MORGAN & CLAYPOOL PUBLISHERS

Introduction to Embedded Systems

Using ANSI C and the Arduino Development Environment

David Russell

SYNTHESIS LECTURES ON DIGITAL CIRCUITS AND SYSTEMS

Mitchell Thornton, *Series Editor*

Introduction to Embedded Systems

Using ANSI C and the Arduino Development Environment

Introduction to Embedded Systems: Using ANSI C and the Arduino Development Environment

David J. Russell

www.morganclaypool.com

ISBN: 9781608454983 paperback
ISBN: 9781608454990 ebook

DOI 10.2200/S00291ED1V01Y201007DCS030

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

Lecture #30
Series Editor: Mitchell A. Thornton, *Southern Methodist University*
Series ISSN
Synthesis Lectures on Digital Circuits and Systems
Print 1932-3166 Electronic 1932-3174

Synthesis Lectures on Digital Circuits and Systems

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

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

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

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

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

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

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

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

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

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

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

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

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

Mismatch and Noise in Modern IC Processes
Andrew Marshall
2009

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

An Introduction to Logic Circuit Testing
Parag K. Lala
2008

Pragmatic Power
William J. Eccles
2008

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

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

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

Pragmatic Logic
William J. Eccles
2007

PSpice for Filters and Transmission Lines
Paul Tobin
2007

PSpice for Digital Signal Processing
Paul Tobin
2007

PSpice for Analog Communications Engineering
Paul Tobin
2007

PSpice for Digital Communications Engineering
Paul Tobin
2007

PSpice for Circuit Theory and Electronic Devices
Paul Tobin
2007

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

Pragmatic Circuits: Frequency Domain
William J. Eccles
2006

Pragmatic Circuits: Signals and Filters
William J. Eccles
2006

High-Speed Digital System Design
Justin Davis
2006

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

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

Introduction to Embedded Systems

Using ANSI C and the Arduino Development Environment

David J. Russell
University of Nebraska-Lincoln

SYNTHESIS LECTURES ON DIGITAL CIRCUITS AND SYSTEMS #30

MORGAN & CLAYPOOL PUBLISHERS

ABSTRACT

Many electrical and computer engineering projects involve some kind of embedded system in which a microcontroller sits at the center as the primary source of control. The recently-developed Arduino development platform includes an inexpensive hardware development board hosting an eight-bit ATMEL ATmega-family processor and a Java-based software-development environment. These features allow an embedded systems beginner the ability to focus their attention on learning how to write embedded software instead of wasting time overcoming the engineering CAD tools learning curve.

The goal of this text is to introduce fundamental methods for creating embedded software in general, with a focus on ANSI C. The Arduino development platform provides a great means for accomplishing this task. As such, this work presents embedded software development using 100% ANSI C for the Arduino's ATmega328P processor. We deviate from using the Arduino-specific Wiring libraries in an attempt to provide the most general embedded methods. In this way, the reader will acquire essential knowledge necessary for work on future projects involving other processors. Particular attention is paid to the notorious issue of using C pointers in order to gain direct access to microprocessor registers, which ultimately allow control over all peripheral interfacing.

KEYWORDS

embedded systems, embedded software, embedded development, microcontroller, microprocessor, ANSI C, Arduino, ATmega328P

To my best friend Jamie
and our three wonderful kids:
Gates, Gracen, and Gavin

Contents

Preface

It is well established that most, if not all, electrical and computer engineering projects involve some kind of "embedded system" in which a microcontroller or microprocessor sits at the center as the primary source of control. As a result, most related undergraduate engineering programs offer at least a semester course in which students are to learn the fundamentals of how an embedded system functions in general and then put their knowledge to practice by creating the embedded programs necessary to control the processor at the heart of the system. Unfortunately, most embedded environments are notorious for a combination of being 1) difficult to install, 2) difficult to use, 3) difficult to learn/understand, 4) difficult to maintain, 5) expensive, or 6) host-platform dependent. Thus, course instructors face a number of headaches, worst of which is wasting invaluable time during the semester trying to teach students how to use CAD tools.

