shifting and masking. With a properly prepared library the bit fields and individual bits of the various SFRs become trivial to access. It has previously been mentioned that any data type can be included in a struct. This includes bit fields of any width, signed or unsigned. To create a bit field in a struct use the `signed` or `unsigned` keyword as appropriate (in this usage they will all be unsigned) and follow the field name with a colon and the number of bits wide the field is. Here is an example using the ADCSRA register (Analog-Digital Converter Status Register A):

```
typedef struct {
        unsigned _ADPS0 :1;
        unsigned _ADPS1 :1;
        unsigned _ADPS2 :1;
        unsigned _ADIE  :1;
        unsigned _ADIF  :1;
        unsigned _ADATE :1;
```

```
        unsigned _ADSC  :1;
        unsigned _ADEN  :1;
} __ADCSRAbits_t;
extern volatile __ADCSRAbits_t   ADCSRAbits
    __asm__ ("0x7A") __attribute__((section("sfr")));
```

The `struct` defined above consists of eight 1-bit wide bit-fields with the names of each field being the same (with the addition of an underscore) as are in the datasheet for the ATMega644p. It is vital to note that the least significant bit (bit 0) is the first listed in the definition. The final line of the above code instantiates the `struct` under the name `ADCSRAbits` and tells the compiler what memory address to reference. Specifically, each part of the final line mean the following:

- `extern`: This keyword explicitly states that the variable defined on this line is a global variable. The variable is therefore accessible from anywhere in the program without special effort.
- `volatile`: Often times a compiler will attempt to optimize the code. This can cause the compiler to store a value retrieved from memory in a temporary variable for usage multiple times, rather than reloading it from memory each time. The `volatile` keyword prevents this optimization which is important in this case as things outside the code can affect what is stored in the register.
- `__ADCSRAbits_t`: The name of the type as defined by the `typedef` above.
- `ADCSRAbits`: The variable name. This is name of the variable that can be referenced in the code.
- `__asm__ ("0x7A")`: This is a special command to the compiler instructing it to include what is in the parenthesis directly into the compiled code as is. In this instance it is telling the compiler that the variable `ADCSRAbits` refers to the register found at the memory address `0x7A`.
- `__attribute__((section("sfr")))`: This provides additional information to the compiler for error checking regarding what the variable is for, and is not required.

Useful as this is, this `struct` provides no method of reading or writing the entire register at once. Another thing that would be useful is to be able to address all three `ADPSn` bits simultaniously. In order to do this another keyword must be introduced: `union`. A `union` allows multiple different data types to address the same memory. In this case the different data types will all be `struct`s though with differing bit-fields. A `union`is declared in much the same way as a `struct`, however each member of a `union` is a separate `struct` which does not require a name. Here is the same type as above (`__ADCSRAbits_t`) adapted to allow easier access to larger bit-fields.

```
typedef union {
        struct {
                unsigned _ADPS0 :1;
                unsigned _ADPS1 :1;
                unsigned _ADPS2 :1;
                unsigned _ADIE  :1;
                unsigned _ADIF  :1;
                unsigned _ADATE :1;
                unsigned _ADSC  :1;
                unsigned _ADEN  :1;
        };
        struct {
                unsigned _ADPS  :3;
                unsigned        :5;
        };
        struct {
                unsigned _w     :8;
        };
} __ADCSRAbits_t;
```

The first `struct` is the same as before. The second `struct` provides access to all three `ADPSn` bits simultaneously. This leaves five bits that already have been named in the first `struct` individually and don't require further naming. Thus the name is left out of that line. Finally a third `struct` groups the entire byte into a single field. All three of these `struct`s provide access to the same register, just with different layouts.

AVR, the makers of the ATMega644p, do not provide a header file with these definitions. It therefore falls upon the individual user to create or find such a header file. While it seems a daunting task the author created such a file with less than ten hours of work, and has proven to be very useful. As an example, in order to set the ADC prescaler to a division factor of 8 (`ADPS2:0 = 0b011`) without changing any other bit in the register now requires only

```
ADCSRAbits._ADPS = 0b011;
```

rather than

```
ADCSRA = 0b011 | (ADCSRA & 0b11111000);
```

While both take a single line, the new method is easier to understand and to code.

Function Arguments

When first creating libraries it is not unusual to unintentionally limit the usefulness by limiting the functions, especially those responsible for initializing peripherals, to specific tasks. It is much more useful to have a single function used for initialization of a peripheral with arguments for the various options and always use that function. This reduces the number of functions required for any particular task as only one is required as opposed to a separate function for each slightly different possibility.

There are two issues with doing this however. First, there could be numerous options, and while from the compiler's side it isn't an issue, it can cause problems for the programmer or whomever has to look at the code later. The second problem comes into play when an additional argument is required at a later date. This could be caused by a change to the hardware, be it update or change to a different controller, or due to a previously overlooked option. If suddenly a new argument is required, either a new function has to be written or all instances of function calls to the old version must be updated. The alternative to this is to only require a single argument for such a function - a `struct` containing all the information about the various options. If something changes in the future, a new field can be added to the `struct` in the library and old code using the functions will not require modification. The old code will simply ignore that additional field. A further benefit is that the struct can be used to store data being returned from the function. This is especially helpful when more than a single value must be returned.

Here is an example of how this can be used in initializing the ADC. All options available on the ATmega644p's ADC are available via this `struct` and function except for the `DIDR0` register which is purely optional anyways (its function is merely to save a tiny bit of power).

```
/**
 * \typedef ADC_init_struct
 * struct containing all config options for ADC
 */
typedef struct {
    char channel;      ///<Channel to use
    char left_adjust;  ///<left or right adjust result
    char reference;    ///<reference voltage source
    char auto_trigger; ///<start conversion based
                       ///<on external source
    char auto_source;  ///<source for auto conversion
    char enable_interrupt; ///<enables conversion
                           ///<complete interrupt
    char prescaler;    ///< ADC clock prescaler
} ADC_init_struct;

int main(void) {
    ADC_init_struct ADC_struct;
    ADC_struct.channel = 0;
    ADC_struct.left_adjust = 0;  //right adjust
    ADC_struct.reference = 1;    //AVCC
    ADC_struct.auto_trigger = 0; //off
    ADC_struct.auto_source = 0;  // doesn't matter
    ADC_struct.enable_interrupt = 0; //off
    ADC_struct.prescaler = 0b110; //divisor = 4
    ADC_init(ADC_struct);
}

/**
```

```
 * \func ADC_init
 * \brief Initializes ADC and runs first (extra long)
 *     conversion
 * \param config  struct containing all
 *     configuration information
 *
 * Accepts a a struct (ADC_init_struct)
 * containing options for every ADC configuration
 * excepting only DIDR0
 */
void ADC_init(ADC_init_struct config) {
    ADMUXbits._MUX = config.channel;
    ADMUXbits._ADLAR = config.left_adjust;
    ADMUXbits._REFS = config.reference;
    ADCSRAbits._ADATE = config.auto_trigger;
    ADCSRBbits._ADTS = config.auto_source;
    ADCSRAbits._ADIE = config.enable_interrupt;
    ADCSRAbits._ADPS = config.prescaler;
    ADCSRAbits._ADEN = 1;
    ADCSRAbits._ADSC = 1;    //begin first conversion.
                //Done for timing purposes
    while (ADCSRA._ADIF == 0) {}
               //wait until conversion complete
}
```

The uses for both arrays and `structs` are far more numerous than mentioned here, and there is a wealth of information available on each for the interested. This has only been a brief introduction including some uses that may not be immediately obvious.

6.3 Pointers

Whenever programming in C, pointers will eventually float to the surface. A pointer is a data type who's value is the memory address for some other piece of data. Pointers have a variety of uses and applications, most of which relate either to improving performance for repetitive operations or for allowing access to stored data that might not otherwise be available in the current program context.

6.3.1 Pointer Syntax

A pointer is declared by adding and asterisk (`*`) before the name of the pointer. The data type when declaring the pointer is the same type as that which the pointer will be referencing, as though a variable of that data type were being declared. This pointer is set equal to the variable it should reference with an ampersand (`&` preceding it. This ampersand causes the address of the variable rather than its value to be retrieved. A pointer to the `int var` could look something like this:

```
int var1 = 1;
int *var_pointer = &var1;
```

At this point the value stored in memory for `var_pointer` is the address of `var1`. When using the pointer to access its referenced value, a process called dereferencing, include the asterisk before the pointer name.

```
*var_pointer = 4;
```

This line does not change the value stored for `var_pointer`, instead the value of `var` was changed to 4. If the asterisk were omitted, the value of the pointer itself would change. This can be useful when using pointers for array operations as will be covered later.

6.3.2 Pointers and arrays

Pointers are commonly found when working with arrays, as they provide an alternative to the square bracket notation mentioned above. As arrays are created as a single block of values when declared, a little bit of math is all that is required to find any location in the array. Even multi-dimensional arrays are created this way, each row being appended to the one before it. To reach any value in a row, assuming there is a pointer to the first value, add the index of the desired value. Any calculations for the width of the data types is taken care of automatically. Note that different processors use different widths for data types. For example, the ATmega644p has 16 bit `int`s, where as most modern computers use 32 bit ints. In the case of two dimensional arrays multiply the row number by the number of columns and then add the column number of the desired value. It is interesting to note that the variable for an array is actually a pointer to this block of memory. This means that the syntaxes for arrays and pointers can be used somewhat interchangeable. Below are several examples

```
int array[] = {0,1,2,3,4};
int *ptr = array;
ptr[0] = 0;      //Using a pointer with array syntax
*(array+3) = 3; //using the array symbol as a pointer
ptr++;           //ptr now points to array[1]
ptr[0] = 1;
```

In these examples, once the array is declared and initialized, none of its values ever change. The values are merely overwritten with the same value as they already were.

6.3.3 Pointers and other data structures

Between the use of pointers, arrays and `struct`s, most any other type of data structure can be created. Here are several common structures and how they can be created.

Linked Lists

A linked list is a series of items, each one linking to the next. These items can be any data type or structure, including pointers, arrays, `struct`s or anything else. The link to the next item in the list is a pointer storing the address. The actual method for creating a linked list could be to use an array with a column for the pointer, however this only works if the array is of `int`s or something similar. What works better is to use `struct`s for each item in the list, with one or more fields containing the object and another field for the pointer to the next `struct`. The following is an example of using a very generic linked list. The `void` type for the `data` pointer allows that data to be of any type, thus allowing the list to be composed of items of differing types.

```
struct llstruct {
    void *data;
    struct llstruct *next;
};

int main(void) {
    struct llstruct item1;
    int array[] = {0,1,2,3,4};
    item1.data = &array;
    struct llstruct item2;
    item1.next = &item2;
    item2.data = &array;
    return 0;
}
```

Doubly Linked Lists

Doubly linked lists are almost identical to linked lists, with the exception that they also have a link to the previous item in the list. This removes the limitation that only items later in the list are findable from a given item. One very handy use for both, singly and doubly linked lists is to have a dynamically resizing array. While somewhat less time and space efficient than a single pre-sized array, each row of the array could be its own item in the list.

Trees

A tree is a collection of items (called nodes) branching out from a single point (called the root). Each node in the tree may have 0 or more child nodes. These data structures are commonly used in a variety of path-finding and other decision making algorithms. The creation of a tree is very similar to linked lists, but with multiple options for the next item. Nodes will very often have a link back to its parent, so the route to the node can be retraced.

6.3.4 Pointers in pass by reference

Normally, when a function is called, copies of any arguments are made and those copies are what are used in the function, and then are lost when the function returns. This is called being passed by value. Fairly often the functions are supposed to change the original values, especially as only a single value can be returned. The solution to this is to pass a pointer to the variable (pass by reference), which will allow the function to work on the original data and location rather than a copy of it. Any changes made in the function will remain after it returns. This is commonly done when a few variables are needed. For larger numbers of variables, they are often put in an array and the array is passed. As the array variable is technically a pointer itself, passing an array to a function is automatically pass by reference. Pointers are also commonly used for passing structs, both due to the time involved in copying all the data in a struct, as well as conserving stack space. The syntax for passing a pointer is to use an asterisk in the function declaration, but not in the call itself.

```
void ptrFunc(int *ptr) {
  *ptr = 5;
}

int main(void) {
  int var = 2;
  int *ptr = &var;
  ptrFunc(ptr);
  return var;           //returned value is 5
}
```

As it is somewhat cumbersome to constantly create pointers for the variables, the address of `var` can be sent using the ampersand notation eliminating the need to create `ptr`.

```
  ptrFunc(&var);
```

6.3.5 Function pointers

When discussing `struct`s it was mentioned that they could be used as a primitive form of object. One important aspect of objects in object oriented programming is to have callable functions which may do different things depending on the object. A similar form is possible using `struct`s and pointers to functions. The creation of a function pointer is very similar to that of any other function. The type is the return type of the function, and it is set equal to the address of the function using an ampersand and the function's name. The only difference is the list of the function's argument's types (no argument names) is included within parentheses immediately after the pointer name, not after the function name. Calling them is done the same way as any other function. Here's an example:

```
int add(int a, int b) {
```

```
  return a+b;
}

int main(void) {
  int *funcPtr(int,int) = &add;
  return funcPtr(4,3);          //returns 7
}
```

6.4 Conclusion

The uses for these various data types are many and varied. Only the barest surface was scratched here, though it should be enough to get one started and stir the imagination for other uses they may have. Most any text focusing on C programing will have more. There is also a wealth of information available on the internet, so just doing a keyword search for any of these data types should return many results. A reasonable starting point is Wikipedia and the sources referenced there, or a book on C programming such as *C for Dummies* [14].

Chapter 7

Serial Communications

Serial communications is a very broad topic that covers numerous formats, standards, customs etc. It refers to any digital signal where data is transferred a single bit at a time. Common examples include USB (Universal Serial Bus), digital telephone lines, telegraphs, RS232 (commonly called the serial port on computers), ethernet and the list continues. There are three serial communications peripherals on the ATMega644p: SPI (Serial Peripheral Interface), USART (Universal Synchronous/Asynchronous Receiver Transmitter) and TWI (2-wire Serial Interface) also known as I^2C(pronounced I-squared C or I-two C).

7.1 USART and RS232 Serial

The USART (Universal Synchronous/Asynchronous Receiver Transmitter) is the peripheral most often used for communications to and from a computer or another microcontroller. This is because it is very simply converted into a signal using the RS232 standard with the use of another IC. This second IC's purpose is to ensure the voltage level of the signal is within the plus or minus 3 to 15 volts, protect the microcontroller from shorts and grounds, and provide the signals for the other pins in the RS232 standard.

7.1.1 RS232

The RS232 serial communications standard is used by the port commonly labeled the "Serial Port" on desktop or laptop computers, using a 25 pin D-sub connector. Of these 25 pins only 10 are used by the standards, and only two actually carry the data being transfered. The two devices communicating under this standard are classified as "Data Terminal Equipment" (DTE) or as "DATA Communication Equipment" (DCE). When communicating with a desktop computer, the computer will be the DTE and the microcontroller will be the DCE. Table 7.1 lists the pins used by RS232 and their assignments.

The ring signal on pin 22 dates back from the time when the standard was

Name	Purpose	Brief	Pin#
Data Terminal Ready	Tells DCE that DTE is ready to be connected	DTR	20
Data Carrier Detect	Tells DTE that DCE is connected	DCD	8
Data Set Ready	Tells DTE that DCE is ready to receive	DSR	6
Ring Indicator	Tells DTE that DCE has detected a ring signal on telephone line	RI	22
Request to Send	Tells DCE to prepare to accept data from DTE	RTS	4
Clear to Send	Allows DTE to transmit		5
Transmitted Data	Data line from DTE to DCE	TxD	2
Received Data	Data line from DCE to DTE	RxD	3
Common Ground		GND	7
Protective Ground		PG	1

Table 7.1: Pin assignments for RS232 standard. [5]

primarily used for communications between teletypewriters and modems. In more modern circumstances the pin triggers a hardware interrupt (see chapter 8 for information regarding interrupts) that informs the DCE to prepare for the possibility of communications.

As useful as the RS232 standard is for communicating with desktop computers and other devices, it does require additional hardware to use with the ATMega644p, as well as most other microcontrollers. Besides requiring the actual port, an RS232 transceiver such as the MAX202CSE produced by MAXIM is required, as well as possibly requiring additional voltage sources or level shifters depending on the voltage that the microcontroller is operated at. This being said, there are two and three wire methods of communication via the USART, which is actually what is used to communicate with an RS232 transceiver.

7.1.2 USART

USARTs generally use a three-wire setup to communicate with other devices such as microcontrollers and RS232 transceivers. Two of these wire are for the actual communications (one for each direction) and the third wire is ground. If the data transfer only requires one direction, the second data line can be left off. The use of separate data lines for each direction does allow for full duplex communications (data is sent and received simultaneously), however this is not required.

Synchronous v. Asynchronous communications

Asynchronous communications means that information can be sent whenever its ready, and the receiver cannot expect a transmission to arrive at specific

times. Synchronous communications means that both the transmitter and receiver know exactly when each byte will start and stop. This requires that data be continually transmitted to keep the two devices in sync. When there is no data to be sent, the ASCII "SYN" character is generally used as padding. Unless the required data transfer rate is very high in comparison to the maximum possible data rate, using asynchronous communications is often a better choice as it is somewhat more robust and easier to maintain.

Character Framing

When using a USART, each character is a separate transmission, with a configurable number of bits per character. The characters transmitted via a USART can be anywhere from 5 bits to 9 bits long with 7bits (ASCII character) or 8 bits (1 byte) being the most common. Each character is preceded by a single start bit (logical low) and followed by one or two stop bits (logical high), and may contain a single parity bit after the final data bit and before the stop bit(s), the total length of a transmission can range from 7 bits to 13 bits [5]. It is vital that both devices are configured to use the same options of character length (5-9), parity (none, even, odd), number of stop bits (1 or 2) and data rate, otherwise errors will arise and if any characters are actually accepted by the receiver, they will likely be gibberish.

Communications Errors

There are several possible errors that can be detected if a problem occurs during transmission or reception

- Overrun Error: Occurs when data is received before the previously received data can be processed. Different devices have different buffer sizes to store received data in, and if the buffer fills up this error is triggered. The ATMega644p can only buffer a single character, so it is important to read that character as soon as possible after it arrives and either use it, ignore it, or (most likely) store it until it can be used.

- Underrun Error: Occurs when the transmit buffer is empty and the USART has completed transmitting the previous character. In general, this is treated as an indication that nothing is being transmitted as in asynchronous communications data does not need to be continually transmitted. This error can indicate a problem in synchronous communications as underrun is more serious there.

- Framing Error: Occurs when the start and stop bits do not align properly. If the receiver does not find the stop bit(s) where they are expected, this error is triggered indicating something went wrong with the transmission. If this occurs regularly it could indicate that the two devices are configured for different character frames.

- Parity Error: Occurs when the parity calculation does not match the parity bit. When parity is enabled and set to even, the parity bit should be logical high when an even number of bits in the character are logical high, or when the number of logical high bits is odd and parity is set to odd. If this bit does not match the receiver's calculations a parity error is thrown indicating that there was an error in the received data. If the transmission lines are particularly noisy and two bit errors occur, they can cancel each other out in the parity calculation and the parity calculations will be correct, even if the data is incorrect. [5]

7.1.3 USART Communications Example

Here is an example program using the USART0 on the ATmega644p. Its purpose is merely to echo anything transmitted to it, and immediate transmit it back.

```
#include <avr/io.h>
#include <util/delay.h>
#include "../libraries/reg_structs.h"

/**
 * \func serial_init
 * \brief initialize USART
 * Settings:  8 bit char, asynch, no parity, 1 stop bit
 */
void serial_init() {
  UCSR0Bbits._RXEN0 = 1;  //enable transmit
  UCSR0Bbits._TXEN0 = 1;  //enable receive
  UCSR0Cbits._UCSZ0 = 3;  //8 bit char
  UBRR0bits._w = 25;  //19.2k baud rate
}

int main(void) {
  serial_init();
  while(1){
    while(!UCSR0Abits._RXC0) {}
    UDR0bits._w = UDR0bits._w;
        //transmit what was received
  }
}
```

Ordinarily contents of the `UDR0` register would be saved to memory, however this is not needed in this case. The act of writing data to `UDR0` is what begins the transmission, so ordinarily it would be set to some value from memory.

7.2 Serial Peripheral Interface (SPI)

Serial Peripheral Interfaces (SPI) are four-wire, full-duplex communications busses run in a master-slave mode. This means that any time communications occur between the devices on the bus, it is the master that initiates the

communications. Slaves can only respond to the master. More than two devices can be connected to the same data bus, however there can only be a single master and each slave requires an individual slave select line. Therefore the actual number of lines on the bus ranges from 3 (single slave, uni-directional communication) to three plus the number of slaves on the bus.

7.2.1 SPI Bus

Figure 7.1 below gives an example of a setup where there are three slaves connected to a single master via SPI. The six wires are:

- SCK: System ClocK. This is a clock signal that the transmission uses. It is generated by the master and is not required to be a constant frequency. While often generated by a hardware clock in the master, it is possible to manually control the line via software in extraordinary situations. Also called SCLK or CLK.
- MOSI: Master Out Slave In. This is the line for transmission from the master to the slaves and is uni-directional. Also called SIMO, SDO, DO, SO.
- MISO: Master In Slave Out. This is the line for transmission from a slave to the master and is also uni-directional. Also called SOMI, SDI, DI, SI.
- SSn: Slave Select n. In single slave situations this is just SS or Slave Select. This line connects the master to only a single slave device and is used to tell that device to pay attention to the transmissions. Only a single slave select should be pulled low at a given time, otherwise the slaves transmissions may interfere with each other. Also called nCS, CSB, CSN, STE, nSS.

7.2.2 Operation

During times at which the master is not specifically communicating with one of the slaves, all of the pins are dormant. SCK is held at either logical high or low depending on the configuration of CPOL, all nSS lines are held at logical high and the MISO and MOSI lines can be anything (often tri-state, see below). Once the SS for a given device is pulled low, one of two thinks will happen depending on the clock phase configuration (CPHA). If the phase is set to 0 both the master and slave immediately pull their respective out lines to the correct level for the first bit to be transmitted. For one or the other this is often logical low, though does not have to be. Half a cycle later the clock switches high to low or low to high (depending on CPOL) and both master and slave sample their input lines. On the trailing edge of the cycle (half a cycle later, when the clock returns to its original value), data is propagated and the MISO and MOSI values change. This continues until the clock is stopped and returned to its idle value. After that the SS pin can be returned to a high value and MISO

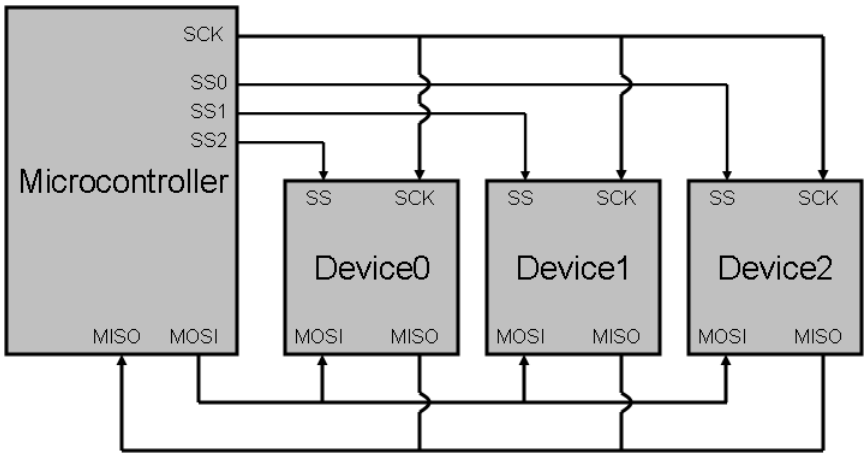


Figure 7.1: 3 devices connected to a microcontroller using SPI.

and MOSI return to being uncared about. Because communication only occurs while the master is driving the clock, if the master is expecting some response from the slave after transmitting, the master must continue operating the clock, and most likely transmitting zeros, until the slave has finished transmitting. In the case where CPHA is set to 1, data is propagated on the leading clock edges and sampled on the trailing, rather than vice versa.

Figure 7.2 shows the states of each data line at each clock cycle four configurations of CPOL and CPHA. The long vertical lines indicate the leading clock edges. Here is a brief explanation of each of the four possible configurations, including the commonly used mode numbers for each:

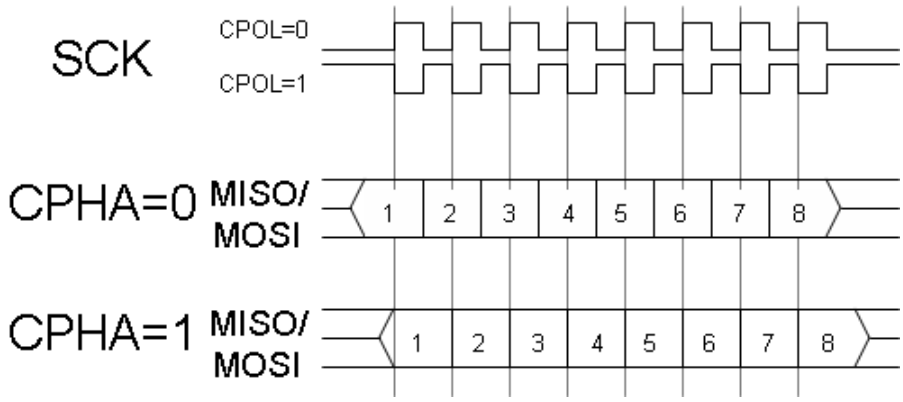


Figure 7.2: The four modes of SPI clock polarity and phase.

CPOL=0 (base clock value is 0)

- CPHA=0: Data is captured on rising edge and propagated on falling edge. (Mode 0)
- CPHA=1: Data is propagated on rising edge and captured on falling edge. (Mode 1)

CPOL=1 (base clock value is 1)

- CPHA=0: Data is captured on falling edge and propagated on rising edge. (Mode 2)
- CPHA=1: Data is propagated on falling edge and captured on rising edge. (Mode 3)

It was mentioned that MISO and MOSI are often set to tri-state when not in use. Tri-state is a special third state in binary logic which is obtained by connecting the line through a large resistor (such as 50kΩ) to ground to prevent it from affecting the level of the line. The large resistor is designed to be enough to prevent it from reducing the voltage on the line any great amount, while at the same time preventing the device from sourcing a Voltage on the line. This is also called a high-impedance state.