All these problems are addressed with the recently-developed Arduino development platform, which includes an inexpensive hardware development board hosting an eight-bit ATMEL ATmega-family processor. For about $30, a student can purchase their own hardware platform and download the reference manual and Java-based software-development tools, which install with no hassle on any host platform (e.g., Mac OSX, Windows, Linux). After the five-minute installation, the user can immediately connect to the embedded development board via USB, compile and download an example program and earn instant gratification by blinking an LED. The true benefit of this development paradigm is that an embedded systems beginner can focus their attention on learning how to write software to interface with peripheral devices instead of spending time fighting the engineering CAD tools learning curve. Additionally, institutions do not need to spend time and money on a dedicated embedded lab, and students can practice on their own hardware.

In the spring 2010 semester, I decided to try the Arduino development environment in the Introduction to Embedded Systems sophomore class offered at UNL. Unfortunately, at the start of the semester, there were no suitable textbooks for presenting general knowledge covering the course objectives via the Arduino platform. As a result, I created the notes which eventually transformed into this work. I have relied on my 14+ years of industry experience with embedded programming to provide students with fundamental yet general knowledge of creating embedded software using 100% ANSI C for the Arduino's ATmega328P processor. I deviate from using the Arduino-specific libraries whenever possible to present the most general methods. Hopefully, this will result in students learning the core concepts that they can reapply on future projects where they are likely to work on different processors.

The book begins with an introduction to embedded systems in general, followed by a brief, yet complete, description of ANSI C provided from the point-of-view of an embedded programmer with emphasis on pointers-to-hardware registers. The first true embedded operation of utilizing

general purpose input/output (GPIO) port pins is presented in fair length, as it represents the spring-board to the rest of the embedded applications including analog-output via Pulse-Width Modulation (PWM), analog-sensory input via Analog-to-Digital Converters (ADCs), serial port interfacing, and the advanced topic of Interrupt Service Routine (ISR) processing.

I would like to thank Dr. Mark Bauer, of the Department of Electrical Engineering, University of Nebraska-Lincoln, for first introducing me to the Arduino and acting as a consultant for the class as I made the assignments. If it were not for Mark's natural enthusiasm of learning how everything works, this book would not exist. Further, I would like to thank the students who took the class during the spring 2010 semester. I'm not sure I would have had the patience they exhibited while waiting for new labs to be posted every couple of weeks. Additionally, I would like to thank Dr. Khalid Sayood, of the Department of Electrical Engineering, University of Nebraska-Lincoln, for introducing me to Joel Claypool and proofreading the initial manuscript. Finally, I thank Joel Claypool and his team at Morgan & Claypool Publishing, including Dr. C. L. Tondo, for allowing me the opportunity to generate my first book.

David J. Russell
July 2010

CHAPTER 1

Introduction

1.1 BACKGROUND

An *embedded system* is an electronic system that contains at least one controlling device, i.e. "the brain", but in such a way that it is hidden from the end user. That is, the controller is **embedded** so far in the system that usually users don't realize its presence.

Example 1.1 Several examples of devices containing embedded systems include automobiles, household appliances such as microwave, toaster, refrigerator, washer, dryer, television, etc., security systems, wireless network router, traffic light, 3G cell phones, cameras, mp3 audio players, DVD players, and various Bluetooth devices such as mouse, earpiece, keyboard, etc. This is a very small list compared to the thousands of embedded systems in operation today.

Given the wide variety of embedded systems listed in Ex. 1.1, a natural question arises. What is the controlling device? Using the small list from the example, consider how different each embedded system is from any other on the list (e.g., how similar is a traffic light with a Bluetooth mouse?). Now, imagine how an engineer would go about designing even the simplest system.