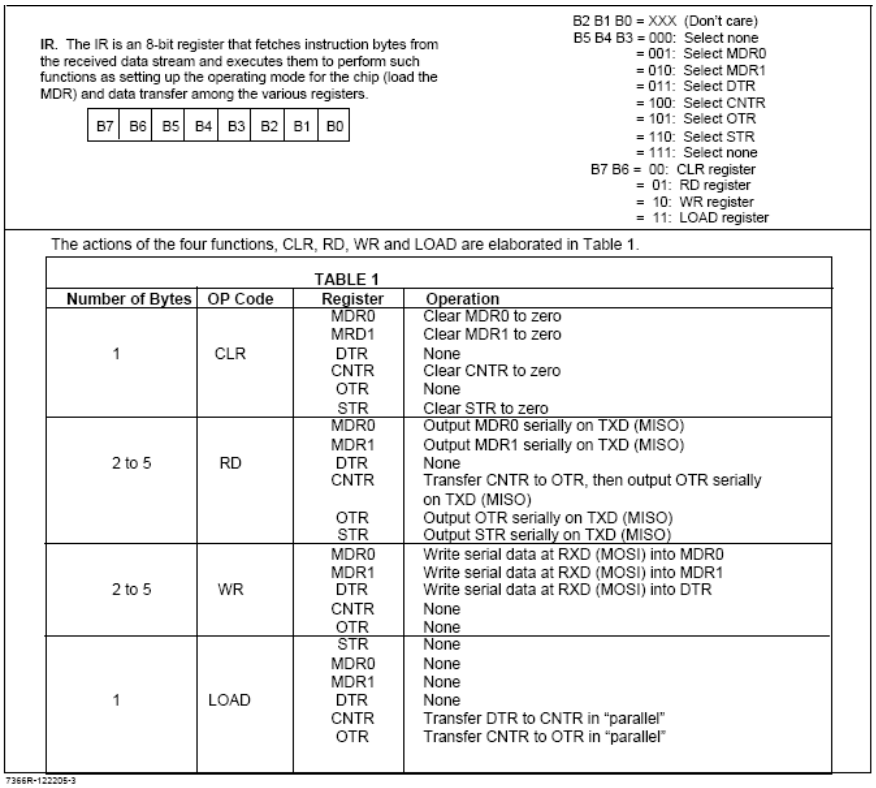
7.2.3 SPI Communications Example

The following program can be used to setup and read the 4-byte counter register from an LS7366R 32-bit quadrature counter. This is a device used for counting ticks on an encoder (see chapter 9). It is somewhat longer than the USART example as both the SPI peripheral on the ATmega644p must be configured to the requirements of the LS7366R as well as the LS7366R needing configuration. Once they are both configured and started, the program waits for one second and then reads the current count. This operation requires transmitting five bytes. The first byte is the actual command, telling the LS7366R what register is being read. The following four transmitted bytes are all ignored by the LS7366R as it is transmitting the counter value. This transmitted data is read from the same register as the transmitted data was written to. This program saves the incoming data byte by byte into an array of `char`s.

Determining commands for encoder counter

Before programming the microcontroller, the commands for the encoder counter must be determined. This includes the command to write to registers, the settings for the registers, and the command to read the counter itself. Figure 7.3 is an excerpt from the data sheet which explains how to form commands to be transmitted to the encoder counter.

The section at the top of figure 7.3 explains the instruction register on the encoder counter, and has a bit by bit description of each command. In this



IR. The IR is an 8-bit register that fetches instruction bytes from the received data stream and executes them to perform such functions as setting up the operating mode for the chip (load the MDR) and data transfer among the various registers.

B7	B6	B5	B4	B3	B2	B1	B0

B2 B1 B0 = XXX (Don't care)
B5 B4 B3 = 000: Select none
= 001: Select MDR0
= 010: Select MDR1
= 011: Select DTR
= 100: Select CNTR
= 101: Select OTR
= 110: Select STR
= 111: Select none
B7 B6 = 00: CLR register
= 01: RD register
= 10: WR register
= 11: LOAD register

The actions of the four functions, CLR, RD, WR and LOAD are elaborated in Table 1.

TABLE 1

Number of Bytes	OP Code	Register	Operation
1	CLR	MDR0	Clear MDR0 to zero
		MRD1	Clear MDR1 to zero
		DTR	None
		CNTR	Clear CNTR to zero
		OTR	None
		STR	Clear STR to zero
2 to 5	RD	MDR0	Output MDR0 serially on TXD (MISO)
		MDR1	Output MDR1 serially on TXD (MISO)
		DTR	None
		CNTR	Transfer CNTR to OTR, then output OTR serially on TXD (MISO)
		OTR	Output OTR serially on TXD (MISO)
		STR	Output STR serially on TXD (MISO)
2 to 5	WR	MDR0	Write serial data at RXD (MOSI) into MDR0
		MDR1	Write serial data at RXD (MOSI) into MDR1
		DTR	Write serial data at RXD (MOSI) into DTR
		CNTR	None
		OTR	None
1	LOAD	STR	None
		MDR0	None
		MDR1	None
		DTR	None
		CNTR	Transfer DTR to CNTR in "parallel"
		OTR	Transfer CNTR to OTR in "parallel"

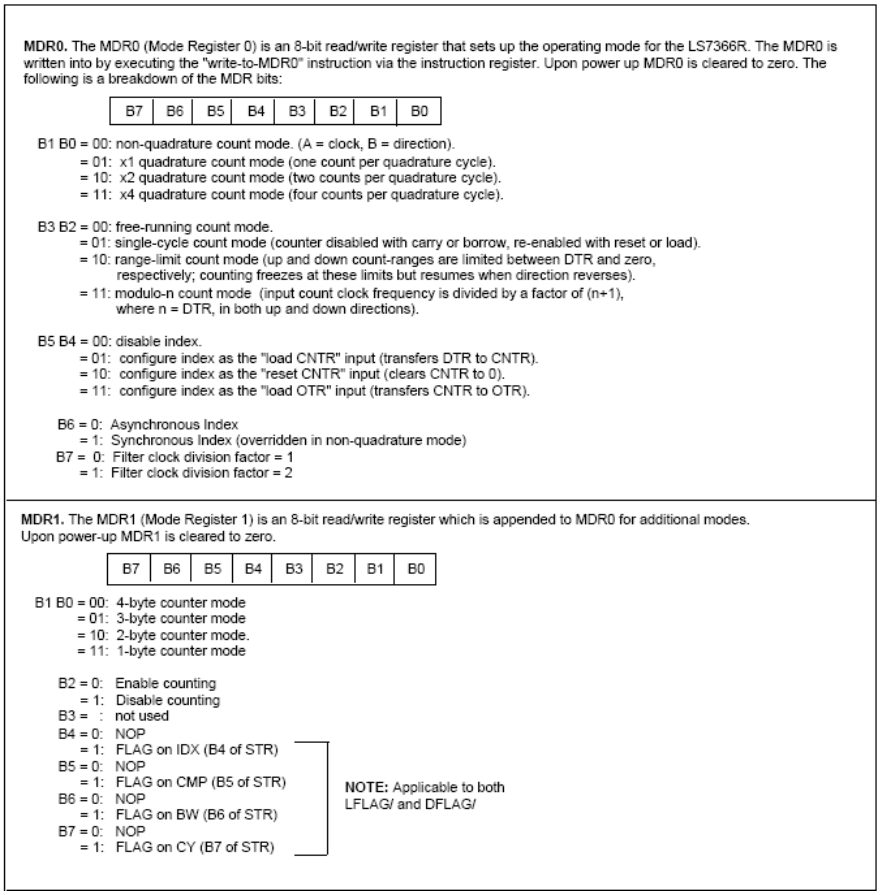
7366R-122205-3

Figure 7.3: How to form commands for the LS7366R Quadrature Encoder Counter. [17]

example the first required command is to write to the `MDR0` (Mode Register 0) register. This register is the first of two controlling how the encoder counter operates. To write to this register requires the write command (B7:6 = 10) and the `MDR0` register (B5:3 = 001) as seen to the right of the `IR` register description. For any command B2:0 can be anything. Here these bits will be 0. The command required to write to `MDR0` is therefore (bit 7 first) `0b10001000`. According to the table in the datasheet extract the write command can be followed by 1-4 bytes of data. In this case, there is only one additional byte: the value to be stored in the `MDR0` register, which will be determined shortly. The other two commands required by this program can be determined in the same way, which are the commands to write to the `MDR1` register (0b10010000), and to read the `CNTR` register (0b01100000). The `MDR1` register is the second mode register and requires again only a single byte of data. The `CNTR` register is the counter register. When reading from `CNTR`, that register is not actually what is transmitted. As is stated in the table in the excerpt, attempting to read `CNTR` actually copies the count into the `OTR` and transmits that. This copy prevents the counting and

transmitting operations from interfering with each other.

With the commands determined, the next step is to determine the data bytes to be written to `MDR0` and `MDR1`. Figure 7.4 contains an excerpt from the datasheet specifying what the various settings for these two registers are.



MDR0. The MDR0 (Mode Register 0) is an 8-bit read/write register that sets up the operating mode for the LS7366R. The MDR0 is written into by executing the "write-to-MDR0" instruction via the instruction register. Upon power up MDR0 is cleared to zero. The following is a breakdown of the MDR bits:

B7	B6	B5	B4	B3	B2	B1	B0

B1 B0 = 00: non-quadrature count mode. (A = clock, B = direction).
= 01: x1 quadrature count mode (one count per quadrature cycle).
= 10: x2 quadrature count mode (two counts per quadrature cycle).
= 11: x4 quadrature count mode (four counts per quadrature cycle).

B3 B2 = 00: free-running count mode.
= 01: single-cycle count mode (counter disabled with carry or borrow, re-enabled with reset or load).
= 10: range-limit count mode (up and down count-ranges are limited between DTR and zero, respectively; counting freezes at these limits but resumes when direction reverses).
= 11: modulo-n count mode (input count clock frequency is divided by a factor of (n+1), where n = DTR, in both up and down directions).

B5 B4 = 00: disable index.
= 01: configure index as the "load CNTR" input (transfers DTR to CNTR).
= 10: configure index as the "reset CNTR" input (clears CNTR to 0).
= 11: configure index as the "load OTR" input (transfers CNTR to OTR).

B6 = 0: Asynchronous Index
= 1: Synchronous Index (overridden in non-quadrature mode)
B7 = 0: Filter clock division factor = 1
= 1: Filter clock division factor = 2

MDR1. The MDR1 (Mode Register 1) is an 8-bit read/write register which is appended to MDR0 for additional modes. Upon power-up MDR1 is cleared to zero.

B7	B6	B5	B4	B3	B2	B1	B0

B1 B0 = 00: 4-byte counter mode
= 01: 3-byte counter mode
= 10: 2-byte counter mode.
= 11: 1-byte counter mode

B2 = 0: Enable counting
= 1: Disable counting
B3 = : not used
B4 = 0: NOP
= 1: FLAG on IDX (B4 of STR)
B5 = 0: NOP
= 1: FLAG on CMP (B5 of STR)
B6 = 0: NOP
= 1: FLAG on BW (B6 of STR)
B7 = 0: NOP
= 1: FLAG on CY (B7 of STR)

NOTE: Applicable to both LFLAG/ and DFLAG/

Figure 7.4: Bit field descriptions for `MDR0` and `MDR1`. [17]

`MDR0`

- B1:0: This controls how the counter handles the quadrature counts. In this example the highest precision will be used, which is the x4 count mode.
- B3:2: This bit field modifies or limits what is counted. Here the free-running mode will be used, which causes every tick to be counted. In the event of overflowing the counter it will continue counting and set a carry bit in another register.

- B5:4: These bits control the operation of an external pin called the index. In this case the pin is unused, so left disabled.
- B6: As the index pin is not being used here, this bit doesn't matter so is left as 0.
- B7: This relates to a frequency crystal external to the encoder counter. As such it is left as 0 in this example. See the data sheet for further information.

With this information, the data byte to be written to this register is `0b00000011`.

`MDR1`

- B1:0 This byte field controls how many bytes wide the counter is. This example uses a 4-byte counter.
- B2 This enables counting. It must be set to 1 before any encoder ticks will be counted.
- B3 Unused, so left as 0.
- B7:4 These bits enable external pins to be used as flags for various conditions. These pins, if connected to the microcontroller, can trigger interrupts when certain conditions are met on the encoder counter. These are all disabled in this example.

The only bit in this register that needs to be set is the counter enable, so the byte to be written to this register is `0b00000100`. At this point all the information required to create the program is gathered. Below is the code for this example.

Example code

```
#include <avr/io.h>
#include <util/delay.h>
#include "../libraries/reg_structs.h"

/**
 * \def SS Slave Select pin
 */
#define SS PORTBbits._P4

/**
 * \func SPI_init()
 * \brief initializes SPI for use with LS7366R
 */
void SPI_init() {
    SPIDDRbits._MOSI = 1;
    SPIDDRbits._MISO = 1;
    DDRBbits._P4 = 1;
    SPCRbits._DORD = 0; //MSB first
```

```
    SPCRbits._MSTR = 1; //Master mode
    SPCRbits._CPOL = 0; //clock low when idle
    SPCRbits._CPHA = 0; //sample on leading edge
    SPCRbits._SPR = 0;  //250ns pulse
    SPSRbits._w = 0;    //flags and double speed bit
    SPCRbits._SPE = 1;  //enable SPI
}
/**
 * \func SPI_xmit
 * \param data byte to xmit
 * \return received byte
 * \brief transmits data and returns received data
 */
char SPI_xmit(char data) {
    SPDRbits._w = data;
    while (!SPSRbits._SPIF) {} //wait till tx done
    return SPDRbits._w;     //return rx data
}

/**
 * \func quad_count_init
 * \brief initializes LS7366R quadrature counter
 */
void quad_count_init() {
    SS = 1; //set slave select
    //MDR0: 0b00 00 00 11: clock/1, async, index
    //    disabled, free-running count
    //    4 counts per quad cycle
    SPI_xmit(0b10001000);
    SPI_xmit(0b00000011);
    //MDR1: 0b0000x000: no flags, counting enabled,
    //    4-byte counter
    SPI_xmit(0b10010000);
    SPI_xmit(0b00000100);
    SS = 0; //clear slave select
}

int main(void) {
    SPI_init();
    quad_count_init();
    _delay_ms(1000);
    char data[4];
    int i = 0;
    SS = 1; // set slave select
    //load CNTR to OTR then read OTR:  0b01 100 xxx
    SPI_xmit(0b01100000);
    for(i=0; i<4; i++){
        data[i] = SPI_xmit(0); //read 4 bytes
    }
    SS = 0; //clear slave select
```

```
    //32 bit count of encoder ticks now stored in
    //4 bytes of data array
    return 0;
}
```

Example timing

In order to better understand what is happening during SPI communication, Figure 7.5 displays the states of all four SPI lines during a read operation of the CNTR register of the LS7366R Quadrature Counter. In the figure the CNTR register is setup to be 16 bits rather than the 32 used above. The sections on the `MISO` line marked tri-state are periods that the LS7366R is not actively controlling the line, even though the microcontroller may be sampling it. The sampled data for these times is garbage and is never read by the code. The section marked "Random Data" is a period that the microcontroller must be transmitting something to cause the clock to oscillate, but the LS7366R ignores the `MOSI` line during this time.

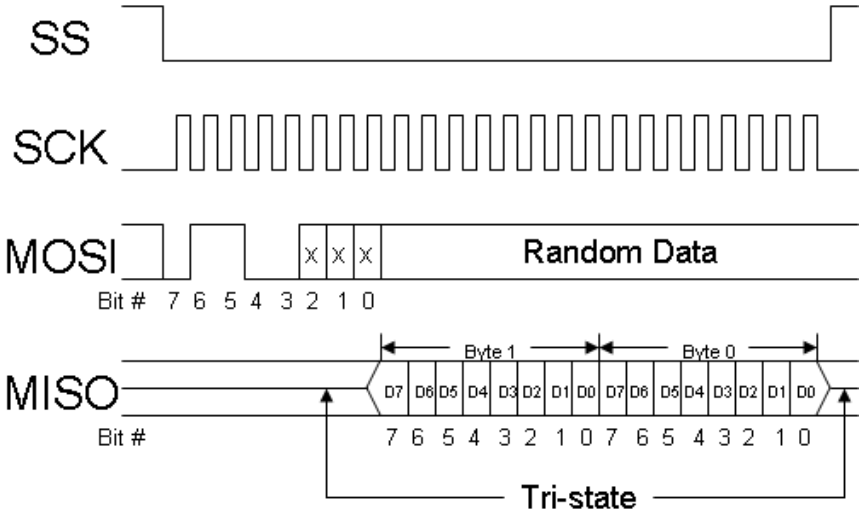


Figure 7.5: States of all four SPI lines during operation to read 16 bit CNTR register from LS7366R Quadrature Counter.

7.2.4 Daisy-Chaining SPI devices

Due to hardware or layout limitations, there are times that using a separate slave select line for each SPI device is not feasible. In this case it may be possible to daisy chain the devices together and use a single slave select line for all of them.

A daisy chain is formed by connecting the output of one device into the input of the next, thereby connecting the devices in series rather than in parallel (see

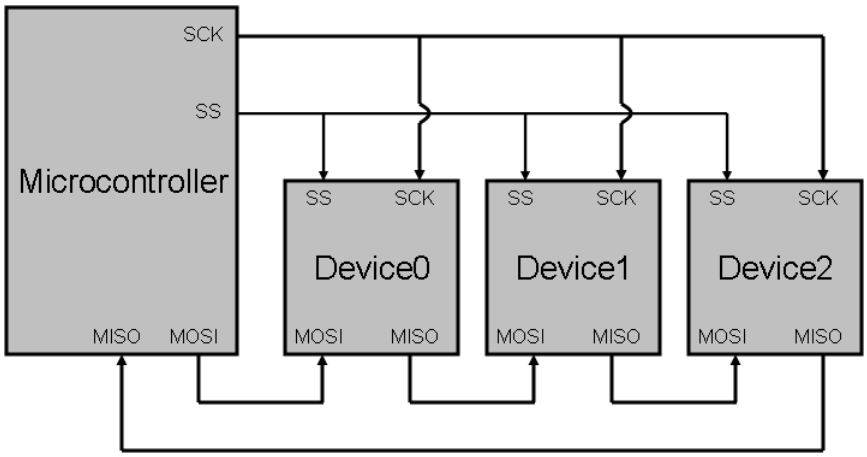


Figure 7.6: 3 devices daisy-chained to a microcontroller using SPI.

figure 7.6). Whenever the clock pulses, the registers in each of the devices shifts one, causing the highest bit of each device to shift into the lowest bit of the next device. Therefore, when sending a command to each of the devices, the command for the last device on the chain must be sent first. Once all the commands are shifted into the devices, the slave select line is driven high and all the devices read their instructions. Reading data back on the MISO line works similarly, in that the appropriate number of bytes must be clocked out to shift the results from all the devices back around to the microcontroller.

Limitations of a daisy-chain

There are several limitations which cause daisy chains to be less useful than they might otherwise be.

- All devices must be daisy-chain compatible. Not all devices that use SPI are set up to be able to daisy-chain. Any device that is not must be controlled using its own slave select line.
- All devices must use the same clock mode. If two devices use different clock operational modes (CPOL or CPHA are different), then they cannot be daisy-chained together as they will be operating out of sync with each other.
- All devices are controlled simultaneously. Whenever a command is sent to one device, all of them receive some form of command. If the programmer is not careful to send a sequence that is known to be ignored by other devices, unexpected behaviors may occur.

Over all, daisy-chains can be useful in certain circumstances, especially when all the devices being controlled are identical. They do suffer serious limitations

that reduce their usability, so take care when attempting to daisy-chain SPI devices.

7.3 Two-Wire Interface (TWI)

The final type of serial communications peripheral on the ATMega644p is the Two-Wire Interface serial. Other common names include TWI, I^2C (I-squared C), I2C (I-two C) or Inter-Integrated Circuit). This is again a master-slave configuration, though multiple masters can exist. With only two wires, one for data and one for the clock, it is only half-duplex meaning data can only be sent in one direction at any given time.

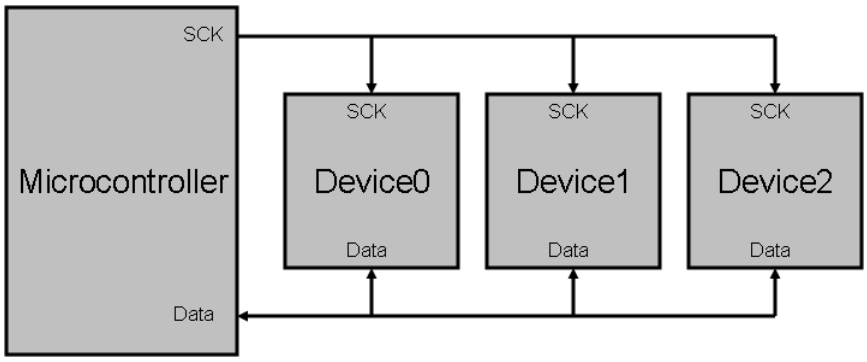


Figure 7.7: 3 devices connected to a microcontroller using TWI.

7.3.1 Operation

As TWI is not limited to two devices, and as there is no slave select line as there was for SPI, each device on the bus must have a unique address. When the master (or any of them in a multi-master setup) is ready to begin transmitting, it begins with a low start bit, followed by the 7-bit address of the slave to which the master is talking. The clock (SCL) then begins timing the transmission after the start bit has been set, and before the address. Unlike SPI, where the data was read and propagated on the rising and falling edges of the clock, in TWI SDA is set while the clock is low and read while the clock is high.

After the address is sent, the master indicates with a single bit whether it will be reading from (1) or writing to (0) the addressed slave. If there is a slave with that address on the bus, it will respond with a single ACK bit (logical low), and then the master or slave will transmit the data, depending on the read/write bit. If the master wants to transmit several bytes in succession to the same slave, it can do so, though the slave will send ACK bits between each byte sent. Likewise, if receiving multiple bytes the master must send ACK bits after every byte received except for the final byte. Finally, once all the data has

been transmitted the master can send a stop bit, indicating that it has finished, or another start bit indicating a new message to a new slave.

7.3.2 Example

While TWI is simpler in physical execution, it has the trade-off of being far more complex programmatically. What follows is a microcontroller retrieving a temperature reading from a TMP102 temperature sensor from Texas Instruments. To accomplish this task the microcontroller must write a single byte to the sensor instructing it which register to read from, followed by the microcontroller reading the two bytes of data from the controller. What complicates the matter is the number of times that the interrupt flag must be cleared before the peripheral can continue due to the number of ACKs and transmissions.

```
#include <avr/io.h>
#include "../old libraries/reg_structs.h"

/**
 * \def address Address of target sensor
 */
#define address 0b10010000

/**
 * \func TWI_config
 * \brief configure TWI peripheral
 */
void TWI_config() {
    TWBRbits._w = 4;    //bit rate setting
    TWSRbits._TWPS = 0; //bit rate setting
    TWCRbits._TWEA = 1; //enable ack
    TWCRbits._TWEN = 1; //enable
}

int main(void) {
  TWI_config();
  //default config is fine for sensor

  char highbyte;
  char lowbyte;

  // Set sensor's register address to data register
  TWCRbits._TWINT = 1;       //clear interrupt
  TWCRbits._TWSTA = 1;       //START condition
  while(!TWCRbits._TWINT) {} //wait until ready
  if (TWSRbits._TWS != 0x01) { //error checking
    return 0;
  }
  TWDRbits._w = address;     //SLA+Write
  TWCRbits._TWINT = 1;       //transmit address
```

```
  while(!TWCRbits._TWINT) {} //wait until ready
  if (TWSRbits._TWS != 5){    //error checking
    return 0;
  }
  TWDRbits._w = 0;   //set sensor register address
  TWCRbits._TWINT = 1;       //transmit data
  while(!TWCRbits._TWINT) {} //wait until ready

  // Read sensor's data register (2 bytes)
  TWCRbits._TWINT = 1;       //clear interrupt
  TWCRbits._TWSTA = 1;       //START condition
  while(!TWCRbits._TWINT) {} //wait until ready
  if (TWSRbits._TWS != 0x02) { //error checking
    return 0;
  }
  TWDRbits._w = address+1;   //SLA+Read
  TWCRbits._TWINT = 1;       //transmit address
  while(!TWCRbits._TWINT) {} //wait until ready
  if (TWSRbits._TWS != 0x09) { //error checking
    return 0;
  }
  TWCRbits._TWINT = 1;       //receive highbyte
  while(!TWCRbits._TWINT) {} //wait until ready
  highbyte = TWDRbits._w;
  TWCRbits._TWINT = 1;       //receive lowbyte
  while(!TWCRbits._TWINT) {} //wait until ready
  lowbyte = TWDRbits._w;

  //generate STOP
  TWCRbits._TWSTO = 1;
  TWCRbits._TWINT = 1;       //send stop condition

  //highbyte and lowbyte now available for use
  return highbyte*256+lowbyte;
}
```

7.4 Comparing TWI and SPI

Many devices will support both TWI and SPI. In such a case, one must examine the needs versus the capabilities of each.

Real-Estate

In terms of pin-outs required, and therefore wires or traces on a PCB, TWI is far superior as it only requires two lines no matter how many devices are on the board. The cost benefit of the reduction was one of the driving forces for the development of TWI for use in small electronics.

Speed

SPI has a major benefit here as it is full-duplex, meaning twice the data can flow over the lines. This is not fully utilized as it is uncommon for master and slave to be transmitting data simultaneously. Another major speed advantage that SPI has is the lack of addresses. For any message this increases the overhead, thereby decreasing the available time to transmit data. This can get especially bad when sending many short messages.

Flexibility

Here each has benefits over the other. SPI has more configurable options, however this can often lead to confusion and problems. On the other hand it is somewhat easier to use once set up. TWI is more consistent from device to device as to levels and clock phase and also has the ability to hot-swap devices on and off the bus, however does have a severe drawback. Slave devices, such as sensors, DACs and ADCs, can run into problems with overlapping addresses if several identical devices are used on the same bus. Careful attention must be payed while designing the system to how many are possible on a single bus and how to set their addresses.