In fact, they all have some common features that can be coarsely quantized by the block diagram shown in Fig. 1.1. Every system contains some input and output elements in order to interact with the environment, i.e., the workers. Additionally, there has to be some governing mechanism that manages the behavior of the system as a whole, i.e., the manager.

During the 1960's, 70's and 80's, the controller would be designed using discrete components to perform a specific function. In this scenario, it would be difficult to compare a toaster system to a traffic light system from the 60's or 70's, yet there was toast! So, it is clear that electrical engineers needed to be capable of designing many circuits in order to handle input devices, determine what to do and/or process some signals, then output feedback to the user via an output device. In this paradigm, every circuit in every system is likely to be designed from scratch. This is expensive since very little, if any, reuse of previously designed systems occurs. Without reuse, we have to recreate sub-systems that may have already been designed in previous scenarios. By not leveraging reuse, manpower must be dedicated to retesting and reworking every sub-circuit for error-free functionality.

As time marches on, research and industry have steered in a direction to lower costs of all systems by reusing as many components and sub-systems as possible. The most dramatic hardware component to allow this reuse is the brain in Fig. 1.1 of every system. All previously designed circuitry for the controller mechanism has been replaced with, at least, a single component, either a Programmable Logic Device (PLD) or a microprocessor/microcontroller.

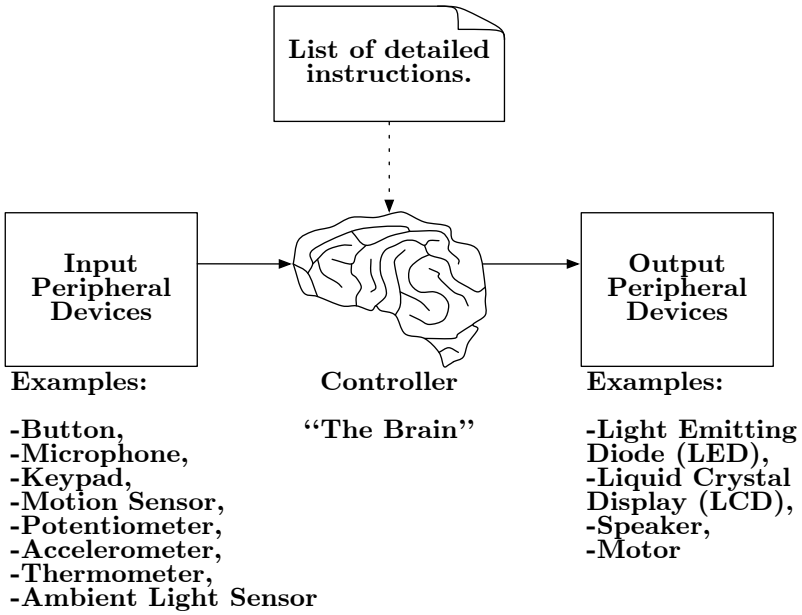


Figure 1.1: Conceptual embedded system block diagram.

A *peripheral device*, or just peripheral, is a device attached to a controlling mechanism, for example a processor or host computer, yet is not actually part of the controlling mechanism, and whose operation is functionally dependent upon the controlling mechanism. Peripherals are used to supplement a system's overall functionality, yet they require some extra mechanism to control their behavior.

For our brain in Fig. 1.1, we need to design and build all the functions to control the peripherals, and make system-level decisions. That is, based on asynchronous input actions, such as a button being pressed down, make corresponding decisions and notify the user via an output device, for example, rotate a motor 90°. Classically, these circuits would be built using standard electronic building blocks, including resistors, capacitors, diodes and transistors. However, the current, almost universal, solution to the customized brain is using a single chip being either a microprocessor/microcontroller or a PLD. Both devices are capable of performing many generic actions. In either case, the problem of designing the brain functionality **still exists**, but now we don't use discrete components to build our circuits. Instead, we have to "tell" the controller what actions to do. We do this by writing down very specific steps in a language that only the controller understands.