Complexity

Again, SPI takes the lead here. While TWI has less physical complexity, the added complexity in the code is vast and often more than makes up for the extra pair of wires required. SPI also does not require the step of ensuring every device has a separate address.

Overall, when given choice between TWI and SPI, and the requirements of the system don't indicate one choice is particularly better, the best choice is most likely the one that is more comfortable for the developers to reduce the likely hood of errors and the costly debugging process.

7.5 Conclusion

There are several additional serial communications protocols, however these are the three (RS232, SPI and TWI) that are primarily found on microcontrollers. More information is available about the history and development of these standards by searching on-line, as well as the standards themselves for interested parties. The information contained herein should be enough to enable their use. The primary source to learn more about the operation of these three peripherals is the datasheet. Chapters 15 through 18 cover all of the serial communications for the ATmega644p [11].

Chapter 8

Interrupts and Timers

Interrupts and accurate timing are vitally important to the smooth operation of a microcontroller. Without the first, the controller would spend most of its time checking to see if events occurred or face the possibility of missing vital signals. Without the second, there would be no method of controlling when tasks occurred, which would cause error in measurements, skewing results and rendering it impossible to control the system properly.

8.1 Interrupts

An interrupt is a notification to the processor that some event has occurred and requires attention, analogous to a phone call during work. Interrupts can be triggered by a number of events including incoming communications, errors occurring in hardware, notification that a task has been completed (such as transmission or an ADC conversion), values of input pins changing or timers, just to name a few. One type of interrupt most people encounter on a regular basis is the interrupt that occurs on a computer every time a key on the keyboard is pressed. Most of the time the keystrokes are buffered and displayed later, without the interrupt if the computer wasn't actively paying attention keystrokes would be missed. These interrupts are vital to the smooth and correct operation of many systems.

8.1.1 How interrupts work

When an interrupt is triggered normal program operation ceases after the currently executing operation finishes. A snap-shot of the current status of the program, including variables and the program counter, is stored as if the program was entering a function. The processor then checks a special block of memory that contains addresses for all of the interrupt service routines (ISRs). These are called the interrupt vectors and are special functions that are called if and when an interrupt is triggered. If an ISR doesn't exist for a given interrupt

the address will be left at zero. It is the programmers responsibility to ensure that there are ISRs available for every enabled interrupt.

In general, there are four parts to interrupts on microcontrollers. The first is the interrupt flag. It is normally a single bit located in one of the SFRs for the peripheral it pertains to, and indicates whether the interrupt event has occurred. When the interrupt event happens the flag is set (to logical 1). This occurs whether or not interrupts are actually enabled, and it is cleared when the interrupt is serviced. Alternatively, the flag can be cleared via software or in certain cases through additional events.

The second part is the interrupt enable bit. This is again a single bit located in one of the appropriate SFRs and tells the hardware whether or not this specific interrupt is to be serviced automatically. If it is not set high then when the triggering event occurs, even though the interrupt flag gets set, no interrupt occurs. At this point it becomes the software's responsibility to check the flag every so often or completely ignore it if the event is of no importance to the program.

Next is the global interrupt enable. Aptly named, the global interrupt enable allows interrupts to occur. If this is off (logical 0) then no interrupts will be triggered, though the flags will still be set. It is common practice to clear this bit during ISRs to prevent interrupts from interrupting the operation of another ISR, which can lead to odd or unexpected behavior.

Finally there is the interrupt service routine mentioned above. This is a set of instructions, much like a function though without the ability to accept arguments or return data, that is run when its corresponding event occurs and both enable bits are set. It is common practice to keep ISRs as short as possible, often simply storing relevant information in memory, for later examination by the program. This has the dual effect of keeping the program running as smoothly as possible with minimal delay and missing fewer interrupts if they occur in close proximity. If a program is stuck in an ISR for an extended period of time other, perhaps more important, events can be missed, causing a loss of data or incorrect data to be stored.

Entering an ISR is much like an ordinary function call, with the exception that the function being interrupted is not expecting it. This places the entire burden of saving and restoring the program context on the ISR. It, therefore, must handle pushing any register it will overwrite onto the stack and popping the data back to its original register when it finishes. Certain other things, such as status registers (`SREG` on the ATmega644p) may also need to be saved. These registers may contain required data regarding the results of previous operations such as compares, carries and overflows. If the status register is not saved, and it is overwritten by the ISR, the program may operate differently from what it is supposed to upon returning from the ISR.

8.1.2 Using Interrupts

Interrupts can be tricky to set up the first time or two as they are very processor specific. This is due to not only needing to know their names, but also knowing

the syntax for writing an ISR for that microcontroller. This section pertains almost exclusively to the ATMega644p. Other AVR microcontrollers use similar syntax, but microcontrollers produced by other companies use completely different methods of defining an ISR.

The first step is to decide if the interrupt is necessary. If the only requirement is to let the controller know that the event has happened at some point in the past (such as a digital input pin has changed value), then it is perhaps not necessary and the program can simply check the interrupt flag every so often. If, on the other hand, it is time critical such as data arriving via the USART and it needs to be read before the next byte arrives then an interrupt is likely warranted. As this is a commonly used interrupt it will be the object of this example.

Now that the interrupt has been determined to be necessary, it must be enabled. In this case the interrupt in question is the RX Complete Interrupt and the interrupt enable bit is `RXCIEn` in the register `UCSRnB` (USART Control and Status Register n B).

```
UCSRnBbits._UCSRnB = 1;
```

Assuming the rest of the USART has been correctly setup, the global interrupt enable flag must be set. This is the `I` bit in the `SREG` register. This status register is for the microcontroller in general and contains several bits related to the arithmetic logic unit (ALU). In this case Atmel, the producers of this microcontroller, have provided easy methods of enabling and disabling this bit. The function `sei()` enables global interrupts and `cli()` disables them. When using these functions the instruction immediately following `sei()` will execute before any interrupts actually trigger, though `cli()` disables them immediately, and will not even allow an interrupt occurring simultaneously to trigger its ISR.

Finally it is time for the ISR itself. This is a function definition so it appears outside the definition for any other function, although it's syntax differs from ordinary C functions. Furthermore, unlike standard functions, it does not require a function prototype, nor does it matter where in the file structure it is located.

```
ISR(USART0_RX_vect) {
  \*  code goes here  i.e...*\
  receivedData[nextOpenSpot] = UDR0bits._w;
}
```

The names of the interrupt vectors, in this case `USART0_RX_vect`, can be determined by looking at Table 9-1 Reset and Interrupt Vectors on page 61 of the datasheet [11], finding the appropriate vector and adding `_vect` to the end of the Source.

8.1.3 The Interrupt Process

When an interrupt is actually received, there are several steps the processor must take to ensure the rest of the program and any stored data is not lost

due to the interrupt. Below is an example of an ISR which acts as an adder, summing the values of the PINA and PINB registers, and using the result to set PORTC. Immediately below the ISR is the assembly code, which will be broken down into segments and explained along with the steps the processor must take. Assembly is an intermediate step between a programming language such as C and the machine code that is written to the microcontroller. In assembly, each line is an individual instruction stored in the program memory and executed in one or more cycles. The first column is the memory address of the instruction, and the following hexadecimal value is the instruction stored there. To the right of that are the human readable opcodes and arguments that correspond to the hexadecimal instructions. Descriptions of all the opcodes for the ATmega644p are in chapter 28 of the datasheet [11]. Different processors may have different instruction sets, so what works for one processor or microcontroller likely will not work for a different type.

```
ISR(TIMER0_COMPA_vect) {
  //load arguments (PINA,B)
  char i = PINA;
  char j = PINB;
  //sum the two and output
  PORTC = i+j;
}
```

```
000000a4 <__vector_16>:
  a4: 1f 92     push r1       ; This register not actually needed
  a6: 0f 92     push r0       ; Used in saving status register
  a8: 0f b6     in r0, 0x3f   ; 63 (Status register SREG)
  aa: 0f 92     push r0       ; push SREG
  ac: 11 24     eor r1, r1    ; clears register
  ae: 8f 93     push r24      ; store registers used in ISR
  b0: 9f 93     push r25
  b2: 90 b1     in r25, 0x00  ; 0  Load PINA (address 0)
  b4: 83 b1     in r24, 0x03  ; 3  Load PINB
  b6: 89 0f     add r24, r25  ; Sum PINA, PORTB.  Save to r24
  b8: 88 b9     out 0x08, r24 ; 8 Set PORTC
  ba: 9f 91     pop r25       ; restore registers used in ISR
  bc: 8f 91     pop r24
  be: 0f 90     pop r0        ; pop old SREG value
  c0: 0f be     out 0x3f, r0  ; 63 restore SREG
  c2: 0f 90     pop r0        ; pop old values
  c4: 1f 90     pop r1
  c6: 18 95     reti          ; return to previous function
```

1. The first step of the interrupt process is the actual receipt of the interrupt signal. If the individual interrupt is enabled, and interrupts are enabled globally by the I bit of the SREG register, the process continues after the currently executing instruction completes. If either are disabled than the flag is set but no interrupt service routine is called.

2. The global interrupt enable is cleared to prevent further interrupts from occurring while one is being serviced.

3. The instruction pointer is pushed to the stack. This is handled by hardware and does not appear anywhere in the program. The instruction pointer must be saved in order to restore the system to the next instruction that would have executed had the interrupt not occurred.

4. The program counter is set to the beginning of the ISR. This is also handled by hardware, however the addresses of all ISRs are stored in the program memory. If the requisite ISR does not exist, the software resets and begins again from the start of the `main` function.

5. The previous program context is saved. This involves `push`ing the values of any registers that will be used during the course of the ISR onto the stack for later retrieval. If this is not done, then when the ISR exits some of the data required by the main program may have changed and can cause problems to occur. Furthermore, the current value of the status register `SREG` must be saved. This register contains carry, overflow and other ALU (arithmetic logic unit) result flags. Failing to save this information can result in errors in the main program. This process is handled by the instructions in the memory block `0xa4` through `0xb0` in the above example.

```
a4: 1f 92     push r1       ; This register not actually needed
a6: 0f 92     push r0       ; Used in saving status register
a8: 0f b6     in r0, 0x3f   ; 63 (Status register SREG)
aa: 0f 92     push r0       ; push SREG
ac: 11 24     eor r1, r1    ; clears register
ae: 8f 93     push r24      ; store registers used in ISR
b0: 9f 93     push r25
```

6. Finally the actual code in the ISR is executed. In this example it is the reading of `PINA` and `PINB`, the addition of the same and saving the result to `PORTC`. These registers are accessed using their memory addresses: `0x00`, `0x03` and `0x08` respectively.

```
b2: 90 b1     in r25, 0x00  ; 0  Load PINA (address 0)
b4: 83 b1     in r24, 0x03  ; 3  Load PINB
b6: 89 0f     add r24, r25  ; Sum PINA, PORTB.  Save to r24
b8: 88 b9     out 0x08, r24 ; 8 Set PORTC
```

7. After the ISR has completed, the previously stored context of the main function must be restored by `pop`ing the previously stored values off the stack and returning them to their previous registers. This is step 4 in reverse, including restoring `SREG` to its previous state.

```
ba: 9f 91     pop r25       ; restore registers used in ISR
bc: 8f 91     pop r24
```

```
be: 0f 90     pop r0        ; pop old SREG value
c0: 0f be     out 0x3f, r0  ; 63 restore SREG
c2: 0f 90     pop r0        ; pop old values
c4: 1f 90     pop r1
```

8. Finally the ISR returns to the main function. The previously stored instruction pointer value is reloaded, global interrupts are re-enabled and the main function picks up where it left off, unaware of the interruption. This is all handled by hardware when the `reti` instruction occurs.

```
c6: 18 95     reti          ; return to previous function
```

8.2 Program startup assembly

While on the topic of assembly code, this is a good point for a tangent into what the compiler puts into the code on its own. The very first instruction, placed at address 0 and preceding even the interrupt vector table is a jump instruction to a block of instructions immediately following the interrupt vector table. This instruction block performs basic start-up tasks such as clearing the status register and defining the stack location. The exact contents vary with compiler and processor, however this is fully automated and not something the programmer need worry about unless programming directly in assembly. The last two instructions in this block are to call the `main` function, and then complete any on-exit tasks upon `main`'s return.

```
00000000 <__vectors>:
   0: 0c 94 3e 00  jmp 0x7c      ;  jump to 0x7c <__ctors_end>
; Interrupt vector table goes here

0000007c <__ctors_end>:
  7c: 11 24        eor r1, r1
  7e: 1f be        out 0x3f, r1  ; 63 Clear SREG
  80: cf ef        ldi r28, 0xFF ; 255 Load stack pointer value
  82: d0 e1        ldi r29, 0x10 ; 16
  84: de bf        out 0x3e, r29 ; 62 set stack pointer to 0x10FF
  86: cd bf        out 0x3d, r28 ; 61
  88: 0e 94 4a 00  call 0x94     ; 0x94 <main>
  8c: 0c 94 85 00  jmp 0x10a     ; 0x10a <_exit>
```

The interrupt vector table consists of a jump instruction for each type of interrupt. If an ISR exists for that interrupt the jump command points to that location, else it points to a specific point as determined by the compiler which handles the bad interrupt. Generally this causes the program to reset back to instruction 0. This can be avoided by ensuring that each enabled interrupt has a corresponding ISR.

```
  40: 0c 94 be 00  jmp 0x17c ; 0x17c <__vector_16>
  44: 0c 94 53 00  jmp 0xa6  ; 0x90 <__bad_interrupt>

00000090 <__bad_interrupt>:
  90: 0c 94 00 00  jmp 0     ; 0x0 <__vectors>
```

8.3 Timing

The ability to keep accurate time can be of vital importance, especially when performing tasks such as deductive reckoning navigation when there is no feedback. The examples given herein will all pertain directly to the timing of loops, as this is where it is most important. Furthermore, these methods can all be used in most any situation.

8.3.1 Basic Methods

The simplest, and least accurate, method for making an event happen at regular intervals is to include a the task and a wait command inside a loop; for or while depending on the needs of the situation. There are two functions that are made available for the ATMega644p, `_delay_ms()` and `_delay_us()` each of which accept a `double` as an argument and delay that many milliseconds or microseconds respectively. Most microcontrollers should have similar functions available in the pre-compiled libraries.

Straight Delay

```
int main(void) {
  DDRA = 0x01;
  while(1) {
    if (PORTA && 0x01 == 1) {
      PORTA = 0;
    } else {
      PORTA = 1;
    }
    _delay_us(1000);
  }
}
```

This is an example that is attempting to turn an LED on or off every millisecond, resulting in a flash rate of 500Hz. The actual rate will be slightly slower, however, as the delay does not take into account the time spent on other instructions in the loop or the loop itself, so the entire loop takes slightly longer than one millisecond, nor is the actual frequency known. Additionally, if there are any interrupts that can occur, individual passes through the loop can be significantly longer.

Calculated Delay

In an effort to get the loop closer to the desired duration, it is possible to attempt to calculate how long the task and looping take. One method is to run the task several times and use some form of stopwatch such as the timer peripheral to time the task. This however is fairly complicated as it requires running the task by itself on hardware. Some IDEs may also provide methods of counting the number of clock cycles required. It is also possible to do so manually, which requires some knowledge of assembly.

```
00000094 <main>:
  94: 81 e0        ldi r24, 0x01  ; 1
  96: 81 b9        out 0x01, r24  ; 1
  98: 41 e0        ldi r20, 0x01  ; 1
  9a: 20 ed        ldi r18, 0xD0  ; 208
  9c: 37 e0        ldi r19, 0x07  ; 7
  9e: 80 b1        in r24, 0x00   ; 0
  a0: 88 23        and r24, r24
  a2: 11 f0        breq .+4       ; 0xa8 <main+0x14>
  a4: 12 b8        out 0x02, r1   ; 2
  a6: 01 c0        rjmp .+2       ; 0xaa <main+0x16>
  a8: 42 b9        out 0x02, r20  ; 2
  aa: c9 01        movw r24, r18
  ac: 01 97        sbiw r24, 0x01 ; 1
  ae: f1 f7        brne .-4       ; 0xac <main+0x18>
  b0: f6 cf        rjmp .-20      ; 0x9e <main+0xa>
```

This is assembly code for the example above. While fully understanding what is occurring here requires sitting down with the datasheet to understand what each of the instructions (opcodes) are (chapter 28 of the ATMega644p datasheet [11]), here is a brief description. Up through address `9c` is setting up the loop and `DDRA`, and is only ever run once. Addresses `9e` through `a8` as well as `b0` are related to the loop, conditional and toggling the LED on or off. Finally addresses `aa` through `ae` are the delay function.

Knowing this, and with datasheet in hand, it is a straightforward process to determine the number of clock cycles required to run the task through each of the conditions and not count the instructions relating to the delay. For this task the the loop and toggle require 8 clock cycles if the conditional is true and only 7 if it is false. This difference can be rectified by manually adding a `NOP` (no op) operation in the else statement using the `__asm__()` function used to manually state memory addresses as mentioned in chapter 5.

```
} else {
  PORTA = 1;
  __asm__("nop");
}
```

This leaves both conditions at eight cycles which, if the microcontroller is running at 8MHz, is one microsecond. Adjusting the delay function results in the following.

```
int main(void) {
  DDRA = 0x01;
  while(1) {
    if (PORTA && 0x01 == 1) {
      PORTA = 0;
    } else {
      __asm__("nop");
      PORTA = 1;
    }
    _delay_us(999);
  }
}
```

```
00000094 <main>:
  94: 81 e0        ldi r24, 0x01  ; 1
  96: 81 b9        out 0x01, r24  ; 1
  98: 41 e0        ldi r20, 0x01  ; 1
  9a: 2e ec        ldi r18, 0xCE  ; 206
  9c: 37 e0        ldi r19, 0x07  ; 7
  9e: 80 b1        in r24, 0x00   ; 0
  a0: 88 23        and r24, r24
  a2: 11 f0        breq .+4       ; 0xa8 <main+0x14>
  a4: 12 b8        out 0x02, r1   ; 2
  a6: 02 c0        rjmp .+4       ; 0xac <main+0x18>
  a8: 00 00        nop
  aa: 42 b9        out 0x02, r20  ; 2
  ac: c9 01        movw r24, r18
  ae: 01 97        sbiw r24, 0x01 ; 1
  b0: f1 f7        brne .-4       ; 0xae <main+0x1a>
  b2: f5 cf        rjmp .-22      ; 0x9e <main+0xa>
```

Having already expended this much effort, it is worth checking the cycle count to ensure the desired length was reached, which does require examining the delay loop. This delay function works by loading a two-byte integer into a register and repeatedly subtracting one. The repeated subtraction requires four clock cycles each time (`SBIW` and `BRNE` operations) except for the final time when the `BRNE` operation requires a single clock cycle. Including the single cycle required for `MOVW`, the entire loop therefore takes exactly 8000 cycles (eq.8.1), which is exactly what was desired.

$$loopLength + 4 * (delayInt - 1) + 3 + 1 =$$
$$8 + 4 * (7 * 256 + 205) + 3 = 8002$$

All said and done, while it is nice to know that the loop is precise down to the uncontrollable jitter of the clock, this was a lot of work which must be redone any time there is a change in the system clock, task, or even sometimes compiler options. Additionally, this still suffers from the effects of interrupts if they are used, so this is not the best system to use either.

8.3.2 Using the Timers

Most, if not all, microcontrollers contain some form of timer peripheral specifically for this sort of task. In fact the only reasons not to use one for this sort of task is either because all the timers on the microcontroller in question are already in use for tasks at different frequencies, and therefore cannot be used to handle both, in which case it is likely time to reassess what microcontroller to use, or because the developer is new and doesn't know how to set it up properly.

The first method is to run a timer and continually check what the value in the counter register is. When the counter matches or exceeds the desired value, reset the counter and perform the task. This does mostly solve the issue faced before of interference from interrupts, poorly timed interrupts can still cause problems, and it is still prone to drift as there is no correction for when the counter exceeds the desired value. This can be fixed by subtracting the desired value from the current value, thereby saving the overrun and turning the drift into a jitter. This method of timing may be best suited to tasks where accuracy is not vital, all timers are in use by other tasks, and there are few other things happening. In this case, if there is a system clock available that simply counts the time from power on (which does not exist on the ATMega644p), or a timer in use with a significant longer duration than is being timed, a modification can be made to this method to change the target value rather than the counter value, allowing this task to piggy back on another clock. Such an event is unlikely enough, and has enough variables that no example is given here.

In most cases, the correct way to do this is to use the timer interrupts. These allow for timings accurate to the prescaler value set in the configuration, which divides the system clock to lower frequencies to allow longer times than 256 clock cycles (or 65536 for a 16-bit timer). Furthermore, while it can still be delayed by other interrupts, such a delay will only affect the one run of the task and subsequent interrupts will occur on schedule.

The ATMega644p has three timer/counters; two 8-bit and one 16-bit. Each one has slightly different options and capabilities, including, for Timer/Counter2, asynchronous operation using an external signal which allows counting external pulses. In this example Timer/Counter0 will be used as it meets all the requirements and has less unused capability. The peripherals registers consist of the following:

- `TCCR0A` and `TCCR0B`: These are two control registers that control the behavior of the timer/counter, including prescaler, when to reset the counter register, and whether the clock toggles external pins to generate PWM signals.

- `TCNT0`: This is the heart of the timer/counter and contains the current count. It can be both read from and written to, however writing to this register while the counter is running causes the next compare match check not to happen. When running in certain modes this can cause an interrupt to be skipped.
- `OCR0A` and `OCR0B`: These are the compare to registers. Every time the `TCNT0` register is incremented, it is compared to both of these registers, which may cause interrupts or the `TCNT0` register to reset depending on the settings in the control registers.
- `TIMSK0`: The Timer Interrupt Mask register contains all the interrupt enable bits for the timer/counter, and have corresponding flags in the Timer Interrupt Flag register.
- `TIFR0`: This is the aforementioned Timer Interrupt Flag register. The three bits used in this register, each of which have interrupt enable bits in `TIMSK0` and accompanying interrupt vectors, are set when `TCNT0` matches `OCR0A`, `OCR0B` or when the timer/counter overflows.

In this example, as interrupts at a specific frequency are required, it makes the most sense to trigger an interrupt and reset the counter when it reaches a specific value. This means using the Clear Timer on Compare Match (CTC) mode of operation and compare register A. As for the clock prescaler, of the options available a prescaler of 64 is best to achieve a 1kHz interrupt frequency on an 8MHz clock and an 8-bit counter. By dividing the prescaler into the system clock frequency, the compare register value can be found to be 125. Finally, the requisite compare match interrupt must be enabled.

The interrupt vector name can be found in the place mentioned before in chapter 9 of the datasheet. In this case the appropriate vector is `Timer0_COMPA_vect` into which the LED toggling code is placed. One last thing is required for this to work properly, and that is an infinite loop of doing nothing after the timer is setup. While this loop can be used for various tasks, the timer/counter will only run as long as there is code executing, which means if the main function ever exits, the interrupts and therefore the LED toggling will also stop.

```
int main(void) {
  DDRAbits._P1 = 1;
  TCCR0Abits._WGM = 2;  //enable CTC mode
  TCCR0Bbits._CS = 3;   //set clock w/ prescale of 8
  TIMSK0bits._OCIE0A = 1;
  OCR0Abits._w = 125;
  sei();            //global interrupt enable
  while(1) {}
}

ISR(TIMER0_COMPA_vect) {
  PINAbits._P1 = 1;     //writing PIN toggles PORT
}
```

There is one final way in which this can be simplified even further. As was previously mentioned, the timer/counters are attached to external pins which can be used to run PWM signals. This is, an essence, a PWM signal with a 50% duty cycle (for more on PWM signals see chapter 9). This will allow the timer to take care of the task in its entirety, and will remove the necessity of any interrupts. This setting appears in the `TCCR0A` register and is called the Compare Match Output A Mode, which needs to be set to toggling the corresponding output compare pin (`OC0A`) which is also pin `PB3`, as opposed to pin `PA0` which has been used previously. To set up the timer/counter to do this requires the following code, and does not require, or enable any interrupts, though it does require the infinite loop.

```
int main(void) {
  DDRAbits._P1 = 1;
  TCCR0Abits._WGM = 2;  //enable CTC mode
  TCCR0Abits._COM0A = 1;         //OC0A toggle on compare
  TCCR0Bbits._CS = 3;   //set clock w/ prescale of 8
  OCR0Abits._w = 125;
  while(1) {}
}
```

8.4 Conclusion

The timers built into microcontrollers are designed to give a wide variety of functions, and automate most common processes. To this end the delay functions should only be necissary on the rare occasion where a specific delay is required, rather than a specific timing. Use the timers for any repeating task.

To find out more about the capabilites and usage of timers, read the data sheet for the microcontroller in question [11]. Some microcontrollers have different capabilities for each of their timers, so read all of the timer information, and don't assume that the others are like the first.

Chapter 9

External Devices

Thus far the focus has been on the peripherals internal to a microcontroller. This is all well and good, but microcontrollers alone, without external devices, are somewhat less than useful. This chapter will cover several types of external devices including motors, servos, and a variety of sensors.

9.1 Motors

Motors are one of the standards when it comes to controlling motion, and the options available reflect this. There are so many options that entire books can be written about the differences between them. This, therefore, shall limit itself to the control of a DC motor. Even the voltage doesn't matter significantly, as it will only change the specific devices used in the control interface and not the type of device used.

9.1.1 Linear Drivers

There are two primary methods of supplying and controlling power to a DC motor; a variable DC voltage or a continuous series of pulses at the maximum voltage of varying voltage. This section covers the former.

The majority of DC motors can be powered by voltage levels up to a certain level; 12V or 24V are common values among hobby and household motors. With no load on the motor's output shaft, and this maximum voltage applied, the motor will spin at it's maximum rated speed, measured in rotations per minute (RPM), or if its held stationary it will provide its rated stall torque. If, however, some lower voltage is applied then the maximum speed and stall torque will reduce linearly reaching 0 rpm and no torque at near 0 volts. In practice small voltages can be supplied to motors without any motion by the motor due to the torque being insufficient to overcome internal friction.

This then becomes the challenge: to design a method of providing a variable DC voltage from 0V up to the maximum the motor can take with sufficient

current that nothing is destroyed. Recalling back to the voltage and current capabilities of microchips in general, and specifically the ATMega644p as covered in Chapter 3, microcontrollers generally cannot supply either the voltage or the current required. Nor, when push comes to shove, can many even provided the variable voltage due to not having a digital-analog-converter (DAC) built into it.