To be clear, we are still designing a solution to the controller brain, but instead of discrete circuit components, we have to write down instructions for a "dumb" yet very capable single component.

Example 1.2 Suppose we want to build a "vending machine system". First, let's identify the components that fit the conceptual block diagram in Fig. 1.1. Our input peripherals will be a money slot and a product button. The output peripherals will include a product drawer and a change return. For the sake of the analogy, we are going to use a person as the brain; perhaps, it might be better to think of this as a "concession stand system", so as not to violate human rights.

The peripherals are fairly self-explanatory. Regarding the person as the brain, there are some fundamental abilities that we are assured all people exhibit; for example, it is a given that all people know how to breathe. **Given:** the person can perform all basic tasks like addition/subtraction, observation, and they can use their hands to grab the product and set it in the product drawer. **Need:** the person needs to be told how much each product costs, which button corresponds to which item, and that money needs to be put in before any change or items may be dispensed. So, the owner of the machine needs to write down the instructions for the worker.

The system presented in Ex. 1.2 seems silly, but consider a couple of real examples that follow the same concept.

Example 1.3 Telephone operators from the 1960's or 1970's were people who would sit and manually connect calls together via a patch pannel.

Example 1.4 Consider the general operation in a high-tier kitchen. The head chef would receive an order ticket for an entire table of customers. The chef shouts out the order to all the line cooks each manning a different station. Each cook responds to the entire order by picking the items for which they are responsible. Each line cook delivers their part of the order back to the chef who assembles the meal and passes it to a waiter for delivery to the table.

In both real examples of Ex. 1.3 and Ex. 1.4, the operator and line cooks have general skills available that allows them to perform many different tasks every day such as brush teeth, drive a car, use a phone, do their taxes, etc. But, for the specific examples, the people need very clear and precise instructions on how to complete the specific task at hand. The operator would have been trained on his/her equipment. The line cooks would have been previously trained, with specific instruction from the chef prior to service; the chef can then issue general commands to the cooks and be assured they will be able to complete their individual tasks.

This is true for our controller brain. A microprocessor/microcontroller is controlled via a software program that lists "native" instructions on what to do. A hardware description defines how internal hardware blocks on a PLD are to be connected together. We can transition from Fig. 1.1 to the more specific block diagram shown in Fig. 1.2.

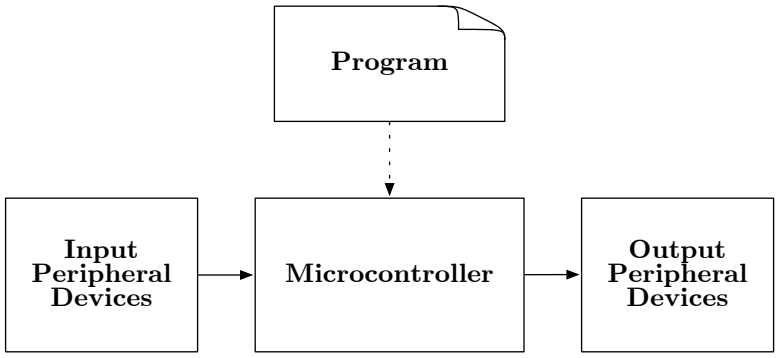


Figure 1.2: Embedded system block diagram using a microcontroller.

The *program* in Fig. 1.2 is our "very specific list of instructions" in the analogy of Ex. 1.2. Then, the process of "creating the program" is where much of system engineering time is spent. Classically, the engineering task would include creating custom circuitry for system control and peripheral interfacing. By contrast, today the engineering task involves creating a list of instructions for system control and peripheral interfacing.

The program exists in the system in some form of memory device. Then, the processor has the ability to access the program, execute individual instructions from the list, pick the next instruction to execute, and continue through the entire list until it is finished.

The goal of the rest of this book is for the reader to learn how to create various programs in order to make systems do interesting things. This includes **making system-level decisions** and **controlling peripheral devices**.

The program is often referred to as *software*, while the physical system components are called *hardware*. Software design is all about creating patterns of 0's and 1's in order to get the hardware to do what we want. Sometimes, it is difficult to understand the importance of embedded programming because software is virtual, whereas hardware is physical. However, the truth is that both are equally important for a successful system.

Returning to our analogy in Ex. 1.2, where we have certain expectations from a random person off the street, what should you expect from a generic processor? It turns out that some things are guaranteed to be universal. A processor is just a synchronous digital circuit. In particular, it is an Application Specific Integrated Circuit (ASIC), a very large digital circuit entirely fabricated on a single chip. As a result, we know that our processor needs DC power (V_{DD}), ground (GND) and a clock signal. A *clock* is a periodic square-wave signal, usually with a 50% duty cycle, i.e., logic '1' 50% of the time. The signal transitions of the clock, i.e., the rising and falling edges, are used as signal

events so that all system components will perform actions at the same time. In other words, they are synchronized to the clock.

We also know that a processor contains a custom instruction decoder sub-circuit that takes as input some single "instruction" and outputs a signal or set of signals that go to other circuit sub-systems within the microctonroller, telling them to perform some function. This is the heart of the processor, and it is the bulk of the Central Processing Unit (*CPU*). Because this is just a custom decoder circuit, instructions valid for processor A are likely completely different for processor B. That is, the *machine language* is unique for every processor.

Along with the CPU, there is an Arithmetic/Logic Unit (*ALU*) that is responsible for performing all of the standard operations such as addition, subtraction, multiplication, etc. Each operation takes in its operands from values stored in CPU *registers*, which are small groups of memory used by the ALU and CPU.

Finally, every processor has access to two conceptual banks of memory. Non-volatile memory holds the program, kind of like a textbook, while volatile memory provides the CPU the freedom to run the program, kind of like scratch paper.

Just as there are common components and features that you can expect out of every microcontroller, there are also common questions that arise at the beginning of every project based on a new component. We need to know the power supply requirement, maximum clock frequency which determines how fast a single instruction can be executed, the machine language, the capabilities of the ALU, the number and size of the CPU registers, how instructions interact with the registers, how much memory is available, etc.

All relevant processor information is found in a document provided by the manufacturer called a data sheet or reference manual. It includes physical characteristics including the package dimensions and operating properties such as temperature effects. It also includes specific details regarding various components/features/blocks of the component such on-board peripherals, internal memory, maximum clock frequency, number and size of registers, etc. It also includes the instruction set on how to control and/or interact with the component.

We will refer to the ATMEL ATmega328P data sheet throughout the book to learn how various aspects of our specific microcontroller works. In particular, whenever necessary, the relevant features, register addresses and meanings will be provided within the respective section.

1.2 DIGITAL REPRESENTATION OF INFORMATION

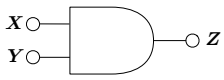
Most computers are *digital*, being constructed out of digital circuits. Voltage levels are used to represent the binary symbols '0' and '1'. Often, $GND =$'0' and $V_{DD} =$'1'. Because voltage is an analog measure, some threshold is set such that if $x < T$, then we have a '0' and if $x > T$, then we have a '1'. The choice of T is such that the largest amount of noise is tolerated; this is called the *noise margin*.

The CPU is simply a large digital circuit. It operates on digital inputs and produces digital outputs. Each individual input/output signal is either a '0' or a '1'. Notice these are the two elements

that make up the set of integers in base-2, also called *binary*. And so, '0' and '1' are BInary digiTS (*BITS* – a term coined by C. E. Shannon as a unit of measure for information content).