Assuming the desired motor and microcontroller have already been chosen, and assuming the microcontroller does not have a DAC peripheral, an external DAC needs to be chosen. There are three points of concern when choosing the DAC. The first is how it interacts with the microcontroller. This involves communications protocol (generally SPI or TWI/I^2C) and signal triggering levels (microcontrollers running at 3.3V may not have the output voltage for the DAC to recognize high bits). The second point of concern is the resolution of the DAC. The number of bits in the resolution determines how many individual voltage values the DAC can output, and therefore also the distance between each value. To find the resolution of a DAC in volts, divide the maximum output range by 2 raised to power equal to the number of bits minus 1. For example, on a 0-5V DAC with 12-bit resolution, the voltage resolution would be:

$$V_{res} = \frac{V_{DACmax} - V_{DACmin}}{2^{DAC_{res}} - 1}$$
$$V_{res} = \frac{5V - 0V}{2^{12} - 1}$$
$$V_{res} = \frac{5V}{4095}$$
$$V_{res} = 1.22mV$$

The final point of concern is the output range. While this can affect the complexity of the calculations required for the remainder of the motor driver, any range can be made to work without great effort. The output range merely affects the range of the amplifier necessary to reach the required voltage and amperage.

The only other major component in a linear motor driver is an operational amplifier, commonly called an op-amp. There is not time here to explain the workings or the myriad ways in which to use them, rather only the specific configuration used herein will be discussed. There are numerous sources available for additional information on op-amps. Wikipedia has a long article (Operational amplifier) [7] on the history and operation of an op-amp, as well as another article (Operational amplifier applications) [8] on various applications and configurations. Another site, www.rfcafe.com, also has a brief description and several of the configurations [9].

The capabilities of the op-amp for this circuit must match or exceed the requirements of the motor, both in terms of voltage and current. Furthermore, in order to run the motor in both directions, the op-amp must be able to handle the motor's voltage both positive and negative.

Once the primary components (microcontroller, DAC, op-amp and motor) are chosen, the process of combining them can begin. Several resistors will also be required, though there values are not known at this point.

Example

For this example, the components to be used will be the ATmega644p, a DAC7554 12-bit DAC from Texas Instruments, an OPA548T operation amplifier from Texas Instruments and a Faulhaber 1524T012SR 12V DC motor. The DAC can be run via SPI and can operate at the same 5V as the ATmega644p. Its output is a 0V to 5V signal with a maximum current of 50mA. This then needs to be converted into a -12V to 12V signal by the op-amp. While there are a few possible ways to configure a conversion between the two voltage ranges, a shift and a gain are required. As it is easier to produce the shift before the gain in a circuit, the conversion should look like this:

$$V_o = gain * (V_i - shift)$$
$$V_o = \frac{12V}{2.5V} * (V_i - 2.5V)$$
$$V_o = 4.8 * (V_i - 2.5V)$$

Figure 9.1 displays a differential amplifier which can be used to produce this equation. The full form of the output equation for this configuration is

$$V_o = \frac{1 + \frac{R_2}{R_1}}{1 + \frac{R_3}{R_4}} * (V_2 - V_3) - \frac{R_2}{R_1} * V_1$$

This design is commonly used with V_3 connected to ground (0V), however the calculations are much easier in this case if V_3 is connected to a 2.5V source while V_1 is connected to ground. All that remains is to find resistor values for R_1 through R_4 that cause the gain term to be as close as possible to 4.8. If the gain is slightly large it means that the maximum voltage may be reached prior to the highest value of the DAC. As long as the supply voltages for the op-amp are limited to the $\pm$12V that is the limit for the motor, nothing will be damaged.

As there are innumerable ways to reach a desired combination with resistors, there is no single right answer. Instead its best to decide on a couple design goals and work towards them. In general the first goal should be fewest number of resisters possible, in this case 4, and the second should be to limit the number of different values. Looking more closely at the equation, it can be found that if R_3 and R_4 are the same value, then the denominator equals two and the numerator can be designed for a value of 9.6, or $\frac{R_2}{R_1} = 8.6$. One of these two values is arbitrary, so let R_1 be a 10kΩ resistor. R_2 could be either an 82kΩ or a 91kΩ resistor. The former would not allow the full 12V to reach the motor, so

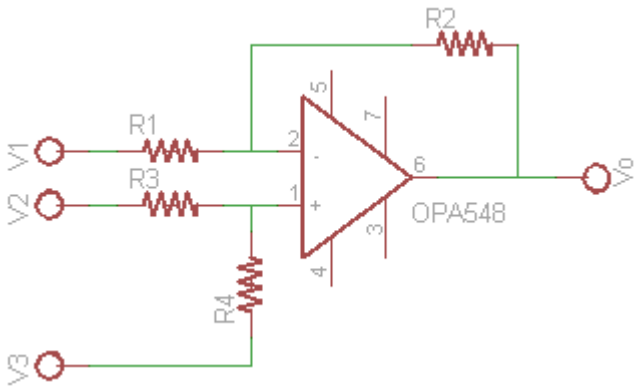


Figure 9.1: Differential op amp.

the later is the better choice if this is an issue. Finally, to keep the number of different components low, R_1 and R_2 can both be declared to be 10kΩ resistors as well.

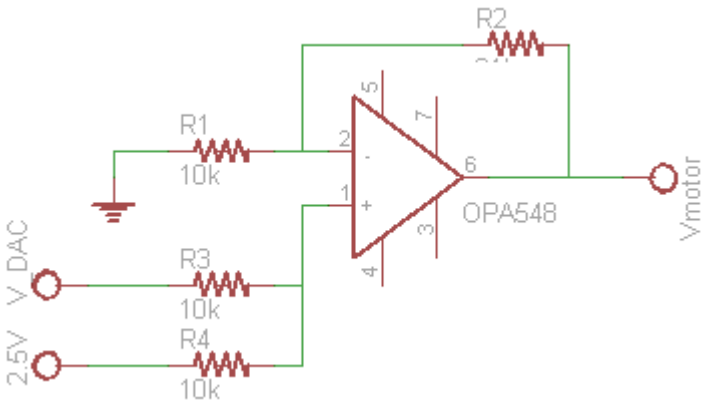
Similar resistors are available in any multiple of ten from 10Ω up to 22MΩ, so why pick 10kΩ? The choice is somewhat arbitrary, however they are commonly used values and meet a minimum requirement. This minimum stems from the voltage difference between V_2 and V_3 and the maximum current that the DAC can supply (50mA). The sum of the resistors R_1 and R_2 must be such that they limit the current across this up to 2.5V difference to below 50mA, which equates out to being at least 50Ω. After choosing these resistors and grounding V_1, the schematic and equation are as Figure 9.2. The unconnected pins in the schematic are for the power supplies, an enable pin, and a current limiter. Additional information can be found on the data sheet as to what should be connected.

9.1.2 PWM Motor Driver

As opposed to the linear motor drivers, which change the motors input voltage to change the speed, running a motor via a PWM signal controls the speed by changing the amount of time the motor is powered for. This has the benefit of increasing the torque because full voltage, and therefore full torque, is available part of the time. This allows for greater loads without stalling when not running at full power.

PWM drivers generally utilize H-bridges. An H-bridge, so called due to its shape in schematics, utilizes four switches, either mechanical or transistors, to allow current to flow through the motor in either direction (see Figure 9.3).

While H-bridges are easy to build, there are numerous chips with one or more H-bridges built in. These chips may require external control of all four switches, however many require only two inputs. Powering one of these inputs



$$V_{motor} = \frac{1 + \frac{91k\Omega}{10k\Omega}}{1 + \frac{10k\Omega}{10k\Omega}} * (V_{DAC} - 2.5V) - \frac{91k\Omega}{10k\Omega} * (0V)$$

$$V_{motor} = 5.05 * (V_{DAC} - 2.5V)$$

Figure 9.2: Schematic and equation for completed differential op-amp design.

while the other is low will cause the motor to spin in one direction and vice versa for the other direction. Some H-bridges can support using one input as direction and the other as an active low pulse, however this can damage or destroy some H-bridges so attention must be payed to ensure this doesn't happen.

Example

Whereas most of the complexity in the linear driver was designing the op amp circuit and the code was just setting the DAC through SPI, here is just the opposite. The H-bridge in question, an L6205 produced by STMicroelectronics requires only two lines from the microcontroller as input, and the two wires to the motor as output. There is some additional circuitry required for the supply and enable, however there is an example of how to do this in the datasheet for the H-bridge. The trade-off comes in via the increased complexity of the code. Instead of setting the SPI and forgetting about it until the next time the speed needs changing, the microcontroller must be constantly generating a PWM signal on one of two pins with the other low. Thankfully, this is possible with one timer and no interrupts.

Recall back to section 8.2 where the registers for timer 0 were discussed. In the description for `TCCR0A` and `TCCR0B` it was mentioned that the clock can toggle external pins. Each clock on the ATmega644p has two external pins that the clock can control. Pins `OC0A` and `OC0B`, also known as `PB3` and `PB4` respectively), can each be set, cleared, or toggled when `TCNT0` matches the corresponding compare register (`OCR0A` or `OCR0B`).

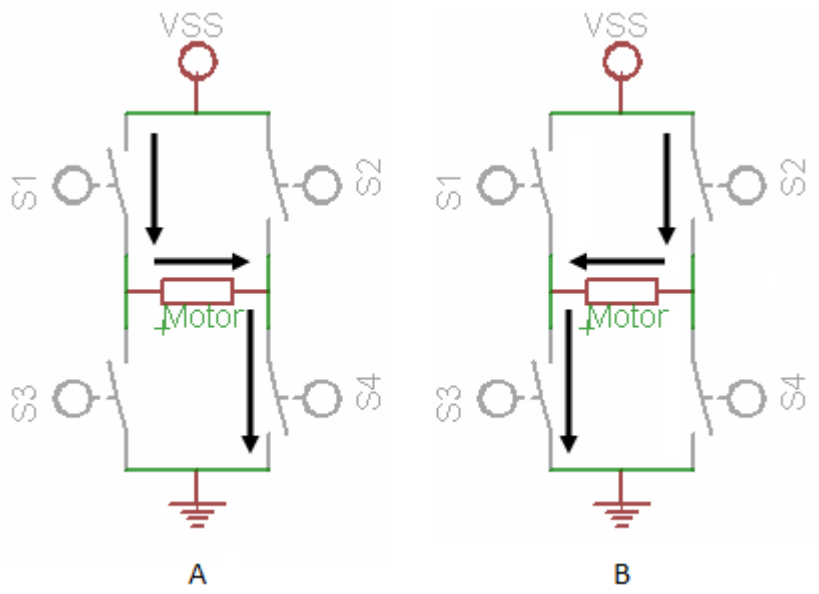


Figure 9.3: Two modes of H-bridge driving.
A) Switches 1 and 4 closed, CW spin of motor.
B) Switches 2 and 3 closed, CCW spin.

As the initial direction of the motor is unknown, the timer should be set up to not enable either pin. As both will be needed eventually, neither compare register can be used to refine the frequency of the PWM signal, the choices are somewhat more limited. Furthermore, the waveform generation mode options are narrowed down to only mode 3, which is the fast PWM with `TCNT` resetting at its maximum value rather than on a compare match.

```
void hbridge_timer_init() {
  TCCR0Abits._w = 0;     //ensure output pins are disabled
  TCCR0Abits._WGM = 3;   //Set mode 3 (fast PWM, top=0xff)
  TCCR0Bbits._WGM = 0;   //remaining bit of mode 3
  TCCR0Bbits._CS = 4;    //256 prescaler.  F = 31.25kHz
        //L6205 H-bridge max frequency = 100kHz
  OCR0Abits._w = 0;     //ensure register clear
  OCR0Bbits._w = 0;     //ensure register clear
  TIMSK0bits._w = 0;    //disable all interrupts
}
```

In order to safely set a speed and direction, several tasks must be done in order. Following this list will prevent both inputs to the H-bridge from ever being high simultaneously, as well as prevent any pulses to be longer than desired.

- Disable clock.

- Clear output pins.
- Disable previous pinout if changing directions.
- Enable new pinout if changing directions.
- Set new compare value. Both compare registers can be set to the same value.
- Clear `TCNT` (optional).
- Enable clock.

```
/**
 * \func hbridge_set_speed
 * \brief Sets speed and direction of H-bridge via timer0
 * \param speed Duty cycle of PWM signal
 * \param direction 0 or !0 sets direction of H-bridge
 *
 * A direction of 0 causes output on the OC0A pin
 * non 0 is OC0B pin.
 */
void hbridge_set_speed(char speed, char direction) {
  char prescale = TCCR0Bbits._CS;  //save clock prescaler
  TCCR0Bbits._CS = 0;  //disable clock
  PORTBbits._P3 = 0;
  PORTBbits._P4 = 0;  //clear output pins
  if(TCCR0Abits._COMA && direction) {
      //swap from 0 direction to !0 direction
      TCCR0Abits._COMA = 0;
      TCCR0Abits._COMB = 2;  //set on compare
  } else if(TCCR0Abits._COM0B && !direction) {
      //swap from !0 direction to 0 direction
      TCCR0Abits._COMB = 0;
      TCCR0Abits._COMA = 2;  //set on compare
  }
  OCR0Abits._w = speed;
  OCR0Bbits._w = speed; //set speed
  TCNT0bits._w = 0;
  TCCR0Bbits._CS = prescale; //restore clock
}
```

9.2 Servos

Another popular choice for creating and controlling motion is a servo. Unlike a motor, which settles into a speed/torque combination based on the load and electrical power supplied, servos have error-correcting negative feedback built in which causes them to seek the specific speed or position they are instructed to. The majority of servos have a limited range of motion, commonly limited

to a mere 180 degrees or less of rotation at the output shaft, and the position they move to is set by the incoming signal. Servos will try with all their might to reach that set point and hold there against any forces, which is useful in many applications. There are some servos that provide continuous rotation with speed control rather than position, but they are less common and accept the same types of signals, so they shall be ignored for now.

Servos have three wires, power, ground and control. The control wire is the only one that need be connected to the microcontroller. Servos work off of a PWM signal, though the duty cycle does not range from 0-100%. Rather than looking at the duty cycle, a servo looks at the width (or duration) of the pulse. A pulse of 1.5 ms will cause the servo to move to its neutral position, which is the center of its range of motion. While the range of motion and timings vary from model to model, the relationship is linear and will appear on the datasheet. Servos expect this control pulse at least every 20ms.

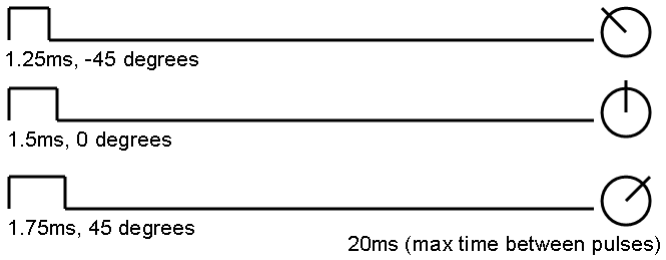


Figure 9.4: Common servo timings for three angles.

Servos have several benefits over motors for certain applications. If, for instance, only a small motion or a specific position is desired a servo is likely the better choice due to its built in error correction. For continuous motion applications such as drives, motors remain the better choice. One other benefit of servos is the reduction in the number of components required. In fact most servos will only require itself, a power supply capable of providing the voltage and current it needs, and the microcontroller to function. No extra interfacing components are required. Combine this with the PWM capabilities of the timers on many microcontrollers, and servo control becomes simpler than anything since chapter 3.

9.3 Sensors

The ability to control ones environment is all well and good, however isn't incredibly useful without the ability to know what is actually occurring, both as a result of said control and the effects of outside influences. To this end there are innumerable sensors: temperature, light, position, distance, the list goes on. From a microcontroller's standpoint, there are really only about four types: those that use a communications protocol, analog signals, a one-bit digital signal (on or off), or a series of pulses.

Digital Signals

One bit digital signals have two possible states: on or off. This often relates to a switch or whether a given device is powered or not. This was covered in depth in the digital input section of chapter 3.

Analog Sensors

A great many sensors are analog. These include thermocouples, photoresistor circuits and most range-finders to name a few. While the data coming from each means different things, once translated into a signal the microcontroller can accept (any amplification or current limiting that is required), the microcontroller sees them as much the same device; an analog signal sent to the ADC. It is up to the developer and the code to interpret what the data means. As such, refer to chapter 4 and the data sheet for the sensor to use these sensors. The data sheet will explain what the voltages mean, and any equations required to convert a voltage into a meaningful result.

Sensors Using Serial Communications

Any analog sensor can be turned into a sensor that communicates via some form of serial through the use of an external ADC, and many have these ADCs built in. These can communicate over any of the various methods of serial communication, however SPI or TWI are the most common. Refer to chapter 7 for more information about serial communications.

Encoders

Encoders are sensors that give information on distance covered, either rotationally or linearly. The rotational encoders are made of a disk with a series of holes and a sensor that can detect these holes. Each time one of these holes is detected a pulse is sent to the microcontroller. When counted, these pulses can be translated into a distance. Basic encoders use one signal line, however this only allows information on total distance covered, and does not provide any information on direction. There are encoders that provide this additional information (quadrature encoders) by providing a second set of pulses slightly offset, however this becomes more difficult for the controller to interpret.

Many microcontrollers have a timer/counter peripheral with an option to use an external clock signal (e.g. timer2 on the ATmega644p). When using a basic encoder, the encoder can be used as the clock signal and the number of encoder ticks that have occured can be read from the counter's data register. The use of quadrature encoders precludes the use of this method. Each time a tick occurs on one of the channels, the second channel must be checked. It's status (high or low) determines the direction, and then a counter can be incremented or decremented as necessary. This can be accomplished using a single pin-change interrupt and a software counter. Additional accuracy can be attained by looking at both the rising edge and falling edge of a signal, looking at one

edge of both signals, or both edges of both signals. The first two options double the number of ticks per distance measured, while the third option doubles it again.

When using high-accuracy encoders and high-speed motors, this many interrupts can cause problems by taking up all of the processors time, or missing interrupts. This situation can also cause the counters to overflow too quickly to be read on 8 or even 16 bit systems causing loss of data. If this is an issue, there are dedicated encoder counters available which can use larger counter registers and are designed to handle the high speed ticks. These can even handle the quadrature calculations, and then communicate with a microcontroller via serial.

9.4 Conclusion

This covers only a tiny fraction of the sensors and actuators available. Completely unmentioned thus far have been inertial navigation sensors (gyros, accelerometers), pneumatic actuators and more. This chapter was not intended to list what was available, but to provide a starting point on how to use whatever might be needed.

Chapter 10

Real-Time Operating Systems

A real-time operating system (RTOS) is any operating system that handles application requests in real time. For personal computers running OSs such as Windows, Mac OS or some *NIX, this entails jumping back and forth between threads as the various running applications need processor time or the user makes a request via the mouse or keyboard. In the case of microcontrollers an RTOS generally ensures that a given set of tasks are completed in an appropriate amount of time and attempts to prevent any of these tasks from consuming excessive processor time or external resources. There are two broad categories of RTOS, preemptive schedulers and cooperative schedulers.

10.1 Preemptive schedulers

A preemptive scheduler is one in which the controlling software, the scheduler, can preempt low priority processes to allow higher priority processes to run. When running multiple tasks or processes the scheduler maintains a priority list containing each process and its priority. When the priority of a process exceeds that of the currently executing process the scheduler interrupts the process, saves its current state and allows the new process to run. Properly executed, this ensures that all processes get at least some CPU time, with the focus on those that are more important. If the scheduler is implemented with a timer which can force changes between processes tasks that are taking longer than expected can be paused and finished later, thus preventing the entire schedule of running processes from being delayed by a hung process.

In modern computing, all of the commonly used operating systems used in a personal computer or larger systems such as academic or business computers use are capable of preemptive scheduling. This includes operating systems such as all versions of Windows since Windows 95, Mac OS X, Linux and Unix based systems as well as others. Preemptive schedulers are less common in

microcomputers (such as cell-phones) and microcontrollers, but certainly do exist. Examples of these are Windows Mobile and FreeRTOS. Even Unix based OSs can often be trimmed down and run on microcontrollers. This being said, for most applications preemptive schedulers are more complex and powerful than are truly necessary for microcontrollers as microcontrollers are usually working on a single problem composed of several related tasks.

10.2 Cooperative schedulers

Cooperative schedulers are less centralized and rely on the programmer and the various tasks to cede processor usage to other tasks. As such, a poorly designed system is likely to hang and cause timing issues. These systems must be designed in such a way that each process is guaranteed to complete in a given time frame, or track its own time usage and pause itself when some time limit is reached. It can then regain processor usage at a later time to complete itself. One of the simpler cooperative schedulers is a round-robin system, of which an example is given below.

Cooperative schedulers were common in personal computing as an intermediate stage between the early non-multitasking systems (such as DOS) and the more modern preemptive schedulers. Well-known examples include the versions of Windows prior to Windows 95 and Mac OS prior to OS X. As cooperative schedulers are easy to implement and work well at maintaining multiple repetitive tasks, they are often used in some of the more complex microcontroller applications, as the example below will demonstrate.

10.3 Round-Robin example

A round-robin RTOS is perhaps the simplest form of RTOS. It consists of a number of scheduling blocks of a pre-determined duration within which each task must fit. An appropriate analogy would be to the periods during a school day. Each one lasts a specific period of time before the student moves on to the next. The passing period between classes is the equivalent of the time taken by the scheduler to switch between tasks. The rest of this chapter is an example of designing and implementing a round-robin RTOS.

Tasks

The first step is to determine the requirements for the system and what the various tasks to be accomplished are. For this example assume a wheeled robot is tasked with manipulating an object with an arm and navigating about its environment. To this end it has a pair of linearly driven (not PWM) drive motors which are also responsible for steering, a servo controlling the arm with a potentiometer for position feedback, a 3-axis IMU with an SPI interface, a pair of IR range-finders and a communications link to a remote computer. The control for each of these nine devices is an individual task. It may eventually be

possible to combine some of these into a single time slot, however for the moment they must all be assessed individually. These are the tasks, the peripherals they require, and any additional notes about them

- Drive motor control (x2): Both motors can be run using the same DAC if the DAC has multiple output channels (most do). The DAC is controlled via the SPI bus and requires only two bytes to set the speed of a motor.
- Servo control: An easy method of controlling the servo is to use a clock in PWM mode. As the clock operates independently of the code and does not even need to throw interrupts, this task could be a short as saving a new value into a compare register for the clock. Most likely there will be a little bit of calculation before hand, but either way this is likely the shortest task.
- Potentiometer: The potentiometer is for feedback from the arm position. While servos often have their own internal feedback for reaching the desired location, external feedback to the controller is also often desirable. This uses the ADC.
- 3-axis IMU: An IMU's (Inertial Motion Unit) communications can come in one of several flavors. The more basic variety outputs an analog signal for each of the axis. This can absorb a significant amount of physical and controller pins, so an ADC is often packaged with an IMU and some form of serial communications such as SPI is used to carry the data to the controller. This is assumed to be the case here. A reading of all three axis would likely require six bytes of communications; half a byte for each request and 10 to 12 bytes of data sent back to the controller.
- IR range-finders (x2): IR range-finders such as those SHARP produces are analog. The downside to them is they only have a refresh time of approximately 40 milliseconds, meaning multiple attempts to read the value in a short period will result in the same reading. If the RTOS is capable of running through all of its tasks more rapidly, then these tasks could be skipped every so often without issue.
- Communications link: This links the robot to a remote station with far more processing power, and allows the robot to focus on controlling itself while the remote station is concerned with higher level decision making, including where the robot should be going and mapping if necessary. This task is the hardest to quantify in a theoretical exercise such as this due to both not knowing the volume of information being transmitted and received, as well as not knowing when during the RTOS cycle the data will be received. This second part is actually very important to note. As the received buffer on the ATMega644p is only a single byte, when data is received it needs to be handled immediately, even though this is likely only to entail storing it for attention later. Furthermore, this means that

each task needs to allocate at least a little extra time to account for any interrupts from the communications task that may occur.

Time Frames

Now that the required tasks have been determined, the next step is to approximate how long each task should take. This could be in terms of clock cycles or milliseconds. In this example the 1MHz clock on the ATMega644p is used which means the usage time for each peripheral is about a tenth of a millisecond (0.0001sec) or less, although any form of data processing needs to be included as well. This time was determined by looking at the length of time it takes the peripheral to perform the hardware portions of the tasks. For the ADC this means the actual conversion (12 clock cycles [11]) and for the various serial communications methods it is calculated from the peripheral's clock frequency and the number of bytes to be transmitted. In all actuality, these tasks, with the possible exception of the communications, should require less than a millisecond. If the actual duration of each task is in doubt, or is needed more specifically, the task can be set up in a trial program using a timer to time exactly how long it requires. With eight tasks requiring under a millisecond each, and a ninth requiring an unknown but still small time frame, one option would be to set the RTOS to have ten 1ms time blocks, allowing the communications a second millisecond if it so requires.

On the other hand, looking back to the IR range-finders, only one out of four passes through the tasks would be useful for these devices. Furthermore, it may be possible to combine some of the short tasks into a single longer task. Depending on the setup the IR range-finders may be combinable into a single task, however as this isn't always the case they have been left separate in this example. The drive motor control tasks, however, can and perhaps should be. Due to how the calculations for this type of drive system works, the values for both motors are determined simultaneously, and it is likely unwise to set the motors at different times or there would be un-desired turning. These two tasks can therefore be combined into one.

Finally one must consider the response time of the entire system. Can the remote computer process the information and return instructions within 10ms, and is 10ms long enough to get a meaningful change out of the physical system? Often times this is a little bit faster than is truly necessary. A time of 20ms for the cycle through all the tasks may be somewhat more appropriate, resulting in a system cycle rate of 50Hz. This also allows a little more flexibility in timing, as well as more time for communications and calculations. Also, if some task, such as the IMU, does need to be run more often, there are empty time slots that can accommodate a second running of the task.

Scheduling

Now that the basic framework is tentatively decided (it can always be changed later if necessary), the time has come to begin deciding the order of the tasks.

For instance in the potentiometer/servo control/feedback loop, does it make sense to read the potentiometer immediately before, immediately after, or some neutral time between runs of the servo control task. If a single sensor is being read multiple times (such as the IMU) does it make more sense to read twice in quick succession to eliminate noise, or separate them to achieve a more accurate time average? These, and similar others, are questions that need to be examined for each system individually, however it is generally best to have the control tasks as soon after the sense tasks as possible. Table 10.1 lists the 20 time slots and what is included in each. The "as needed" tasks can be used for additional communication or calculation needs as they arise, or are there as padding in case a task runs long. Communication has also been given several time slots to ensure sufficient time for it and to keep the remote computer as up-to-date as possible, and the potentiometer and servo tasks have been combined into a single task as there is no need for communications between reading the potentiometer and updating the servo.

Slot	Task
0	IMU Reading
1	
2	Communication
3	
4	Drive Motor Control
5	
6	Potentiometer and Servo
7	
8	Communication
9	
10	As needed
11	
12	IMU Reading
13	
14	IR Range-finder
15	Communication
16	
17	As needed
18	
19	

Table 10.1: Schedule of tasks in round-robin RTOS example.

Coding

Finally it is time to start programming the RTOS. There are two schools of thought on how to do this. The first approach is to program it top down; that is to start with the framework of the RTOS and work ones way down to the

individual tasks. The second approach is to program the tasks first and test them before creating the RTOS framework. This second method allows one to test each segment as its completed both for operation and for timing, and prevents mistakes in the timing approximations done previously to cause problems in an already completed framework. With the amount of spare time in this system timing problems are not a great concern, and either method is perfectly feasible. As this chapter is focused on the RTOS and the individual peripherals have already been covered, assume that the functions mentioned have been implemented elsewhere. Furthermore, it is very useful to have each time block or set of blocks be a single function for both readability and modularity's sakes.

Initialization

Before entering the loop that is the RTOS, every device that will be used needs to be set up, variables initialized and any other preparatory work completed. In this example it will require two timers (one for the servo (chapter 9), one for the RTOS), the SPI (chapter 7), the ADC (chapter 4) and one of the USARTs (chapter 7). The RTOS clock should be set to trigger an interrupt every millisecond. One way to do this is:

```
/** initRTOS()
 * Initialize and setup RTOS using timer 2.
 * TCNT2 register and RTOS counter
 * should both be reset just prior to entering RTOS loop
 * Sets timer to 1ms interrupts
 */
void initRTOS() {
        TCCR2Abits._WGM = 2;  //set mode to CTC
        TCCR2Bbits._CS = 0x01;  //set prescaler to 8
        OCR2Abits._w = 125;  //125 counts at 1MHz/8 = 1ms
        TIMSK2bits._OCIEA = 0x02;  //int on compare match
}
```

Interrupt

The interrupt service routine (ISR) for the RTOS is quite short. It simply increments a counter and once the counter exceeds the number of scheduled blocks, returns the counter to 0 through the use of a modulus function.

```
ISR(TIMER2_COMPA_vect){
        RTOScounter = (RTOScounter+1)%25;
}
```

RTOS Loop

The RTOS loop itself is an infinite `while` loop using a `switch-case` to determine which task to run. A `switch-case` operates similarly to a series of `if then else` statements, but is faster and continues executing until a `break` is reached,

even if it involves entering the next `case`. Before entering the loop the RTOS timer and counter need to be reset as stated in the comments for the `initRTOS` function and interrupts enabled. This then is the remainder of the round-robin RTOS example.

```
long time;
int RTOScounter;

int main (void) {
  initialization_functions(); //including initRTOS()

  time = 0; //Time in cycles since RTOS started
              //one cycle = 20ms
  RTOScounter = 0;
  int taskID = 0;
  TCNT2bits._w = 0;
  while(1) {
    switch(RTOScounter) {
    case 0:
      IMUReadingTask();
      break;
    case 1:
      break;
    case 2:
      CommunicationTask();
      break;
    case 3:
      break;
    case 4:
      DriveMotorControlTask();
      break;
    case 5:
      break;
    case 6:
      PotServoTask();
      break;
    case 7:
      break;
    case 8:
      CommunicationTask();
      break;
    case 9:
      break;
    case 10:
      break;
    case 11:
      break;
    case 12:
      IMUReadingTask();
      break;
```

```
    case 13:
      break;
    case 14:
      IRTask();
      break;
    case 15:
      CommunicationTask();
      break;
    case 16:
      break;
    case 17:
      break;
    case 18:
      break;
    case 19:
      break;
    }
    time++;
    while(RTOSCounter == taskID){}
    //this loop prevents a task from running twice
    //during its block and waits until the
    //RTOSCounter changes before continuing
  }
}
```

Breaks are not actually required for every `case`. They are only necessary in the `case`s before those with actual tasks to perform. Likewise the `time` variable is not necessary, it merely tracks the cycles passed since the RTOS loop started and can be used for time stamps. Another possible use for it is tracking when to sample the IR sensors, as their refresh rate is such that they will only update approximately every other pass. It does no harm to reread them every pass but if time does become an issue only reading one of them per pass and alternating which is read is one method of saving a fraction of a second.

10.4 Conclusion

When using microcontrollers for complicated applications involving a variety of individual tasks, implementing an RTOS can be very useful in guaranteeing some form of scheduling for the tasks. Whether it is more advantageous to use a preemptive or cooperative scheduler is dependent on the application and tasks in question. Often times, however, the most basic scheduler is sufficient, if not overkill. For further information about the various forms of RTOS and how to implement them, search the local library or bookstores for books on real time operating systems, embedded systems programming architecture and similar topics.

Chapter 11

The final example

Thus far, this text has addressed many topics. Now it is time to put everything together into a single example containing something from most every chapter. This example will cover the entire process, from problem statement to completed code with varying levels of analysis. Sections whose analysis has been completed in depth in previous channels (such as the setup of certain devices) will be skimmed over. Refer to the previous chapters for more information in these cases.

11.1 Problem statement

The task in this example is to drive a small (under 5kg) robot parallel to a wall at a distance of 30cm ±10% (3cm). The robot will travel at a minimum of 4cm/s and accelerate from stop at a minimum acceleration of 15m/s^2. The robot is 30cm long.

11.2 Design decisions

As with many problems, there are three segments to the solution: sense, which detects the current state of the robot; decide, which decides what the robot should do; and control, which implements the decision. These three segments will be examined in the order of control, sense and finally decide. This order is in part by choice and in part mandated by the order in which design decisions effect other decisions that must be made down the line. The remainder of this section follows the logic for making some of the design decisions. While some of the data behind the decisions have been omitted due to the vast array of decisions that go into any solution, the primary requirements have been worked out start to finish.

11.2.1 Control

Upon examining the problem statement, it can be noted that there is a minimum speed and acceleration and that the robot must be steerable. No requirements specify a specific drive or steering system, which means these must be decided upon. A good rule of thumb when making a design decision is to accept the simplest solution that is good enough. While better solutions may exist, they are likely more costly in materials, manufacturing or time. A solution that is good enough is one that meets every requirement with a reasonable margin for error.

There are innumerable drive choices from wheeled robots, to legged robots to other, more esoteric drive systems such as imitating a worm. The simplest and easiest to control of these is a wheeled robot, therefore that is what should be used. Within the category of wheeled robotics there is a wide range of options including number of wheels, number of powered wheels and steering method. Again the simplest of each option should be chosen. This leads to using four wheels (simpler chassis design and stabler in motion than three wheels) with two of them powered by motors and using skid steer. Skid steer, also known as differential steering, uses a difference in the speed of the left and right wheels to turn the robot.

Motor choice

As the problem statement provides a minimum speed, acceleration and a maximum weight for the robot, some of the minimum motor specifications can be calculated. As an example, the required speed of the motor output shaft is calculated here. The wheels are assumed to be 2.875 inches (7.30 cm) in diameter. This choice was made based on the availability of the wheels to the author at the time of writing.

$$shaft\_speed = \frac{min\_speed}{wheel_circumference}$$
$$shaft\_speed = \frac{4cm/s}{7.30cm * \pi}$$
$$shaft\_speed = 10.5rpm$$

While there are numerous suitable motors available, one that fits the bill is a 12V Globe E-2135 gearmotor with a 187.68:1 ratio gearhead [13]. These motors have a maximum rpm of over 5000, resulting in an output shaft speed of over 26rpm, which is more than sufficient. This excess allows the motors to be run at an average of half speed, see section 11.2.3 below for the reasoning behind this choice. The calculations for whether this motor meets the acceleration requirements has been left as an exercise for the reader.

The motor control will be the same setup as used previously in 9.1.1 Linear drivers, using an external 12-bit DAC and an op-amp.

11.2.2 Sense

In order to maintain a set distance, the robot must be capable of detecting whether or not it is in the acceptable range. The most basic form would be a switch on a 30cm-long arm sticking out from the side of the robot. If the switch becomes un-pressed it means the robot is moving away from the wall and a correction must be made. While acceptable by some interpretations of the problem statement, it is a rather silly solution that is prone to problems such as catching on the wall. Some form of ranging sensor must therefore be used.

Some possible choices include infra-red (IR) sensors, ultrasonic rangefinders, RADAR and LADAR (laser rangefinding). RADAR and LADAR can be eliminated straight off. These two methods are best used for long range detection and range-finding, and are prohibitively expensive. This leaves the ultrasonic sensors, which are prone to interference from similar devices as well as the environment, and IR sensors which are non-linear.

In order to decide between the two, the problem should be examined in further detail. First off is the range of operation. The sensors must be able to distinguish three ranges: under 27cm, between 27 and 33cm (the acceptable range), and above 33cm. These three ranges tell the robot that it either needs to move further from the wall, stay put, or move closer. Common hobby versions of both types of sensors are capable of this.

While a single distance measurement can tell which direction the robot needs to go, it does not tell the robot whether it is parallel or not. While not absolutely necessary, having some measure of the angle the robot forms with the wall will allow for a better control algorithm. Not only will the robot be able to adjust its turning speed based on the angle with the wall, it will be capable of more accurately calculating its distance as the sensor will not always be directed perpendicular to the wall. A simple solution to this angle measurement problem is to use two ranging sensors, one fore and one aft. The difference between these measurements will give an indication of the angle. As ultra-sonics can interfere with each other, IR sensors are the better choice here.

Having chosen IR sensors, a specific model that fits the requirements of operational range (30cm) and precision (3cm) must be found. SHARP produces a line of IR sensors with varying minimum and maximum ranges. One of these sensors, the GP2D12, has an operational range of 10.16cm to 80.01cm, easily covering the required range with room to spare. The sensor produces an analog voltage from 0.4V at 80cm, to 2.4V at 10 cm, though non-linearly. The precision must be calculated at 30cm to ensure compatibility with the requirements.

Examining the distance vs. voltage graph in the data sheet (see figure 11.1), the slope of the relationship at 30cm can be found to be approximately 24.3mV/cm. Recalling that the ADC on the microcontroller is 10-bit, it has a resolution of 4.9mV with a 5V reference voltage. This equates to determining the distance to within 2mm, which is well within the requirements.

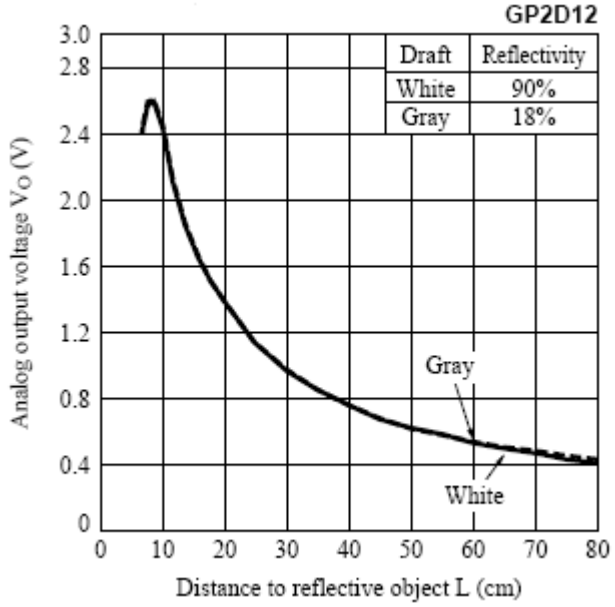


Figure 11.1: Range vs Voltage data for the GP2D12 SHARP IR sensor. [19]

11.2.3 Decision

The final portion of the problem to examine is the process by which the robot decides what control signals are actually sent by examining the sensor results. To this end, some form of control equation must be derived, which the robot then uses to set the speeds of each motor thereby steering the robot. When creating a control law, there are several goals that can be designed for: speed of reaction, simplicity (low processor time and/or memory usage) and optimizing energy usage. As these three are highly dependent on each other, the general procedure is to design for one and possibly maintain some form of minimum requirement on the others. In this case, the design goal will be to use a minimum of processor time and memory.

Starting from scratch, there are two parameters that effect the decision on how the robot moves: distance from the wall and angle with the wall. As they are independent, each one will be the basis for one term of the final control equation.

$$control = distance\_error\_term + angle\_term$$

The incoming signals from the sensors supply the distance to the wall perpendicular to the direction of travel from the front and from the rear of the robot. From these, the distance can be easily calculated. Then angle is slightly more difficult to calculate in that it requires a bit of geometry and trigonometry

(figure 11.2), but still relatively simple.

$$distance\_error = \frac{front\_distance + rear\_distance}{2} - 30cm$$
$$angle = arctan(\frac{front\_distance - rear\_distance}{robot_length}$$

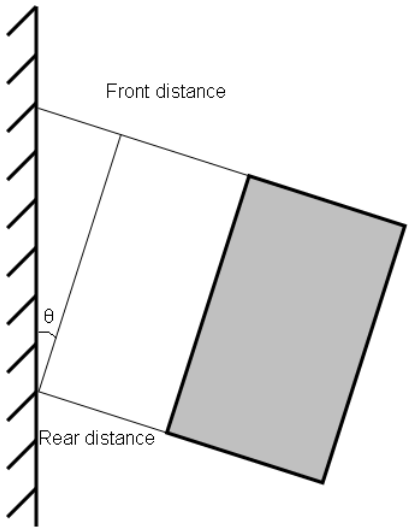


Figure 11.2: Calculating angle formed with the wall based off distance measurements.

These can then be combined together with some form of multiplier on one or both terms to achieve a reasonable control law. There are two problems however. First, the sensors are non-linear which means that if the angle isn't very small, it will have a great deal of error in the calculation; and second, this involves both float math as well as trigonometric functions. This will be very costly in terms of time and memory as the microcontroller cannot easily handle float math, and both the trigonometric functions and converting the ADC reading into an actual distance will require large lookup tables as well as a non-insignificant amount of processor time.

A surprisingly elegant solution in this case is to just use the actual ADC values instead of calculating exact distances, and use the difference in the readings without actually calculating the angle in the following manner:

$$control = distance\_mod * (\frac{front\_ADC + rear\_ADC}{2} - ADC_{30cm})$$
$$+angle\_mod * (front\_ADC - rear\_ADC)$$

Using this control law will have several effects

- The further from the desired distance the robot is, the faster it will attempt to correct.
- The further from the wall the slower this correction will be. This allows for gentle correction away from the wall where there is space, but quicker reaction when the robot is close to the wall
- The further off parallel the robot is, the faster it will attempt to turn back to parallel. Turns will again be magnified when close to the wall.
- Only basic arithmetic is used, allowing for quick calculations. Furthermore, if both tuning values (*distance_mod* and *angle_mod*) are multiples of 2 (as they turn out to be for satisfactory responses), the only operations are addition, subtraction, left-shifting and right-shifting: all of which require a single clock cycle. Multiplication by values other than a multiple of 2 would require an additional cycle.

The value that results from this equation is then added to or subtracted from some average motor speed to turn the robot. In this example, as the DAC used is 12-bits, the control value is multiplied by 4 (left-shifted twice) and applied to both motor values. This maintains the average speed while turning the robot.

11.3 File structure

This program consists of three `.c` files, their associated header files, three peripheral libraries and `reg_structs.h`, a file which contains all the register `struct`s which have been used since chapter 6. The three libraries are for the USART, SPI and ADC, which are the peripherals used in this example. The code files for this program in particular are `RTOSmain.c`, which contains the `main` function and the RTOS itself, `tasks.c` containing the functions called by the RTOS for each task as well as initialization and `motor.c` which contains functions directly related to the motors and encoders. As seen in figure 11.3, the libraries including `reg_structs.h` are in a sub-folder. In actuality they are in a separate folder that is linked to the project by the compiler to appear as a sub-folder. If the libraries are moved, which they shouldn't be, only the compiler's options need to be changed rather than the code itself.

11.3.1 `RTOSmain.c`

There have been some changes to `RTOSmain.c` since the previous example. As a separate header file was created, the `#include` statements and global variable declarations have been removed to there. Additionally, the initialization function was moved out of this file and into `tasks.c`. Finally, a second ISR was added.

The new ISR triggers whenever data arrives via the USART. Its sole task is to store that incoming data in an array of chars (`rx_data` acting as a buffer, and increment a variable (`rx_bytes`) tracking where in the buffer the next byte

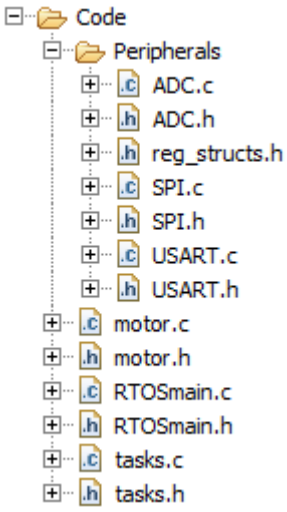


Figure 11.3: File structure of final example showing all files not installed with the compiler.

should be placed. This information is handled later by the communications task. Elsewhere, primarily in the USART library, `int`s are used instead of `char`s for transmitted and received data. This is due to the possibility of having 9-bit characters and the interest in making the library as generic as possible. The ISR uses `char`s as in this application the serial characters are only 8-bits.

```
ISR(USART0_RX_vect) {
    rx_data[rx_bytes] = (char)UDR0bits._w;
    rx_bytes++;
}
```

11.3.2 `RTOSmain.h`

This header file is where the global variable declarations and `#include` statements have been moved to. There are also a few new variable declarations. These are variables for storing the IR sensor readings for use in the motor control task, as well as the serial input and output buffers and their respective position trackers. The `set_point` variable stores the desired distance from the wall as the value that should be read from the IR sensor when that distance is achieved. Finally, an additional inclusion, `tasks.h` has been added which in turn `#include`s all other necessary files.

```
//IR reading and distance storage
//196 = 1 foot
int fIR = 196;
int bIR = 196;
int set_point = 196;
```

```
//USART tx and rx data buffers and current locations
int tx_data[20];
unsigned char tx_bytes = 0;  //track next byte location
char rx_data[50];
unsigned char rx_bytes = 0;  //track next byte location
```

11.3.3 tasks.h

This file should be relatively easy to read and understand except for one thing. There are several variables that are declared with the `extern` keyword. This keyword tells the compiler that these variables are also declared in another file, but that they should be accessible as if declared here as well. In this case, the variables are the motor speed and communications buffer variables that were declared initially in `RTOSmain.h`.

```
extern int fIR;
extern int bIR;
extern int set_point;

extern int tx_data[20];
extern unsigned char tx_bytes;
extern char rx_data[50];
extern unsigned char rx_bytes;
extern long time;
```

11.3.4 tasks.c

This file contains the functions for all of the individual tasks as well as the initialization function. This initialization function (`initialization_functions()`) merely calls further initialization functions for various tasks: the RTOS, ADC, USART and motor functions (which includes the SPI). The RTOS initialization function (`initRTOS()`) which was in `RTOSmain.c` previously is also in this file.

```
void initialization_functions() {
    ADC_init(&ADC_struct); //initialize ADC
    motor_init();   //initialize motor
    USART_init(&USART_struct);
    initRTOS();
}
```

`CommunicationsTask()`

This task covers the transmission of any outgoing data accumulated in `tx_data` and the distribution of the data that has been saved in `rx_data`. After transmitting all the requisite data (by calling `USART_TX()` the function enters a `while` loop which steps over the commands stored in `rx_data`. The loop enters a `switch` statement using the first byte of data. This byte is the command and

causes different `case`s to trigger based on its value. All of the commands are an ASCII character somehow related to the command ('d' for setting the desired distance to the wall for example). Currently only a single command is available, however more can be added easily if desired.

Each command has a specific number of data bytes which are transmitted immediately after. These are used in the execution of the command, and then the `while` loop counter variable `i` is increased by the number of bytes involved in each command.

If the command code is not recognized, which can happen if there is a transmission error or the wrong number of bytes is somehow sent, the `default` block is triggered. This block sends the command code 'x' back to the computer along with all the data that it didn't reach. This tells the computer that an error has occurred and what data was involved. It is then the computers job to determine what commands must be resent.

```
void CommunicationTask(){
    USART_TX(tx_data,tx_bytes);
    tx_bytes = 0;
    unsigned char i = 0;
    //work through received data
    while(i<rx_bytes) {
        //finite state machine for various commands
        switch (rx_data[i]) {
        case 'd': //distance setting
            set_point = (int)rx_data[i+1]<<8 +
                (int)rx_data[i+2];
            i+=3;
            break;
        default:
//error occurred, send back all remaining data
//uneven command lengths means can't necessarily resync.
//remote controller will have to resend commands
            tx_data[tx_bytes] = 'x'; //error indicator
            tx_bytes++;
            for(i=i;i<rx_bytes;i++) {
                tx_data[tx_bytes] = rx_data[i];
                tx_bytes++;
            }
            USART_TX(tx_data,tx_bytes);
        }
    }
}
```

`DriveMotorControlTask()`

This function takes the IR sensor data stored in `fIR` and `bIR` and uses them to calculate the motor speeds to maintain or achieve the desired distance from the wall. The calculation works by first finding the difference between the IR

sensor readings to obtain a measure of what angle the robot is at relative to the wall. Additionally an error term is calculated by subtracting the desired distance from the current location of the robot, which is found by taking the average of the readings. The angle term is multiplied by a `#define`d constant which is a tuning value to adjust the relative importance of the two terms.

$$tune * (fIR - bIR) + \frac{fIR + bIR}{2} - distance$$

This calculation is done entirely using the ADC readings of the IR sensors. This means that the results are non-linear and may not be exactly what is expected looking at the equation without that knowledge. In this instance the non-linearity is acceptable and the function does work.

At this point, a single value has been calculated. This is then used as a speed modifier when setting the motor values, after being multiplied by 4 to increase the speed of the robots corrections. This modifier is added to the DAC setting for half-speed for each motor. This speed setting was a somewhat arbitrary choice. In the code the left and right motors are set to 1020 and 3060 respectively to travel straight. The difference is caused by the fact the motors are pointed in opposite directions, meaning they must rotate in opposite directions to enable the robot to travel forward rather than spin in place. As the DAC is 12 bits and due to how the amplifier circuit is designed, a value of 0 causes full speed in one direction, 4080 is full speed in the other and 2040 is stopped. After setting the two motors, this task then reads the current values of the encoder counter chips and queues that data for transmission to the computer.

```
    int res = TURN_TUNE*(fIR-bIR)+(fIR+bIR)/2-set_point;

    // set motors
    set_motor(LEFT_MOTOR, 1020+(res<<2));
    set_motor(RIGHT_MOTOR, 3060+(res<<2));
```

`IRTask()`

As was previously mentioned, the IR sensors have a refresh rate that is slower than the RTOS, so only one is read each pass. This is taken care of by the `if` statement checking if the `time` variable is even or odd. The sensor reading along with which sensor made the reading is queued for transmission with the command code 'i' (for Infra-red) for the front IR sensor, or 'j' for the rear, as well as stored either the `fIR` or textttbIR as appropriate. The 'j' comes from being the ASCII code numerically 1 greater than 'i'.

Other tasks

Dummy functions are included for the other tasks in the RTOS, however as they are not used in this example they have been left un-implemented.

11.3.5 `motor.h` and `motor.c`

`motor.h` contains a number of `#define` statements used primarily used for tracking which device is which. This includes the slave select pins for all three SPI devices (DAC and two encoder counters.

`motor.c` contains three functions: motor initialization, setting motor speeds and reading the encoder counters. The `motor_init()` function handles all the initializations required to use the motors which include configuring the SPI, both encoder counters and the DAC. There is a major problem that is run into here however, as the DAC expects that the SPI clock signal will be high when not in use while the encoder counters expect it to be low. This requires two separate configurations which will be swapped between whenever a different device must be accessed. As two configurations are required, two separate `SPI_struct_type`s must be created, one for each configuration (see the section on the library files below). The only difference between these `struct`s is the `clockpolarity` field.

```
//DAC init struct.  See SPI.h for struct definition
SPI_struct_type DAC_struct = {.intenable = 0,
                .dataorder = MSBfirst,
                .clockpolarity = 1, //leading edge fall
                .clockphase = 0, //sample on leading
                .clockprescale = SPIPRE2};

//Encoder init struct.  See SPI.h for struct definition
SPI_struct_type Enc_struct = {.intenable = 0,
                .dataorder = MSBfirst,
                .clockpolarity = 0,
                .clockphase = 0,
                .clockprescale = SPIPRE2};
```

The `motor_init()` function handles the switches between configurations when it needs to, whereas the other two functions have logic at the beginning of the function that only changes the configuration if the other configuration was used last. This is the purpose of the `cur_device` variable and the `ENC_DEVICE` and `DAC_DEVICE` definitions in the header file.

```
     //setup SPI for DAC
    if(cur_device == ENC_DEVICE) {
        SPI_init(&DAC_struct);
        cur_device = DAC_DEVICE;
    }
```

11.3.6 ADC, SPI and USART libraries

All three of these libraries are set up in the same method. Each one contains an initialization function (i.e. `ADC_init()`) who's only argument is a pointer to a specific type of `typedef`ed `struct`. These `struct`s contain fields for all the configuration options. As mentioned in chapter 6, these `struct`s allow for more generic initialization functions and make future changes to the functions

easier. The other functions in each library are those required for operation, be it reading, writing or some combination thereof.

In addition to the `struct typedef`s, the header files have numerous `#define` statements. The majority of these are the options for configuration, giving textual names that are usable instead of requiring the memorization of the numeric values or constantly referencing the datasheet.

11.4 Conclusion

While complex tasks may seem daunting at first, breaking it up into individual tasks and tackling each one individually should make any task achievable. The important things to remember are to make liberal use of libraries, and keep code as generic as possible to make future programming quicker and easier.