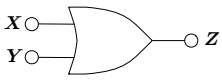
1.3 DIGITAL LOGIC FUNDAMENTALS

Digital logic is an entire subject dedicated to learning all that is required in order to design and construct a microprocessor. Clearly, this is beyond the scope of this book. The reader might consider Katz (2004) and Brown and Vranesic (2008). Interestingly, once we have the microcontroller placed in our system, we can actually perform the low-level generic digital logic functions given input and output signals to and from the processor. It is a well understood result that given two signals and the following fundamental boolean algebra functions, any digital logic system can be implemented.



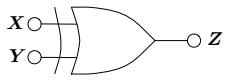
$\cap$, $\bullet$, AND

X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1



$\cup$, +, OR

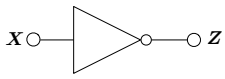
X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1



$\oplus$, XOR

X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	0

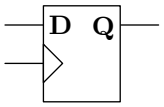
Note that ANDing a signal with 0 will always clear to 0, $x \cdot 0 = 0$, ORing a signal with 1 will always set to 1, $x + 1 = 1$, and XOR a signal with 1 will invert it, $x \oplus 1 = \bar{x}$, while XOR with 0 will pass it through, $x \oplus 0 = x$.



–, NOT

X	Z
0	1
1	0

The final component that needs to be introduced is the D flip-flop, which is a single bit of memory. For example, all the ATmega328P microcontroller registers are simply eight D flip-flops that are accessed simultaneously.



D	Q	Q^+
0	0	0
0	1	0
1	0	1
1	1	1

1.4 DIGITAL VECTORS

Because we need many bits in order to do anything meaningful, we often group them together into n-bit vectors. The most common atomic grouping is called a *byte*, which is a vector of eight bits. Most of the time the smallest unit the processor will operate on is a byte. Additionally, many times processors will operate on multiple bytes at the same time.

Example 1.5 Suppose the CPU is required to logically AND two inputs together. It would have to perform the logical AND operation bit-by-bit on two input bytes, probably sitting in CPU registers. Such an operation is shown in Fig. 1.3

Generally, bytes will be the smallest unit we deal with, so we will often need to look at 8-bit patterns of 0's and 1's.

Example 1.6 Consider the byte containing the bit patterns (0, 1, 0, 0, 1, 1, 0, 1). It is easy to see that writing down bit patterns is not very efficient. We might try converting it to decimal, i.e., base-10, by noting that $(0, 1, 0, 0, 1, 1, 0, 1) \times (128, 64, 32, 16, 8, 4, 2, 1) = 77$.

The *decimal*, i.e., base-10, representation from Ex. 1.6 is certainly more compact, but it was computationally expensive for us to convert between the two. Instead, a much more convenient system is *hexadecimal*, i.e., base-16, or just *hex*. Every byte may be represented by two hexadecimal digits, where in base-16 the digits are $\{0, \ldots, 9, A, \ldots, F\}$. We have a nice one-to-one correspondence between each hexadecimal digit and a 4-bit base-2 vector, sometimes called a *nibble*. A complete listing of conversions is given in Table 1.1.

Example 1.7 Now, given the binary string $\{01001101\}_2$, we have the direct lookup conversion of $\{4D\}_{16}$. Similarly, if we start with the hex value of $\{53\}_{16}$, we have the direct bit string of $\{01010011\}_2$.