Afterword

This text has attempted to provide solutions to several of the problems faced by RBE3001 students in Worcester Polytechnic Institute's Robotics Engineering course. The first two chapters, as well as the schematic analysis for the digital I/O and the ADC attempted to provide students with a better understanding of what microcontrollers are and how they function, as well as teaching them how to read, understand and learn from datasheets and schematics. Chapters 5 and 6 were aimed at improving the coding practices of students, both in terms of improving code re-use through libraries as well as properly commenting and documenting their code. Additionally, several uses of `struct`s were presented which will again aid in code re-use and organization. Chapter 10 demonstrated that if the organization of the code is properly thought out from the beginning, the execution can be much improved via structures such as an RTOS. The remaining chapters introduced and used various peripherals and external devices useful in robotics.

In addition to these tasks, the entire text had the underlaying goal of teaching the students how to obtain this information for themselves in the future. Good practices are demonstrated, and the origins of important information are displayed and explained initially before backing off on the detailed explanations to allow readers to work on each skill in progressive steps. It has been to goal of this text to allow readers to be able to easily branch off from the devices presented herein into other microcontrollers and devices using the datasheets and a minimum of other resources.

Appendix A

regstructs.h

This is a header file that allows access to all of the registers of the ATmega644p as structs via the methods presented in chapter 6. None of the names contained herein overlap with the ones provided in WinAVR, so there is no conflict and registers can still be accessed in the methods used in the first five chapters. Mixing paradigns is highly not reccommended.

```
/**
 * \file reg_structs.h
 * \author Peter Alley
 * \date 11-18-09
 *
 * This file redefines all of the registers of the
 * ATmega644p as structs to allow for easy
 * access to individual bit fields in each register.
 */

#ifndef REG_STRUCTS_H_
#define REG_STRUCTS_H_

/**
 * Generic 8-bit register
 */
typedef union {
        struct {
                unsigned _P0:1;
                unsigned _P1:1;
                unsigned _P2:1;
                unsigned _P3:1;
                unsigned _P4:1;
                unsigned _P5:1;
                unsigned _P6:1;
                unsigned _P7:1;
        };
        struct {
```

```
                unsigned _w:8;
        };
} __8bitreg_t;

/**
 * DIO
 * PIN, DDR, PORT for A,B,C,D
 */

/**
 * ADC port
 */

/**
 * SPI port
 */
typedef union {
        struct {
                unsigned        :5;
                unsigned _MOSI  :1;
                unsigned _MISO  :1;
                unsigned _SCK   :1;
        };
        struct {
                unsigned _w     :8;
        };
} __SPIPORTbits_t;

/**
 * TIFR0
 */
typedef union {
        struct {
                unsigned _TOV0  :1;
                unsigned _OCF0A :1;
                unsigned _OCF0B :1;
                unsigned        :5;
        };
        struct {
                unsigned _w:8;
        };
} __TIFR0bits_t;

/**
 * TIFR1
 */
typedef union {
        struct {
                unsigned _TOV1  :1;
                unsigned _OCF1A :1;
```

```
                unsigned _OCF1B :1;
                unsigned        :2;
                unsigned _ICF1  :1;
                unsigned        :2;
        };
        struct {
                unsigned _w     :8;
        };
} __TIFR1bits_t;

/**
 * TIFR2
 */
typedef union {
        struct {
                unsigned _TOV2  :1;
                unsigned _OCF2A :1;
                unsigned _OCF2B :1;
                unsigned        :5;
        };
        struct {
                unsigned _w     :8;
        };
} __TIFR2bits_t;

/**
 * PCIFR
 */
typedef union {
        struct {
                unsigned _PCIF0 :1;
                unsigned _PCIF1 :1;
                unsigned _PCIF2 :1;
                unsigned _PCIF3 :1;
                unsigned        :4;
        };
        struct {
                unsigned _w     :8;
        };
} __PCIFRbits_t;

/**
 * EIFR
 */
typedef union {
        struct {
                unsigned _INTF0 :1;
                unsigned _INTF1 :1;
                unsigned _INTF2 :1;
                unsigned        :5;
```

```
        };
        struct {
                unsigned _w     :8;
        };
} __EIFRbits_t;

/**
 * EIMSK
 */
typedef union {
        struct {
                unsigned _INT0  :1;
                unsigned _INT1  :1;
                unsigned _INT2  :1;
                unsigned        :5;
        };
        struct {
                unsigned _w     :8;
        };
} __EIMSKbits_t;

/**
 * GPIOR0
 */
typedef union {
        struct {
                unsigned _b0    :1;
                unsigned _b1    :1;
                unsigned _b2    :1;
                unsigned _b3    :1;
                unsigned _b4    :1;
                unsigned _b5    :1;
                unsigned _b6    :1;
                unsigned _b7    :1;
        };
        struct {
                unsigned _w     :8;
        };
} __GPIOR0bits_t;

/**
 * EECR
 */
typedef union {
        struct {
                unsigned _EERE  :1;
                unsigned _EEPE  :1;
                unsigned _EEMPE :1;
                unsigned _EERIE :1;
                unsigned _EEPM0 :1;
```

```
                unsigned _EEPM1 :1;
                unsigned        :2;
        };
        struct {
                unsigned        :4;
                unsigned _EEPM  :2;
                unsigned        :2;
        };
        struct {
                unsigned _w     :8;
        };
} __EECRbits_t;

/**
 * EEDR
 * 8bitreg
 */

/**
 * EEAR
 */
typedef union {
        struct {
                unsigned _low   :8;
                unsigned _high  :8;
        };
        struct {
                unsigned _w     :16;
        };
}__16bitreg_t;

/**
 * EEARL
 * 8bitreg
 */

/**
 * EEARH
 * 8bitreg
 */

/**
 * GTCCR
 */
typedef union {
        struct {
                unsigned _PSRSYNC       :1;
                unsigned _PSRASY        :1;
                unsigned                :5;
                unsigned _TSM           :1;
```

```
        };
        struct {
                unsigned _w     :8;
        };
} __GTCCRbits_t;

/**
 * TCCR0A
 */
typedef union {
        struct {
                unsigned _WGM00         :1;
                unsigned _WGM01         :1;
                unsigned                :2;
                unsigned _COM0B0        :1;
                unsigned _COM0B1        :1;
                unsigned _COM0A0        :1;
                unsigned _COM0A1        :1;
        };
        struct {
                unsigned _WGM   :2;
                unsigned        :2;
                unsigned _COMB  :2;
                unsigned _COMA  :2;
        };
        struct {
            unsigned            :4;
            unsigned _COM0B     :2;
            unsigned _COM0A     :2;
        };
        struct {
                unsigned _w     :8;
        };
} __TCCR0Abits_t;

/**
 * TCCR0B
 */
typedef union {
        struct {
                unsigned _CS00  :1;
                unsigned _CS01  :1;
                unsigned _CS02  :1;
                unsigned _WGM02 :1;
                unsigned        :2;
                unsigned _FOC0B :1;
                unsigned _FOC0A :1;
        };
        struct {
                unsigned _CS    :3;
```

```
                unsigned _WGM   :1;
                unsigned        :2;
                unsigned _FOC   :2;
        };
        struct {
                unsigned _w     :8;
        };
} __TCCR0Bbits_t;

/**
 * TCNT0
 * 8bitreg
 */

/**
 * OCR0A
 * 8bitreg
 */

/**
 * OCR0B
 * 8bitreg
 */

/**
 * GPIOR1
 * 8bitreg
 */

/**
 * GPIOR2
 * 8bitreg
 */

/**
 * SPCR
 */
typedef union {
        struct {
                unsigned _SPR0  :1;
                unsigned _SPR1  :1;
                unsigned _CPHA  :1;
                unsigned _CPOL  :1;
                unsigned _MSTR  :1;
                unsigned _DORD  :1;
                unsigned _SPE   :1;
                unsigned _SPIE  :1;
        };
        struct {
                unsigned _SPR   :2;
```

```
                unsigned        :6;
        };
        struct {
                unsigned _      w:8;
        };
} __SPCRbits_t;

/**
 * SPSR
 */
typedef union {
        struct {
                unsigned _SPI2X :1;
                unsigned        :5;
                unsigned _WCOL  :1;
                unsigned _SPIF  :1;
        };
        struct {
                unsigned _w     :8;
        };
} __SPSRbits_t;

/**
 * SPDR
 * 8bitreg
 */

/**
 * ACSR
 */
typedef union {
        struct {
                unsigned _ACIS0 :1;
                unsigned _ACIS1 :1;
                unsigned _ACIC  :1;
                unsigned _ACIE  :1;
                unsigned _ACI   :1;
                unsigned _ACO   :1;
                unsigned _ACBG  :1;
                unsigned _ACD   :1;
        };
        struct {
                unsigned _w     :8;
        };
} __ACSRbits_t;

/**
 * MONDR / OCDR
 */
typedef union {
```

```
        struct {
                unsigned _OCDR0 :1;
                unsigned _OCDR1 :1;
                unsigned _OCDR2 :1;
                unsigned _OCDR3 :1;
                unsigned _OCDR4 :1;
                unsigned _OCDR5 :1;
                unsigned _OCDR6 :1;
                unsigned _OCDR7 :1;
        };
        struct {
                unsigned        :6;
                unsigned _IDRD  :1;
        };
        struct {
                unsigned _w     :8;
        };
} __OCDRbits_t;

/**
 * SMCR
 */
typedef union {
        struct {
                unsigned _SE    :1;
                unsigned _SM0   :1;
                unsigned _SM1   :1;
                unsigned _SM2   :1;
                unsigned        :4;
        };
        struct {
                unsigned        :1;
                unsigned _SM    :3;
                unsigned        :5;
        };
        struct {
                unsigned _w     :8;
        };
} __SMCRbits_t;

/**
 * MCUSR
 */
typedef union {
        struct {
                unsigned _PORF  :1;
                unsigned _EXTRF :1;
                unsigned _BORF  :1;
                unsigned _WDRF  :1;
                unsigned _JTRF  :1;
```

```
                unsigned        :3;
        };
        struct {
                unsigned _w     :8;
        };
} __MCUSRbits_t;

/**
 * MCUCR
 */
typedef union {
        struct {
                unsigned _IVCE  :1;
                unsigned _IVSEL :1;
                unsigned        :2;
                unsigned _PUD   :1;
                unsigned _BODSE :1;
                unsigned _BODS  :1;
                unsigned _JTD   :1;
        };
        struct {
                unsigned _w     :8;
        };
} __MCUCRbits_t;

/**
 * SPMCSR
 */
typedef union {
        struct {
                unsigned _SPMEN         :1;
                unsigned _PGERS         :1;
                unsigned _PGWRT         :1;
                unsigned _BLBSET        :1;
                unsigned _RWWSRE        :1;
                unsigned _SIGRD         :1;
                unsigned _RWWSB         :1;
                unsigned _SPMIE         :1;
        };
        struct {
                unsigned _w     :8;
        };
} __SPMCSRbits_t;

/**
 * WDTCSR
 */
typedef union {
        struct {
                unsigned _WDP0  :1;
```

```
                unsigned _WDP1  :1;
                unsigned _WDP2  :1;
                unsigned _WDE   :1;
                unsigned _WDCE  :1;
                unsigned _WDP3  :1;
                unsigned _WDIE  :1;
                unsigned _WDIF  :1;
        };
        struct {
                unsigned _WDP   :3;
                unsigned        :5;
        };
        struct {
                unsigned _w     :8;
        };
} __WDTCSRbits_t;

/**
 * CLKPR
 */
typedef union {
        struct {
                unsigned _CLKPS0        :1;
                unsigned _CLKPS1        :1;
                unsigned _CLKPS2        :1;
                unsigned _CLKPS3        :1;
                unsigned                :3;
                unsigned _CLKPCE        :1;
        };
        struct {
                unsigned _CLKPS :4;
                unsigned        :4;
        };
        struct {
                unsigned _w     :8;
        };
} __CLKPRbits_t;

/**
 * PRR
 */
typedef union {
        struct {
                unsigned _PRADC         :1;
                unsigned _PRUSART0      :1;
                unsigned _PRSPI         :1;
                unsigned _PRTIM1        :1;
                unsigned _PRUSART1      :1;
                unsigned _PRTIM0        :1;
                unsigned _PRTIM2        :1;
```

```
                unsigned _PRTWI         :1;
        };
        struct {
                unsigned _w     :8;
        };
} __PRRbits_t;

/**
 * OSCCAL
 * 8bitreg
 */

/**
 * PCICR
 */
typedef union {
        struct {
                unsigned _PCIE0 :1;
                unsigned _PCIE1 :1;
                unsigned _PCIE2 :1;
                unsigned _PCIE3 :1;
                unsigned        :4;
        };
        struct {
                unsigned _PCIE  :4;
                unsigned        :4;
        };
        struct {
                unsigned _w     :8;
        };
} __PCICRbits_t;

/**
 * EICRA
 */
typedef union {
        struct {
                unsigned _ISC00 :1;
                unsigned _ISC01 :1;
                unsigned _ISC10 :1;
                unsigned _ISC11 :1;
                unsigned _ISC20 :1;
                unsigned _ISC21 :1;
                unsigned        :2;
        };
        struct {
                unsigned _ISC0  :2;
                unsigned _ISC1  :2;
                unsigned _ISC2  :2;
                unsigned        :2;
```

```
        };
        struct {
                unsigned _w     :8;
        };
} __EICRAbits_t;

/**
 * PCMSK0
 */
typedef union {
        struct {
                unsigned _PCINT0        :1;
                unsigned _PCINT1        :1;
                unsigned _PCINT2        :1;
                unsigned _PCINT3        :1;
                unsigned _PCINT4        :1;
                unsigned _PCINT5        :1;
                unsigned _PCINT6        :1;
                unsigned _PCINT7        :1;
        };
        struct {
                unsigned _w     :8;
        };
} __PCMSK0bits_t;

/**
 * PCMSK1
 */
typedef union {
        struct {
                unsigned _PCINT8        :1;
                unsigned _PCINT9        :1;
                unsigned _PCINT1        0:1;
                unsigned _PCINT1        1:1;
                unsigned _PCINT1        2:1;
                unsigned _PCINT1        3:1;
                unsigned _PCINT1        4:1;
                unsigned _PCINT1        5:1;
        };
        struct {
                unsigned _w     :8;
        };
} __PCMSK1bits_t;

/**
 * PCMSK2
 */
typedef union {
        struct {
                unsigned _PCINT16       :1;
```

```
                unsigned _PCINT17       :1;
                unsigned _PCINT18       :1;
                unsigned _PCINT19       :1;
                unsigned _PCINT20       :1;
                unsigned _PCINT21       :1;
                unsigned _PCINT22       :1;
                unsigned _PCINT23       :1;
        };
        struct {
                unsigned _w     :8;
        };
} __PCMSK2bits_t;

/**
 * TIMSK0
 */
typedef union {
        struct {
                unsigned _TOIE0         :1;
                unsigned _OCIE0A        :1;
                unsigned _OCIE0B        :1;
                unsigned                :5;
        };
        struct {
                unsigned _w     :8;
        };
} __TIMSK0bits_t;

/**
 * TIMSK1
 */
typedef union {
        struct {
                unsigned _TOIE1         :1;
                unsigned _OCIE1A        :1;
                unsigned _OCIE1B        :1;
                unsigned                :2;
                unsigned _ICIE1         :1;
                unsigned                :2;
        };
        struct {
                unsigned _w     :8;
        };
} __TIMSK1bits_t;

/**
 * TIMSK2
 */
typedef union {
        struct {
```

```
                unsigned _TOIE2         :1;
                unsigned _OCIE2A        :1;
                unsigned _OCIE2B        :1;
                unsigned                :5;
        };
        struct {
                unsigned _w     :8;
        };
} __TIMSK2bits_t;

/**
 * PCMSK3
 */
typedef union {
        struct {
                unsigned _PCINT24       :1;
                unsigned _PCINT25       :1;
                unsigned _PCINT26       :1;
                unsigned _PCINT27       :1;
                unsigned _PCINT28       :1;
                unsigned _PCINT29       :1;
                unsigned _PCINT30       :1;
                unsigned _PCINT31       :1;
        };
        struct {
                unsigned _w     :8;
        };
} __PCMSK3bits_t;

/**
 * ADCW
 * 16bitreg
 */

/**
 * ADCL
 * 8bitreg
 */

/**
 * ADCH
 * 8bitreg
 */

/**
 * ADCSRA
 */
typedef union {
        struct {
                unsigned _ADPS0 :1;
```

```
                unsigned _ADPS1 :1;
                unsigned _ADPS2 :1;
                unsigned _ADIE  :1;
                unsigned _ADIF  :1;
                unsigned _ADATE :1;
                unsigned _ADSC  :1;
                unsigned _ADEN  :1;
        };
        struct {
                unsigned _ADPS  :3;
                unsigned        :5;
        };
        struct {
                unsigned _w     :8;
        };
} __ADCSRAbits_t;

/**
 * ADCSRB
 */
typedef union {
        struct {
                unsigned _ADTS0 :1;
                unsigned _ADTS1 :1;
                unsigned _ADTS2 :1;
                unsigned        :3;
                unsigned _ACME  :1;
                unsigned        :1;
        };
        struct {
                unsigned _ADTS  :3;
                unsigned        :5;
        };
        struct {
                unsigned _w     :8;
        };
} __ADCSRBbits_t;

/**
 * ADMUX
 */
typedef union {
        struct {
                unsigned _MUX0  :1;
                unsigned _MUX1  :1;
                unsigned _MUX2  :1;
                unsigned _MUX3  :1;
                unsigned _MUX4  :1;
                unsigned _ADLAR :1;
                unsigned _REFS0 :1;
```

```
                unsigned _REFS1 :1;
        };
        struct {
                unsigned _MUX   :5;
                unsigned        :1;
                unsigned _REFS  :2;
        };
        struct {
                unsigned _w     :8;
        };
} __ADMUXbits_t;

/**
 * DIDR0
 */
typedef union {
        struct {
                unsigned _ADC0D :1;
                unsigned _ADC1D :1;
                unsigned _ADC2D :1;
                unsigned _ADC3D :1;
                unsigned _ADC4D :1;
                unsigned _ADC5D :1;
                unsigned _ADC6D :1;
                unsigned _ADC7D :1;
        };
        struct {
                unsigned _w     :8;
        };
} __DIDR0bits_t;

/**
 * DIDR1
 */
typedef union {
        struct {
                unsigned _AIN0D :1;
                unsigned _AIN1D :1;
                unsigned        :6;
        };
        struct {
                unsigned _w     :8;
        };
} __DIDR1bits_t;

/**
 * TCCR1A
 */
typedef union {
        struct {
```

```
                unsigned _WGM10         :1;
                unsigned _WGM11         :1;
                unsigned                :2;
                unsigned _COM1B0        :1;
                unsigned _COM1B1        :1;
                unsigned _COM1A0        :1;
                unsigned _COM1A1        :1;
        };
        struct {
                unsigned _WGM   :2;
                unsigned        :2;
                unsigned _COMB  :2;
                unsigned _COMA  :2;
        };
        struct {
                unsigned        :4;
                unsigned _COM1B :2;
                unsigned _COM1A :2;
        };
        struct {
                unsigned _w     :8;
        };
} __TCCR1Abits_t;

/**
 * TCCR1B
 */
typedef union {
        struct {
                unsigned _CS10  :1;
                unsigned _CS11  :1;
                unsigned _CS12  :1;
                unsigned _WGM12 :1;
                unsigned _WGM13 :1;
                unsigned        :1;
                unsigned _ICES1 :1;
                unsigned _ICNC1 :1;
        };
        struct {
                unsigned _CS    :3;
                unsigned _WGM   :2;
                unsigned        :3;
        };
        struct {
                unsigned _w     :8;
        };
} __TCCR1Bbits_t;

/**
 * TCCR1C
```

```
 */
typedef union {
        struct {
                unsigned        :6;
                unsigned _FOC1A :1;
                unsigned _FOC1B :1;
        };
        struct {
                unsigned        :6;
                unsigned _FOC   :2;
        };
        struct {
                unsigned _w     :8;
        };
} __TCCR1Cbits_t;

/**
 * TCNT1
 * 16bitreg
 */

/**
 * TCNT1H
 * 8bitreg
 */

/**
 * TCNT1L
 * 8bitreg
 */

/**
 * ICR1
 * 16bitreg
 */

/**
 * ICR1H
 * 8bitreg
 */

/**
 * ICR1L8bitreg
 */

/**
 * OCR1A
 * 16bitreg
 */
```

```
/**
 * OCR1AH
 * 8bitreg
 */

/**
 * OCR1AL
 * 8bitreg
 */

/**
 * OCR1B
 * 16bitreg
 */

/**
 * OCR1H
 * 8bitreg
 */

/**
 * OCR1BL
 * 8bitreg
 */

/**
 * TCCR2A
 */
typedef union {
        struct {
                unsigned _WGM20          :1;
                unsigned _WGM21          :1;
                unsigned                 :2;
                unsigned _COM2B0         :1;
                unsigned _COM2B1         :1;
                unsigned _COM2A0         :1;
                unsigned _COM2A1         :1;
        };
        struct {
                unsigned _WGM            :2;
                unsigned                 :2;
                unsigned _COMB           :2;
                unsigned _COMA           :2;
        };
        struct {
                unsigned         :4;
                unsigned _COM1B  :2;
                unsigned _COM1A  :2;
        };
        struct {
```

```
                unsigned _w     :8;
        };
} __TCCR2Abits_t;

/**
 * TCCR2B
 */
typedef union {
        struct {
                unsigned _CS20  :1;
                unsigned _CS21  :1;
                unsigned _CS22  :1;
                unsigned _WGM22 :1;
                unsigned        :2;
                unsigned _FOC2B :1;
                unsigned _FOC2A :1;
        };
        struct {
                unsigned _CS    :3;
                unsigned _WGM   :1;
                unsigned        :1;
                unsigned _FOC   :2;
        };
        struct {
                unsigned _w     :8;
        };
} __TCCR2Bbits_t;

/**
 * TCNT2
 * 8bitreg
 */

/**
 * OCR2A
 * 8bitreg
 */

/**
 * OCR2B
 * 8bitreg
 */

/**
 * ASSR
 */
typedef union {
        struct {
                unsigned _TCR2BUB       :1;
                unsigned _TCR2AUB       :1;
```

```
                unsigned _OCR2BUB       :1;
                unsigned _OCR2AUB       :1;
                unsigned _TCN2UB        :1;
                unsigned _AS2           :1;
                unsigned _EXCLK         :1;
                unsigned                :1;
        };
        struct {
                unsigned _w     :8;
        };
} __ASSRbits_t;

/**
 * TWBR
 * 8bitreg
 */

/**
 * TWSR
 */
typedef union {
        struct {
                unsigned _TWPS0 :1;
                unsigned _TWPS1 :1;
                unsigned        :1;
                unsigned _TWS3  :1;
                unsigned _TWS4  :1;
                unsigned _TWS5  :1;
                unsigned _TWS6  :1;
                unsigned _TWS7  :1;
        };
        struct {
                unsigned _TWPS  :2;
                unsigned        :1;
                unsigned _TWS   :5;
        };
        struct {
                unsigned _w     :8;
        };
} __TWSRbits_t;

/**
 * TWAR
 */
typedef union {
        struct {
                unsigned _TWGCE :1;
                unsigned _TWA0  :1;
                unsigned _TWA1  :1;
                unsigned _TWA2  :1;
```

```
                unsigned _TWA3  :1;
                unsigned _TWA4  :1;
                unsigned _TWA5  :1;
                unsigned _TWA6  :1;
        };
        struct {
                unsigned        :1;
                unsigned _TWA   :7;
        };
        struct {
                unsigned _w     :8;
        };
} __TWARbits_t;

/**
 * TWDR
 * 8bitreg
 */

/**
 * TWCR
 */
typedef union {
        struct {
                unsigned _TWIE  :1;
                unsigned        :1;
                unsigned _TWEN  :1;
                unsigned _TWWC  :1;
                unsigned _TWSTO :1;
                unsigned _TWSTA :1;
                unsigned _TWEA  :1;
                unsigned _TWINT :1;
        };
        struct {
                unsigned _w     :8;
        };
} __TWCRbits_t;

/**
 * TWAMR
 */
typedef union {
        struct {
                unsigned _TWAM0 :1;
                unsigned _TWAM1 :1;
                unsigned _TWAM2 :1;
                unsigned _TWAM3 :1;
                unsigned _TWAM4 :1;
                unsigned _TWAM5 :1;
                unsigned _TWAM6 :1;
```

```
                unsigned _TWAM7 :1;
        };
        struct {
                unsigned _w     :8;
        };
} __TWAMRbits_t;

/**
 * UCSR0A
 */
typedef union {
        struct {
                unsigned _MPCM0 :1;
                unsigned _U2X0  :1;
                unsigned _UPE0  :1;
                unsigned _DOR0  :1;
                unsigned _FE0   :1;
                unsigned _UDRE0 :1;
                unsigned _TXC0  :1;
                unsigned _RXC0  :1;
        };
        struct {
                unsigned _w     :8;
        };
} __UCSR0Abits_t;

/**
 * UCSR0B
 */
typedef union {
        struct {
                unsigned _TXB80         :1;
                unsigned _RXB80         :1;
                unsigned _UCSZ02        :1;
                unsigned _TXEN0         :1;
                unsigned _RXEN0         :1;
                unsigned _UDRIE0        :1;
                unsigned _TXCIE0        :1;
                unsigned _RXCIE0        :1;
        };
        struct {
                unsigned w      :8;
        };
} __UCSR0Bbits_t;

/**
 * UCSR0C
 */
typedef union {
        struct {
```

```
                unsigned _UCPOL0        :1;
                unsigned _UCSZ00        :1;
                unsigned _UCSZ01        :1;
                unsigned _USBS0         :1;
                unsigned _UPM00         :1;
                unsigned _UPM01         :1;
                unsigned _UMSEL00       :1;
                unsigned _UMSEL01       :1;
        };
        struct {
                unsigned        :1;
                unsigned _UCSZ0 :2;
                unsigned        :1;
                unsigned _UPM0  :2;
                unsigned _UMSEL0:2;
        };
        struct {
                unsigned _w     :8;
        };
} __UCSR0Cbits_t;

/**
 * UBRR0
 * 16bitreg
 */

/**
 * UBRR0H
 * 8bitreg
 */

/**
 * UBRR0L
 * 8bitreg
 */

/**
 * UDR0
 * 8bitreg
 */

/**
 * UCSR1A
 */
typedef union {
        struct {
                unsigned _MPCM1 :1;
                unsigned _U2X1  :1;
                unsigned _UPE1  :1;
                unsigned _DOR1  :1;
```

```
                unsigned _FE1   :1;
                unsigned _UDRE1 :1;
                unsigned _TXC1  :1;
                unsigned _RXC1  :1;
        };
        struct {
                unsigned _w     :8;
        };
} __UCSR1Abits_t;

/**
 * UCSR1B
 */
typedef union {
        struct {
                unsigned _TXB81         :1;
                unsigned _RXB81         :1;
                unsigned _UCSZ12        :1;
                unsigned _TXEN1         :1;
                unsigned _RXEN1         :1;
                unsigned _UDRIE1        :1;
                unsigned _TXCIE1        :1;
                unsigned _RXCIE1        :1;
        };
        struct {
                unsigned _w     :8;
        };
} __UCSR1Bbits_t;

/**
 * UCSR1C
 */
typedef union {
        struct {
                unsigned _UCPOL1        :1;
                unsigned _UCSZ10        :1;
                unsigned _UCSZ11        :1;
                unsigned _USBS1         :1;
                unsigned _UPM10         :1;
                unsigned _UPM11         :1;
                unsigned _UMSEL10       :1;
                unsigned _UMSEL11       :1;
        };
        struct {
                unsigned                :1;
                unsigned _UCSZ1         :2;
                unsigned                :1;
                unsigned _UPM1          :2;
                unsigned _UMSEL1        :2;
        };
```

```
        struct {
                unsigned _w     :8;
        };
} __UCSR1Cbits_t;

/**
 * UBRR1
 * 16bitreg
 */

/**
 * UBRR1H
 * 8bitreg
 */

/**
 * UBRR1L
 * 8bitreg
 */

/**
 * UDR1
 * 8bitreg
 */

extern volatile __8bitreg_t     PINAbits
  __asm__ ("0x20") __attribute__((section("sfr")));
extern volatile __8bitreg_t     DDRAbits
  __asm__ ("0x21") __attribute__((section("sfr")));
extern volatile __8bitreg_t     PORTAbits
  __asm__ ("0x22") __attribute__((section("sfr")));
extern volatile __8bitreg_t     PINBbits
  __asm__ ("0x23") __attribute__((section("sfr")));
extern volatile __8bitreg_t     DDRBbits
  __asm__ ("0x24") __attribute__((section("sfr")));
extern volatile __8bitreg_t     PORTBbits
  __asm__ ("0x25") __attribute__((section("sfr")));
extern volatile __8bitreg_t     PINCbits
  __asm__ ("0x26") __attribute__((section("sfr")));
extern volatile __8bitreg_t     DDRCbits
  __asm__ ("0x27") __attribute__((section("sfr")));
extern volatile __8bitreg_t     PORTCbits
  __asm__ ("0x28") __attribute__((section("sfr")));
extern volatile __8bitreg_t     PINDbits
  __asm__ ("0x29") __attribute__((section("sfr")));
extern volatile __8bitreg_t     DDRDbits
    __asm__ ("0x2A") __attribute__((section("sfr")));
extern volatile __8bitreg_t     PORTDbits
    __asm__ ("0x2B") __attribute__((section("sfr")));
extern volatile __8bitreg_t     ADCPINbits
```

```
    __asm__ ("0x20") __attribute__((section("sfr")));
extern volatile __8bitreg_t     ADCDDRbits
    __asm__ ("0x21") __attribute__((section("sfr")));
extern volatile __8bitreg_t     ADCPORTbits
    __asm__ ("0x22") __attribute__((section("sfr")));
extern volatile __SPIPORTbits_t SPIDDRbits
    __asm__ ("0x24") __attribute__((section("sfr")));
extern volatile __SPIPORTbits_t SPIPORTbits
    __asm__ ("0x25") __attribute__((section("sfr")));
extern volatile __TIFR0bits_t   TIFR0bits
    __asm__ ("0x35") __attribute__((section("sfr")));
extern volatile __TIFR1bits_t   TIFR1bits
    __asm__ ("0x36") __attribute__((section("sfr")));
extern volatile __TIFR2bits_t   TIFR2bits
    __asm__ ("0x37") __attribute__((section("sfr")));
extern volatile __PCIFRbits_t   PCIFRbits
    __asm__ ("0x3B") __attribute__((section("sfr")));
extern volatile __EIFRbits_t    EIFRits
    __asm__ ("0x3C") __attribute__((section("sfr")));
extern volatile __EIMSKbits_t   EIMSKbits
    __asm__ ("0x3D") __attribute__((section("sfr")));
extern volatile __GPIOR0bits_t  GPIOR0bits
    __asm__ ("0x3E") __attribute__((section("sfr")));
extern volatile __EECRbits_t    EECRbits
    __asm__ ("0x3F") __attribute__((section("sfr")));
extern volatile __8bitreg_t     EEDRbits
    __asm__ ("0x40") __attribute__((section("sfr")));
extern volatile __16bitreg_t    EEARbits
    __asm__ ("0x41") __attribute__((section("sfr")));
extern volatile __8bitreg_t     EEARLbits
    __asm__ ("0x41") __attribute__((section("sfr")));
extern volatile __8bitreg_t     EEARHbits
    __asm__ ("0x42") __attribute__((section("sfr")));
extern volatile __GTCCRbits_t   GTCCRbits
    __asm__ ("0x43") __attribute__((section("sfr")));
extern volatile __TCCR0Abits_t  TCCR0Abits
    __asm__ ("0x44") __attribute__((section("sfr")));
extern volatile __TCCR0Bbits_t  TCCR0Bbits
    __asm__ ("0x45") __attribute__((section("sfr")));
extern volatile __8bitreg_t     TCNT0bits
    __asm__ ("0x46") __attribute__((section("sfr")));
extern volatile __8bitreg_t     OCR0Abits
    __asm__ ("0x47") __attribute__((section("sfr")));
extern volatile __8bitreg_t     OCR0Bbits
    __asm__ ("0x48") __attribute__((section("sfr")));
extern volatile __8bitreg_t     GPIOR1bits
    __asm__ ("0x4A") __attribute__((section("sfr")));
extern volatile __8bitreg_t     GPIOR2bits
    __asm__ ("0x4B") __attribute__((section("sfr")));
extern volatile __SPCRbits_t    SPCRbits
```

```
    __asm__ ("0x4C") __attribute__((section("sfr")));
extern volatile __SPSRbits_t    SPSRbits
    __asm__ ("0x4D") __attribute__((section("sfr")));
extern volatile __8bitreg_t     SPDRbits
    __asm__ ("0x4E") __attribute__((section("sfr")));
extern volatile __ACSRbits_t    ACSRbits
    __asm__ ("0x50") __attribute__((section("sfr")));
extern volatile __OCDRbits_t    OCDRbits
    __asm__ ("0x51") __attribute__((section("sfr")));
#define         MONDRbits               OCDRbits
#define         __MONDRbits_t   __OCDRbits_t
extern volatile __SMCRbits_t    SMCRbits
    __asm__ ("0x53") __attribute__((section("sfr")));
extern volatile __MCUSRbits_t   MCUSRbits
    __asm__ ("0x54") __attribute__((section("sfr")));
extern volatile __MCUCRbits_t   MCUCRbits
    __asm__ ("0x55") __attribute__((section("sfr")));
extern volatile __SPMCSRbits_t  SPMCSRbits
    __asm__ ("0x57") __attribute__((section("sfr")));

extern volatile __WDTCSRbits_t  WDTCSRbits
    __asm__ ("0x60") __attribute__((section("sfr")));
extern volatile __CLKPRbits_t   CLKPRbits
    __asm__ ("0x61") __attribute__((section("sfr")));
extern volatile __PRRbits_t     PRRbits
    __asm__ ("0x64") __attribute__((section("sfr")));
extern volatile __8bitreg_t     OSCCALbits
    __asm__ ("0x66") __attribute__((section("sfr")));
extern volatile __PCICRbits_t   PCICRbits
    __asm__ ("0x68") __attribute__((section("sfr")));
extern volatile __EICRAbits_t   EICRAbits
    __asm__ ("0x69") __attribute__((section("sfr")));
extern volatile __PCMSK0bits_t  PCMSK0bits
    __asm__ ("0x6B") __attribute__((section("sfr")));
extern volatile __PCMSK1bits_t  PCMSK1bits
    __asm__ ("0x6C") __attribute__((section("sfr")));
extern volatile __PCMSK2bits_t  PCMSK2bits
    __asm__ ("0x6D") __attribute__((section("sfr")));
extern volatile __TIMSK0bits_t  TIMSK0bits
    __asm__ ("0x6E") __attribute__((section("sfr")));
extern volatile __TIMSK1bits_t  TIMSK1bits
    __asm__ ("0x6F") __attribute__((section("sfr")));
extern volatile __TIMSK2bits_t  TIMSK2bits
    __asm__ ("0x70") __attribute__((section("sfr")));
extern volatile __PCMSK3bits_t  PCMSK3bits
    __asm__ ("0x73") __attribute__((section("sfr")));
extern volatile __16bitreg_t    ADCWbits
    __asm__ ("0x78") __attribute__((section("sfr")));
extern volatile __8bitreg_t     ADCLbits
    __asm__ ("0x78") __attribute__((section("sfr")));
```

```
extern volatile __8bitreg_t     ADCHbits
    __asm__ ("0x79") __attribute__((section("sfr")));
extern volatile __ADCSRAbits_t  ADCSRAbits
    __asm__ ("0x7A") __attribute__((section("sfr")));
extern volatile __ADCSRBbits_t  ADCSRBbits
    __asm__ ("0x7B") __attribute__((section("sfr")));
extern volatile __ADMUXbits_t   ADMUXbits
    __asm__ ("0x7C") __attribute__((section("sfr")));
extern volatile __DIDR0bits_t   DIDR0bits
    __asm__ ("0x7E") __attribute__((section("sfr")));
extern volatile __DIDR1bits_t   DIDR1bits
    __asm__ ("0x7F") __attribute__((section("sfr")));
extern volatile __TCCR1Abits_t  TCCR1Abits
    __asm__ ("0x80") __attribute__((section("sfr")));
extern volatile __TCCR1Bbits_t  TCCR1Bbits
    __asm__ ("0x81") __attribute__((section("sfr")));
extern volatile __TCCR1Cbits_t  TCCR1Cbits
    __asm__ ("0x82") __attribute__((section("sfr")));
extern volatile __16bitreg_t    TCNT1bits
    __asm__ ("0x84") __attribute__((section("sfr")));
extern volatile __8bitreg_t     TCNT1Hbits
    __asm__ ("0x84") __attribute__((section("sfr")));
extern volatile __8bitreg_t     TCNT1Lbits
    __asm__ ("0x85") __attribute__((section("sfr")));
extern volatile __16bitreg_t    ICR1bits
    __asm__ ("0x86") __attribute__((section("sfr")));
extern volatile __8bitreg_t     ICR1Hbits
    __asm__ ("0x86") __attribute__((section("sfr")));
extern volatile __8bitreg_t     ICR1Lbits
    __asm__ ("0x87") __attribute__((section("sfr")));
extern volatile __16bitreg_t    OCR1Abits
    __asm__ ("0x88") __attribute__((section("sfr")));
extern volatile __8bitreg_t     OCR1AHbits
    __asm__ ("0x88") __attribute__((section("sfr")));
extern volatile __8bitreg_t     OCR1ALbits
    __asm__ ("0x89") __attribute__((section("sfr")));
extern volatile __16bitreg_t    OCR1Bbits
    __asm__ ("0x8A") __attribute__((section("sfr")));
extern volatile __8bitreg_t     OCR1BHbits
    __asm__ ("0x8A") __attribute__((section("sfr")));
extern volatile __8bitreg_t     OCR1BLbits
    __asm__ ("0x8B") __attribute__((section("sfr")));
extern volatile __TCCR2Abits_t  TCCR2Abits
    __asm__ ("0xB0") __attribute__((section("sfr")));
extern volatile __TCCR2Bbits_t  TCCR2Bbits
    __asm__ ("0xB1") __attribute__((section("sfr")));
extern volatile __8bitreg_t     TCNT2bits
    __asm__ ("0xB2") __attribute__((section("sfr")));
extern volatile __8bitreg_t     OCR2Abits
    __asm__ ("0xB3") __attribute__((section("sfr")));
```

```
extern volatile __8bitreg_t     OCR2Bbits
    __asm__ ("0xB4") __attribute__((section("sfr")));
extern volatile __ASSRbits_t    ASSRbits
    __asm__ ("0xB6") __attribute__((section("sfr")));
extern volatile __8bitreg_t     TWBRbits
    __asm__ ("0xB8") __attribute__((section("sfr")));
extern volatile __TWSRbits_t    TWSRbits
    __asm__ ("0xB9") __attribute__((section("sfr")));
extern volatile __TWARbits_t    TWARbits
    __asm__ ("0xBA") __attribute__((section("sfr")));
extern volatile __8bitreg_t     TWDRbits
    __asm__ ("0xBB") __attribute__((section("sfr")));
extern volatile __TWCRbits_t    TWCRbits
    __asm__ ("0xBC") __attribute__((section("sfr")));
extern volatile __TWAMRbits_t   TWAMRbits
    __asm__ ("0xBD") __attribute__((section("sfr")));
extern volatile __UCSR0Abits_t  UCSR0Abits
    __asm__ ("0xC0") __attribute__((section("sfr")));
extern volatile __UCSR0Bbits_t  UCSR0Bbits
    __asm__ ("0xC1") __attribute__((section("sfr")));
extern volatile __UCSR0Cbits_t  UCSR0Cbits
    __asm__ ("0xC2") __attribute__((section("sfr")));
extern volatile __16bitreg_t    UBRR0bits
    __asm__ ("0xC4") __attribute__((section("sfr")));
extern volatile __8bitreg_t     UBRR0Lbits
    __asm__ ("0xC4") __attribute__((section("sfr")));
extern volatile __8bitreg_t     UBRR0Hbits
    __asm__ ("0xC5") __attribute__((section("sfr")));
extern volatile __8bitreg_t     UDR0bits
    __asm__ ("0xC6") __attribute__((section("sfr")));
extern volatile __UCSR1Abits_t  UCSR1Abits
    __asm__ ("0xC8") __attribute__((section("sfr")));
extern volatile __UCSR1Bbits_t  UCSR1Bbits
    __asm__ ("0xC9") __attribute__((section("sfr")));
extern volatile __UCSR1Cbits_t  UCSR1Cbits
    __asm__ ("0xCA") __attribute__((section("sfr")));
extern volatile __16bitreg_t    UBRR1bits
    __asm__ ("0xCC") __attribute__((section("sfr")));
extern volatile __8bitreg_t     UBRR1Hbits
    __asm__ ("0xCC") __attribute__((section("sfr")));
extern volatile __8bitreg_t     UBRR1Lbits
    __asm__ ("0xCD") __attribute__((section("sfr")));
extern volatile __8bitreg_t     UDR1bits
    __asm__ ("0xCE") __attribute__((section("sfr")));


#endif /*REG_STRUCTS_H_*/
```

Appendix B

Code for final RTOS example

B.1 RTOSmain.h

```
/*
 * RTOSmain.h
 *
 *  Created on: Feb 26, 2011
 *      Author: alleypj
 */

#ifndef RTOSMAIN_H_
#define RTOSMAIN_H_

#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include "tasks.h"
#include "Peripherals\reg_structs.h"

char RTOScounter;   //track what task is next
long time = 0;

//IR reading and distance storage
//196 = 1 foot
int fIR = 196;
int bIR = 196;
int set_point = 196;

//USART tx and rx data buffers and current locations
int tx_data[20];
unsigned char tx_bytes = 0;  //track next byte location
```

```
char rx_data[50];
unsigned char rx_bytes = 0;  //track next byte location

#endif
```

B.2 RTOSmain.c

```
/*
 * RTOSmain.c
 *
 *  Created on: Jan 21, 2011
 *      Author: alleypj
 */

#include "RTOSmain.h"

int main (void) {
  initialization_functions(); //including initRTOS()
  sei();
  time = 0; //Time in ms since RTOS started
  RTOScounter = 0;
  char taskID = 0;
  TCNT2 = 0;
  while(1) {
    taskID = RTOScounter;
    switch(RTOScounter) {
    case 0:
      IMUReadingTask();
      break;
    case 1:
      break;
    case 2:
      CommunicationTask();
      break;
    case 3:
      break;
    case 4:
      DriveMotorControlTask();
      break;
    case 5:
      break;
    case 6:
      PotServoTask();
      break;
    case 7:
      break;
    case 8:
      CommunicationTask();
      break;
    case 9:
      break;
    case 10:
      break;
    case 11:
```

```
      break;
    case 12:
      IMUReadingTask();
      break;
    case 13:
      break;
    case 14:
      IRTask();
      break;
    case 15:
      CommunicationTask();
      break;
    case 16:
      break;
    case 17:
      break;
    case 18:
      break;
    case 19:
      break;
    }
    time++;
    while(RTOScounter == taskID){}
    //this loop prevents a task from running twice during
    //its block.  Waits until the RTOSCounter changes
    //before continuing
  }
}

/** \ISR
 * \brief saves incoming data to array for later perusal
 */
ISR(USART0_RX_vect) {
    rx_data[rx_bytes] = (char)UDR0bits._w;
    rx_bytes++;
}

/** \ISR
 * \brief Increments RTOS task
 */
ISR(TIMER2_COMPA_vect){
    RTOScounter = (RTOScounter+1)%25;
}
```

B.3 tasks.h

```
/*
 * tasks.h
 *
 *  Created on: Feb 26, 2011
 *      Author: alleypj
 */

#ifndef TASKS_H_
#define TASKS_H_

#include "motor.h"
#include "Peripherals\ADC.h"
#include "Peripherals\USART.h"

/*
 * \def TURN_TUNE
 *  tuning constant for motor control task
 */
#define TURN_TUNE 2

extern int fIR;
extern int bIR;
extern int set_point;

extern int tx_data[20];
extern unsigned char tx_bytes;
extern char rx_data[50];
extern unsigned char rx_bytes;
extern long time;

void initialization_functions();
void initRTOS();
void IMUReadingTask();
void CommunicationTask();
void DriveMotorControlTask();
void IRTask();
void PotServoTask();


#endif /* TASKS_H_ */
```

B.4 tasks.c

```
/*
 * tasks.c
 *
 *  Created on: Feb 26, 2011
 *      Author: alleypj
 */

#include "tasks.h"
#include <util/delay.h>
#include <avr/interrupt.h>

// ADC initialization and usage struct.  Defined in ADC.h
ADC_struct_type ADC_struct = {.reference = AREFselect,
        .justify = RIGHTjustify,
        .prescale = ADCPRE64,
        .autotrigger = 0,
        .intenable = 0};

// USART init and usage struct.  Defined in USART.h
USART_struct_type USART_struct = {.baud = 384,
        .intRX = 0,
        .intTX = 0,
        .intDataRegister = 0,
        .enRX = 1,
        .enTX = 1,
        .charsize = 8,
        .mode = ASYNC,
        .parity = NO_PARITY,
        .stopbit = STOP1,
        .clockpolarity = 0};

/** \func initialization_functions
 * \brief runs init functions for all peripherals
 */
void initialization_functions() {
    USART_init(&USART_struct);
    ADC_init(&ADC_struct); //initialize ADC
    motor_init();   //initialize motor
    initRTOS();
}

/** \func initRTOS()
 * \brief Sets up RTOS on timer 2.
 * Initialize and setup RTOS using timer 2.  TCNT2
 * register and RTOS counter should both be reset
 * just prior to entering RTOS loop
 * Sets timer to 1ms interrupts
```

```
 */
void initRTOS() {
    TCCR2Abits._WGM = 2;  //set CTC mode, no output pin
    TCCR2Bbits._CS = 4;  //set prescaler to 64
    OCR2Abits._w = 125;  //125 counts at 8MHz/64 = 1ms
    TIMSK2bits._OCIE2A = 1;  //enable int on compare
}

void IMUReadingTask() {
    //dummy function in example
}

/** \func CommunicationTask
 * \brief runs communications task for RTOS
 * Transmits queued data, then assesses received data.
 * If an error is found in the data, all remaining data
 * is transmitted back with an error indicator so that
 * the remote computer can resend it correctly.
 */
void CommunicationTask(){
    USART_TX(tx_data,tx_bytes);
    tx_bytes = 0;
    unsigned char i = 0;
    //work through received data
    while(i<rx_bytes) {
        //finite state machine for various commands
        switch (rx_data[i]) {
        case 'd': //distance setting
            set_point = ((int)rx_data[i+1]<<8) +
                (int)rx_data[i+2];
            i+=3;
            break;
        default:
//error occurred, send back all remaining data
//uneven command lengths means can't necessarily resync.
//remote controller will have to resend commands
            tx_data[tx_bytes] = 'x'; //error indicator
            tx_bytes++;
            for(i=i;i<rx_bytes;i++) {
                tx_data[tx_bytes] = rx_data[i];
                tx_bytes++;
            }
            USART_TX(tx_data,tx_bytes);
        }
    }
}

/** \func DriveMotorControlTask
 * \brief Sets motor values and gets encoder counts
 * Sets most recently received motor values.  Also gets
```

```
 * counts from both encoder counters and queues data
 * for transmit.
 */
void DriveMotorControlTask(){
    int res = TURN_TUNE*(fIR-bIR)+(fIR+bIR)/2-set_point;

    // set motors
    set_motor(LEFT_MOTOR, 1020+(res<<2));
    set_motor(RIGHT_MOTOR, 3060+(res<<2));
    //left encoder + queue for transmit
    read_encoder(LEFT_MOTOR);
    tx_data[tx_bytes] = 'e'+LEFT_MOTOR; //left encoder
    tx_bytes++;
    int i;
    for (i=0; i<4; i++) {
        tx_data[tx_bytes+i] = SPI_data[i];
    }
    tx_bytes+=4;
    //right encoder and queue for transmit
    read_encoder(RIGHT_MOTOR);
    tx_data[tx_bytes] = 'e'+RIGHT_MOTOR; //right encoder
    tx_bytes++;
    for (i=0; i<4; i++) {
 //       tx_data[tx_bytes+i] = SPI_data[i];
    }
 //   tx_bytes+=4;
}

/** \func IRTask
 * \brief reads one IR sensor or other and transmits
 * Only one IR sensor is read each pass as their update
 * is slower than what the RTOS runs at
 */
void IRTask(){
    tx_data[tx_bytes] = 'i';  //IR data indicator
    if(time%2 == 0) {
        tx_data[tx_bytes] = 'i';  //IR data indicator
        ADC_struct.channel = 0;
        ADC_convert(&ADC_struct);
        fIR = ADC_struct.data;
        tx_data[tx_bytes+1] = 0; //which IR sensor
    } else {
        tx_data[tx_bytes] = 'i'+1;  //IR2 data indicator
        ADC_struct.channel = 1;
        ADC_convert(&ADC_struct);
        bIR = ADC_struct.data;
        tx_data[tx_bytes+1] = 1; //which IR sensor
    }
    tx_data[tx_bytes+2] = (char)(ADC_struct.data>>8);
    tx_data[tx_bytes+3] = (char)(ADC_struct.data%256);
```

```
    tx_bytes+=4;
}
void PotServoTask(){
    //dummy function in example
}
```

B.5 motor.h

```
/*
 * motor.h
 *
 *  Created on: Feb 26, 2011
 *      Author: alleypj
 */

#ifndef MOTOR_H_
#define MOTOR_H_

#include "Peripherals\SPI.h"

/**
 * \def DAC_DEVICE
 *   used to determine which device the SPI is setup for
 * \def ENC_DEVICE
 *   used to determine which device the SPI is setup for
 */
#define DAC_DEVICE 0
#define ENC_DEVICE 1

/**
 * \def LEFT_MOTOR
 *   DAC port for left motor
 * \def RIGHT_MOTOR
 *   DAC port for right motor
 */
#define LEFT_MOTOR 0
#define RIGHT_MOTOR 1

/**
 * \def DAC_SS_DDR
 *   Data direction pin for DAC SS
 * \def DAC_SS
 *   Slave select pin for DAC
 * \def ENC_L_SS_DDR
 *   Data direction for left enc SS
 * \def ENC_L_SS
 *   Slave select pin for left encoder counter
 * \def ENC_R_SS_DDR
 *   Data direction for right enc SS
 * \def ENC_R_SS
 *   Slave select pin for right encoder counter
 */
#define DAC_SS_DDR DDRDbits._P4
#define DAC_SS PORTDbits._P4
#define ENC_L_SS_DDR DDRCbits._P5
```

```
#define ENC_L_SS PORTCbits._P5
#define ENC_R_SS_DDR DDRCbits._P4
#define ENC_R_SS PORTCbits._P4

//saves which device the SPI is setup for
char cur_device;

//location for storing xmit/rec SPI data
char SPI_data[5];

void motor_init();
void read_encoder(char encoder);
void set_motor(char motor, int speed);

#endif /* MOTOR_H_ */
```

B.6 motor.c

```
/*
 * motor.c
 *
 *  Created on: Feb 26, 2011
 *      Author: alleypj
 */

#include "motor.h"

//DAC init struct.  See SPI.h for struct definition
SPI_struct_type DAC_struct = {.intenable = 0,
                .dataorder = MSBfirst,
                .clockpolarity = 1, //leading edge fall
                .clockphase = 0, //sample on leading
                .clockprescale = SPIPRE2};

//Encoder init struct.  See SPI.h for struct definition
SPI_struct_type Enc_struct = {.intenable = 0,
                .dataorder = MSBfirst,
                .clockpolarity = 0,
                .clockphase = 0,
                .clockprescale = SPIPRE2};

/** \func motor_init
 * \brief Initializes SPI, DAC and encoder counter
 */
void motor_init() {
    DAC_SS_DDR = 1;
    ENC_L_SS_DDR = 1;
    ENC_R_SS_DDR = 1;
    SPI_init(&DAC_struct);
    set_motor(LEFT_MOTOR, 1020);
    set_motor(RIGHT_MOTOR, 3060);

    //setup encoder counter
    SPI_init(&Enc_struct);
    cur_device = ENC_DEVICE;
    ENC_L_SS = 1;
    ENC_R_SS = 1; //config both encoders at once
    //MDR0: 0b00 00 00 11: clock/1, asynch, index
    //    disabled, free-running count
    //    4 counts per quad cycle
    SPI_data[0] = 0b10001000;
    SPI_data[1] = 0b00000011;
    //MDR1: 0b0000x000: no flags, counting enabled,
    //    4-byte counter
    SPI_data[2] = 0b10010000;
```

```
    SPI_data[3] = 0b01100000;
    SPI_txrx(SPI_data,4,0);
    ENC_L_SS = 0;
    ENC_R_SS = 0;
}

/** \func read_encoder
 * \brief reads selected encoder
 * \param encoder to read (0/1)
 * \return 32bit encoder value
 */
void read_encoder(char encoder) {
    //setup SPI for encoder
    if(cur_device == DAC_DEVICE) {
        SPI_init(&Enc_struct);
        cur_device = ENC_DEVICE;
    }
    // select correct encoder counter
    switch(encoder) {
    case LEFT_MOTOR:
        ENC_L_SS = 1;
        break;
    case RIGHT_MOTOR:
        ENC_R_SS = 1;
        break;
    }
    //load CNTR to OTR then read OTR:  0b01 100 xxx
    SPI_data[0] = 0b01100000;
    SPI_txrx(SPI_data,1,4);
    ENC_L_SS = 0;
    ENC_R_SS = 0;
}

/** \func set_motor
 * \brief sets the given motor to the given speed
 * \param motor motor to set
 * \param speed speed for motor. 2048 is stationary
 */
void set_motor(char motor, int speed) {
    //setup SPI for DAC
    if(cur_device == ENC_DEVICE) {
        SPI_init(&DAC_struct);
        cur_device = DAC_DEVICE;
    }
    //create top byte:  function, channel and top 4 bits
    SPI_data[0] = 128 + (motor<<4) + (speed>>8);
    //lower data byte:  8 data bites
    SPI_data[1] = speed%256;
    SPI_txrx(SPI_data,2,0);
}
```

B.7 ADC.h

```
/*
 * ADC.h
 *
 *  Created on: Feb 26, 2011
 *      Author: alleypj
 */

#ifndef ADC_H_
#define ADC_H_

#include <avr/io.h>
#include "reg_structs.h"

/**
 * \def ADC_enable()
 *   Enable ADC presented as a function
 * \def ADC_disable()
 *   Disable ADC presented as a function
 */
#define ADC_enable() ADCSRAbits._ADEN = 1
#define ADC_disable() ADCSRAbits._ADEN = 0

/**
 * list of defined values for initialization
 */
//references
#define AREFselect 0
#define AVCCselect 1
#define INT11select 2  //internal 1.1V
#define INT256select 3 //internal 2.56V
//justification
#define RIGHTjustify 0
#define LEFTjustify 1
//Channel selection
#define ADC_GROUND 0b11111
#define VBG 0b11110
//channel numbers can be entered directly
//differential input defines not created yet

//Prescaler bits
#define ADCPRE2 1
#define ADCPRE4 4
#define ADCPRE8 3
#define ADCPRE16 4
#define ADCPRE32 5
#define ADCPRE64 6
#define ADCPRE128 7
```

```
//Auto Trigger Source
//Not implemented currently, use direct values

/**
 * ADC_struct_type
 * Contains options for initializing and using the ADC.
 * Fields will accept values directly from datasheet or
 * can use #defined values.
 */
typedef struct {
    //related to initialization
    char reference;
    char justify;
    char prescale;
    char autotrigger;
    char autosource;
    char disableDIO;
    char intenable;
    //related to use
    char channel; //default to ground
    int data;
} ADC_struct_type;

void ADC_init(ADC_struct_type *init);
void ADC_convert(ADC_struct_type *arg);

#endif /* ADC_H_ */
```

B.8 ADC.c

```
/*
 * ADC.c
 *
 *  Created on: Feb 26, 2011
 *      Author: alleypj
 */

#include "ADC.h"

/** \func ADC_init
 * \brief initializes ADC using given settings
 * \param init struct with settings
 * Also runs first conversion as it is longer than
 * all subsequent conversions.
 */
void ADC_init(ADC_struct_type *init) {
    ADMUXbits._REFS = init->reference;
    ADMUXbits._ADLAR = init->justify;
    ADCSRAbits._ADPS = init->prescale;
    ADCSRAbits._ADATE = init->autotrigger;
    ADCSRBbits._ADTS = init->autosource;
    DIDR0bits._w = init->disableDIO;

    ADC_enable();
    ADMUXbits._MUX = ADC_GROUND;
    ADCSRAbits._ADSC = 1;
    //run first (extra long) conversion
    while(!ADCSRAbits._ADIF) {} //wait for completion
    ADCSRAbits._ADIE = init->intenable;
}

/** \func ADC_convert
 * \brief Runs conversion on specified channel
 * \param arg same as init struct
 * Channel to run the conversion on and location
 * to store result are both within the struct
 */
void ADC_convert(ADC_struct_type *arg) {
    ADMUXbits._MUX = arg->channel;
    ADCSRAbits._ADSC = 1;       //run conversion
    while(!ADCSRAbits._ADIF) {} //wait for completion
    arg->data = ADCWbits._w;
}
```

B.9 SPI.h

```
/*
 * SPI.h
 *
 *  Created on: Feb 26, 2011
 *      Author: alleypj
 */

#ifndef SPI_H_
#define SPI_H_

#include "reg_structs.h"

/**
 * \def SPI_enable()
 *   Enable SPI presented as a function
 * \def SPI_disable()
 *   Disable SPI presented as a function
 */
#define SPI_enable() SPCRbits._SPE = 1
#define SPI_disable() SPCRbits._SPE = 0

/**
 * value definitions for SPI_struct
 */
//data order
#define MSBfirst 0
#define LSBfirst 1
//Clock polarity, clock phase see datasheet
//prescale, includes double speed bit
#define SPIPRE2 4
#define SPIPRE4 0
#define SPIPRE8 5
#define SPIPRE16 1
#define SPIPRE32 6
#define SPIPRE64 2
#define SPIPRE128 3

/**
 * SPI_struct_type
 * Contains options for initializing and using the SPI.
 * Fields will accept values directly from datasheet or
 * can use #defined values.
 */
typedef struct {
    char intenable;
    char dataorder;
    char clockpolarity;
```

```
    char clockphase;
    char clockprescale;
} SPI_struct_type;

void SPI_init(SPI_struct_type *init);
void SPI_txrx(char *data, char txbytes, char rxbytes);
char SPI_txbyte(char data);

#endif /* SPI_H_ */
```

B.10 SPI.c

```
/*
 * SPI.c
 *
 *  Created on: Feb 26, 2011
 *      Author: alleypj
 */

#include "SPI.h"

/**\func SPI_init
 * \brief Initializes SPI peripheral
 * \param init pointer to struct of settings
 */
void SPI_init(SPI_struct_type *init) {
    SPIDDRbits._MOSI = 1;
    SPIDDRbits._MISO = 1;
    SPCRbits._MSTR = 1; //Master mode
    SPCRbits._SPIE = init->intenable;
    SPCRbits._DORD = init->dataorder;
    SPCRbits._CPOL = init->clockpolarity;
    SPCRbits._CPHA = init->clockphase;
    //prescale split up between 2 registers
    SPCRbits._SPR = init->clockprescale%4;
    SPSRbits._SPI2X = init->clockprescale >> 2;
    SPI_enable();
}

/**\func SPI_txtr
 * \brief transmit and receive SPI
 * \param data array of chars to tx/save received data
 * \param txbytes number of bytes to tx (from data)
 * \param rxbytes number of bytes to save after tx
 *
 * Transmits given number of bytes (stored in data), then
 * transmits 0s to capture rxbytes number of chars to
 * save in data.  SS must be chosen prior to this
 * function and cleared after.
 * Does not work with daisy-chained devices.
 */
void SPI_txrx(char data[], char txbytes, char rxbytes) {
    int i = 0;
    for(i=0; i<txbytes; i++) {
        SPI_txbyte(data[i]);
    }
    for(i=0; i<rxbytes; i++) {
        data[i] = SPI_txbyte(0);
    }
```

```
}

/**\func SPI_txbyte
 * \brief transmit/receive 1 byte
 * \param data char to transmit
 * \return received char
 */
char SPI_txbyte(char data) {
    SPDRbits._w = data;
    while(!SPSRbits._SPIF) {} //wait for completion
    return SPDRbits._w;
}
```

B.11 USART.h

```
/*
 * USART.h
 *
 *  Created on: Feb 26, 2011
 *      Author: alleypj
 */

#ifndef USART_H_
#define USART_H_

#include "reg_structs.h"

//mode options, only ASYNCH implemented
#define ASYNC 0
#define SYNC  1    //not fully implemented
#define M_SPI  2    //not at all implemented

// Parity options
#define NO_PARITY 0
#define EVEN_PARITY 2
#define ODD_PARITY 3

//Stop bits
#define STOP1 0
#define STOP2 1

//Clock polarity refer to datasheet

/**
 * USART_struct_type
 * Contains options for initializing and using the USART.
 * Fields will accept values directly from datasheet or
 * can use #defined values.
 */
typedef struct {
    char baud;
    char intRX;
    char intTX;
    char intDataRegister;
    char enRX;
    char enTX;
    char charsize;
    char mode;  //ignored currently
    char parity;
    char stopbit;
    char clockpolarity;
} USART_struct_type;
```

```
void USART_init(USART_struct_type *init);
void USART_TX(int data[], char num_char);
void USART_TX_char(char data);

#endif /* USART_H_ */
```

B.12 USART.c

```
/*
 * USART.c
 *
 *  Created on: Feb 26, 2011
 *      Author: alleypj
 */

#include "USART.h"

/** \func USART_init
 * \brief initialize USART
 * \param pointer to struct of init paramters
 *
 * Baud rate should be given as actual/100
 * (i.e. 9600 would be 96).  Does not currently
 * support synchronous communications
 */
void USART_init(USART_struct_type *init){
    UCSR0Abits._U2X = 1; //double tx speed
    UCSR0Bbits._RXCIE = init->intRX;
    UCSR0Bbits._TXCIE = init->intTX;
    UCSR0Bbits._UDRIE = init->intDataRegister;
    UCSR0Bbits._RXEN = init->enRX;
    UCSR0Bbits._TXEN = init->enTX;
    UCSR0Cbits._UMSEL = 0;
    UCSR0Cbits._UPM = init->parity;
    UCSR0Cbits._USBS = init->stopbit;
    UCSR0Cbits._UCPOL = init->clockpolarity;
    if(init->charsize <= 8) {
        UCSR0Cbits._UCSZ = init->charsize-5;
        UCSR0Bbits._UCSZ = 0;
    } else {
        UCSR0Cbits._UCSZ = 3;
        UCSR0Bbits._UCSZ = 1;
    }
    UBRR0bits._w = F_CPU/(8*init->baud*100)-1;
}

/** \func USART_init
 * \brief Transmits a number of chars
 * \param data array of data to transmit
 * \param num_char number of data to transmit
 *
 * data array must be ints as characters can be 9 bits
 * will not function properly if TX interrupt is enabled
 */
void USART_TX(int data[],char num_char) {
```

```
    int i = 0;
    for(i=0; i<num_char; i++) {
        USART_TX_char(data[i]);
    }
}

/** \func USART_TX_char
 * \brief Transmits a single character
 * \param character to transmit (up to 8 bits)
 * 9th bit not functioning currently
 */
void USART_TX_char(char data) {
    //wait for any previous transmissions
    while(!UCSR0Abits._TXC0) {}
//    UCSR0Bbits._TXB8 = (data>>8) & 1; //9th bit
    UDR0bits._w = data;
}
```

Appendix C

Setting up and using Eclipse

C.1 Set up Eclipse for use with the AVR

This is how to obtain Eclipse and set it up for use in programming the ATmega644p. This can take quite a while (an hour plus) to complete.

1. Download and install the Sun JDK (Java Development Kit). Downloads and instructions can be found at `http://www.oracle.com/technetwork/java/javase/downloads/index.html`.

2. Download Eclipse IDE for C/C++ Developers from `http://www.eclipse.org/downloads/`. At the time of writing, the newest version is 3.6 Helios.

3. Extract the archive file to a destination of your choice. There is no installer so a common location would be `C:`
   `Program Files`
   `Eclipse` or `/usr/bin/eclipse` on a unix based system. Open the standalone executable called `eclipse` or `eclipse.exe` to ensure that it works.

Windows

4. Download and install WinAVR. This is the compiler and headerfiles required for the AVR microcontrollers. Navigate to `http://sourceforge.net/projects/winavr/files/WinAVR/` and select the most recent version. Click on the installer to download. Run the installer upon completion. The defaults should be acceptable, though the install path can be changed if desired. Ensure that "Add directories to PATH" is selected before clicking "Install".

5. Download and install Doxygen. This is the stand alone program that will create documentation from comments in the code (see chapter 5).

`http://ftp.stack.nl/pub/users/dimitri/doxygen-1.6.2-setup.exe`

Linux

4. On a debian based system only a single line is required to install all the necessary packages:
   `sudo apt-get install avarice avr-libc avra avrdude avrp avrprog binutils-avr gcc-avr gdb-avr simulavr doxygen`
   On non-debian based systems, these are the packages that must be found and installed:
   - avarice - use GDB with Atmel's JTAG ICE for the AVR
   - avr-libc - Standard C library for Atmel AVR development
   - avra - Assembler for Atmel AVR microcontrollers
   - avrdude - software for programming Atmel AVR microcontrollers
   - avrp - Programmer for Atmel AVR microcontrollers
   - avrprog - Programmer for Atmel AVR microcontrollers
   - binutils-avr - Binary utilities supporting Atmel's AVR targets
   - gcc-avr - The GNU C compiler (cross compiler for avr)
   - gdb-avr - The GNU Debugger for avr
   - simulavr - Atmel AVR simulator
   - doxygen - Documentation system for C, C++, Java, Python and other languages

Installing Plug-ins

6. Start Eclipse and go to Help -> Install New Software...
7. Click the Add button in the 'Install New Software' window, type 'AVR plugin' in the 'Name' field and '`http://avr-eclipse.sourceforge.net/updatesite/`' in the 'Location' field. Click OK. (See figure C.1)
8. One or more items should have appeared with checkboxes in the 'Install New Software' window. Check all the boxes or hit the 'Select All' button and hit 'Next'. (See figure C.1)
9. A review of packages to be added will be displayed, click 'Next' again.
10. Accept the licensing terms and click 'Finish'. This will start the installation. There may be a message regarding unsigned content. Click 'OK' and allow installation to complete. When Eclipse asks to restart, select 'Not Now'
11. Repeat steps 6 through 10 using the address `http://download.gna.org/eclox/update` for the Doxygen plugin. Restart Eclipse this time.

Eclipse is now ready to be used in the development of programs for the ATmega644p (or any other member of the AVR family of microcontrollers).

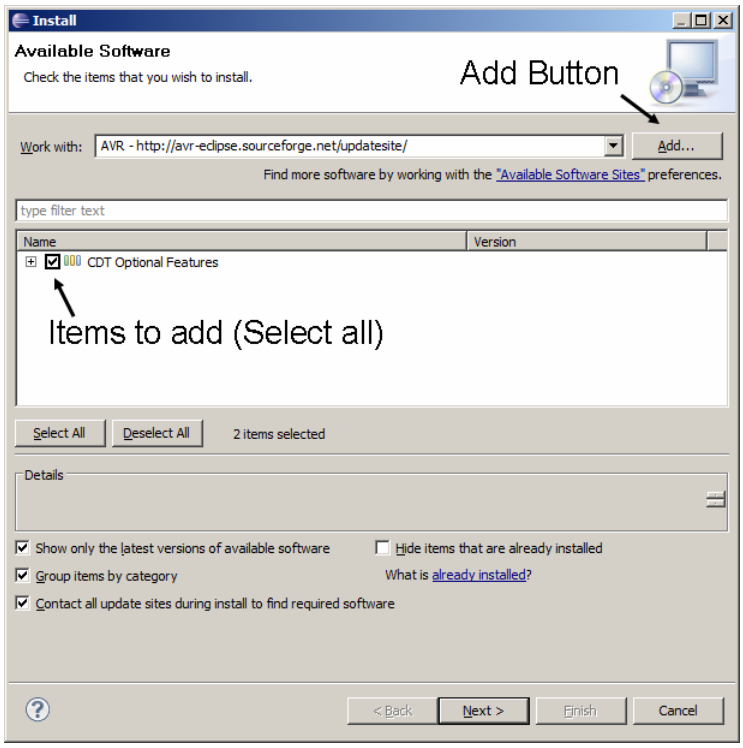


Figure C.1: Install New Software window from Eclipse

C.2 Making a project

1. Open the New Project wizard for C projects by: (See figure C.2)
   - File -> New -> C Project
   - Clicking the 'New' button on the toolbar, expanding 'C/C++', selecting 'C Project' and hitting 'Next'
   - Clicking the arrow next to the 'New' button on the toolbar and selecting 'C Project'
   - Right-clicking in the 'Project Explorer', highlighting 'New' and selecting 'C Project'
2. Once the 'C Project' window is open, enter a project name, expand 'AVR Cross Target Application' under 'Project type:' and select 'Empty Project'. Select 'AVR-GCC Toolchain' under 'Toolchains:' and click 'Finish'. (See figure C.3)
3. Right-click on the newly created project and choose 'Properties'.
4. Expand 'AVR' and select 'Target Hardware'

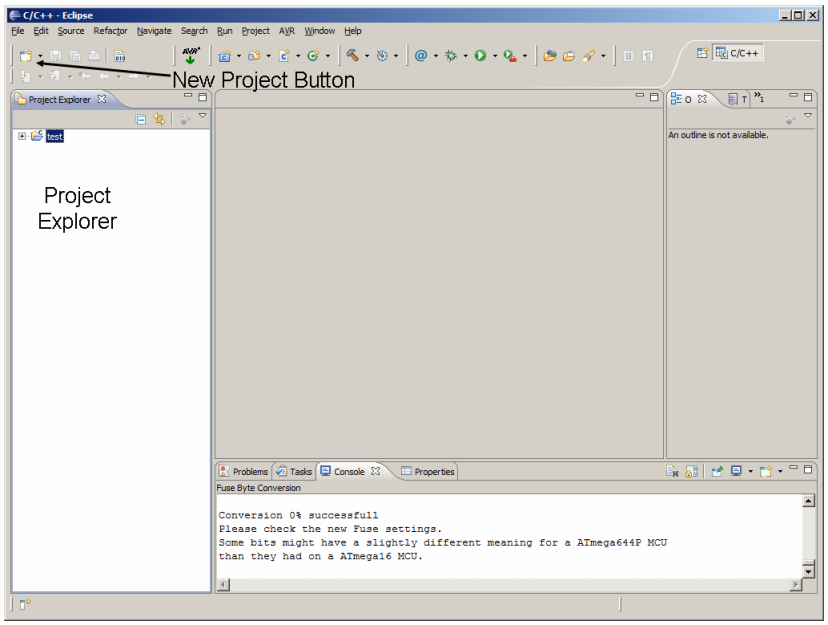


Figure C.2: Eclipse window with 'Project Explorer' and new project button marked.

5. In the 'MCU Type dropdown box select 'ATmega644p'. If the STK500 is connected to the computer and turned on, the 'Load from MCU' button should automatically detect the microcontroller.

6. Change the MCU Clock Frequency to 8000000 (8 million) and click 'Apply' in the lower right.

7. Select the 'AVRDude' properties.

8. Under the 'Programmer' tab select a configuration from the 'Programmer configuration' drop down menu. If there are no choices available, select 'New' to create a configuration. Name it whatever you like, and otherwise the defaults are acceptable for using an STK500 for any AVR microcontroller. Click 'OK' and select the new configuration.

9. Under the 'Fuses' tab select the 'direct hex values' radio button and click the icon to the right that looks like a pencil over top 0s and 1s.

10. Select options as seen in Fig ???. This is a good initial setup for the microcontroller. After finishing and clicking 'OK', the hex values in the boxes should read C2 D1 FF.

11. Click 'OK'

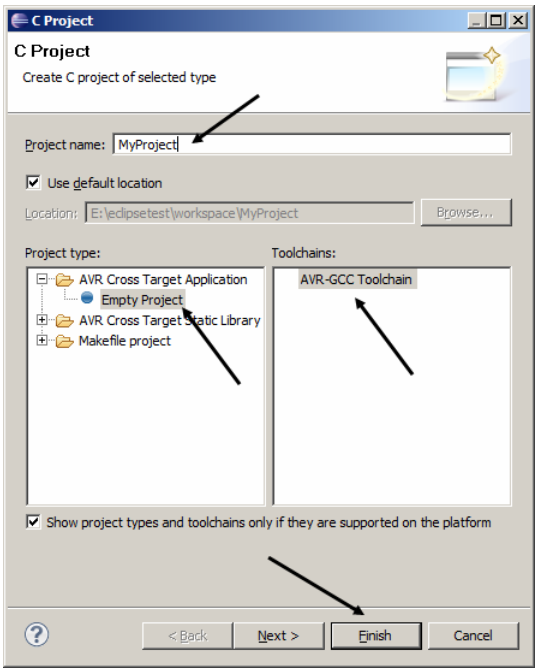


Figure C.3: C Project window.

The project is now setup to compile for the ATmega644p, and download properly.

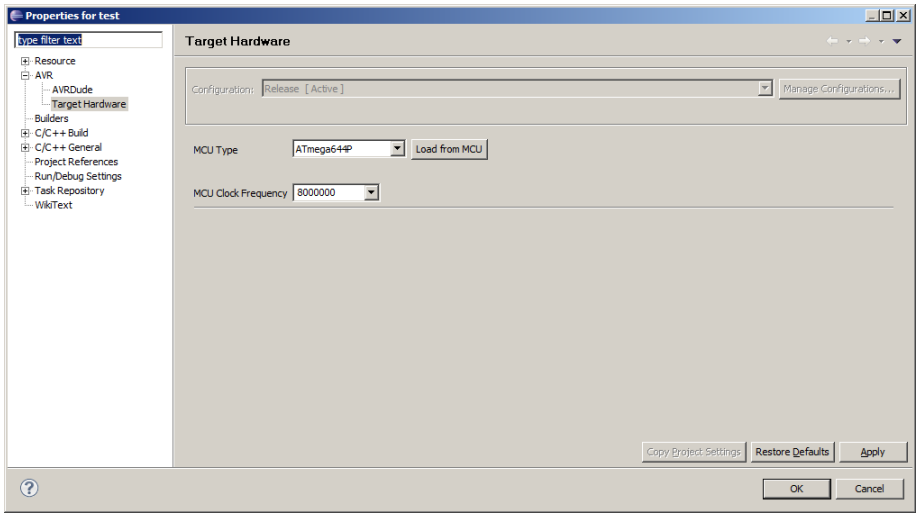


Figure C.4: Properties window for setting MCU type and clock frequency.

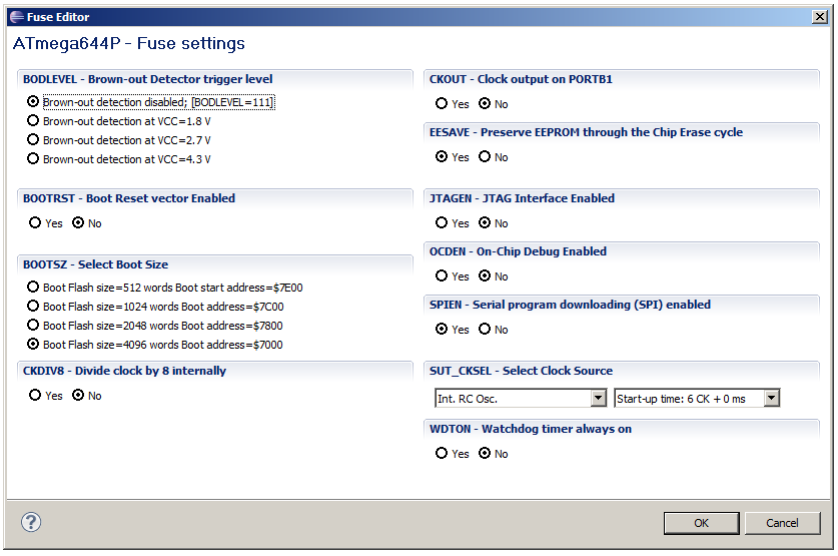


Figure C.5: Worksheet for setting internal fuses.

Bibliography

[1] Atmega644p. `http://www.atmel.com/dyn/products/product_card.asp?part_id=3896&category_id=163&family_id=607&subfamily_id=760`.

[2] Atmel avr studio 5. `http://www.atmel.com/microsite/avr_studio_5/default.asp?source=cms&icn=hmap1-AVR_STUDIO&ici=apr_2011`.

[3] Can programming be liberated from the von neumann style? a functional style and its algebra of program.

[4] Doxygen.com. `http://www.doxygen.com`.

[5] *EIA Standard RS-232-C Interface Between Data Terminal Equipment and Data Communication Equipment Employing Serial Data Interchange.*

[6] Microchip technology inc. `http://www.microchip.com`.

[7] Operational amplifier. `http://en.wikipedia.org/wiki/Operational_amplifier`.

[8] Operational amplifier applications. `http://en.wikipedia.org/wiki/Operational_amplifier_applications`.

[9] Operational amplifiers (opamps). (`http://www.rfcafe.com/references/electrical/opamps.htm`.

[10] Stk500. `http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2735`.

[11] Atmel. *8-bit AVR Microcontroller with 16/32/64K Bytes In-System Programmable Flash.*

[12] Jack Copeland. A brief history of computing.

[13] Globe Motors. *IM-13 GEARMOTORS.*

[14] Dan Gookin. C for dummies, May 2004.

[15] Walt Kester, editor. *The Data Conversion Handbook.* Elsevier: Newnes, 2005.

[16] Lite-On. *LTL2F3VxKNT/ 2H3VxKNT/ 2P3VxKNT/ 2R3VxKNT SERIES.*

[17] LSI/CSI. *LS7366R: 32-BIT QUADRATURE COUNTER WITH SERIAL INTERFACE.*

[18] SGS-Thomson. *M74HCT08: Quad 2-input AND gate.*

[19] SHARP. *GP2D12/GP2D15 General Purpose Type Distance Measuring Sensors.*

[20] Texas Instruments. *LM193, LM293, LM293A, LM393, LM393A, LM2903, LM2903Q DUAL DIFFERENTIAL COMPARATORS.*

[21] John von Neumann. First draft of a report on the edvac.