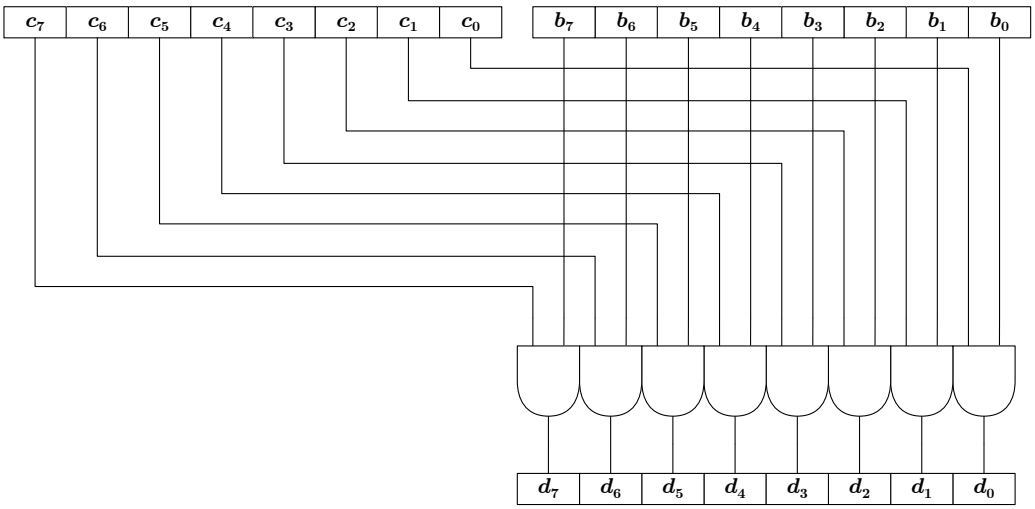


Figure 1.3: Logical AND of input signals via the ANDing of two 8-bit vectors. This is required since the CPU generally can't access individual bits – the smallest unit in most CPUs is an 8-bit byte.

Table 1.1: Binary to Hexadecimal Conversions

Base			Base			Base			Base		
2	10	16	2	10	16	2	10	16	2	10	16
0000	0	0	0100	4	4	1000	8	8	1100	12	C
0001	1	1	0101	5	5	1001	9	9	1101	13	D
0010	2	2	0110	6	6	1010	10	A	1110	14	E
0011	3	3	0111	7	7	1011	11	B	1111	15	F

Note that we must be careful about which base we are in.

Example 1.8 What is 101? This can mean two different things depending of in we are in base-2 or base-16, or base-10 for that matter. If we explicitly qualify $\{101\}_2 = \{5\}_{10} = \{5\}_{16}$, but $\{101\}_{10} = \{1100101\}_2 = \{65\}_{16}$.

The numbers presented in Ex. 1.7 and Ex. 1.8 are clearly presented by adding the extra base identifier. In programming, these identifiers are replaced by the prefix notations `0x` for hex values, and numbers without any marking are identified as decimal. Typically, binary strings are not stated in favor of the more compressed hex representation.

1.5 INFORMATION REPRESENTATION IN A DIGITAL PROCESSOR

1.5.1 NUMBERS

1.5.1.1 Unsigned Integers

All bits of an n-bit binary string are used to represent the inclusion or exclusion of a power of 2. The base-10 positive integer is represented via the string $(b_{n-1}, ..., b_1, b_0)$, which is interpreted as

$$\{N\}_{10} = \sum_{i=0}^{n-1} \left(b_i \cdot 2^i \right). \tag{1.1}$$

Example 1.9 A typical byte has $n = 8$. Suppose $\{N\}_2 = (10011100)$, then

$$\begin{aligned} \{N\}_{10} &= 0 \cdot 2^0 \\ &+ 0 \cdot 2^1 \\ &+ 1 \cdot 2^2 \\ &+ 1 \cdot 2^3 \\ &+ 1 \cdot 2^4 \\ &+ 0 \cdot 2^5 \\ &+ 0 \cdot 2^6 \\ &+ 1 \cdot 2^7 \\ &= (4 + 8 + 16 + 128) = 156 \end{aligned}$$

For n-bit bytes, we are able to represent any integer in the range $\{0, ..., 2^n - 1\}$.

Example 1.10 A typical byte has $n = 8$, which implies the range of values representable is $\{0, ..., 255\}$. Other likely values of n are 16 and 32, which have ranges of $\{0, ..., 65,535\}$ and $\{0, ..., 4,294,967,295\}$, respectively.

Note that we have implied a bit ordering in our definition of the bit string. We have placed the most significant bit (*MSB*) as the first element of the bit vector and the least significant bit (*LSB*) as the last element of the vector. This is called *big-endian* bit format. However, some manufacturers will reverse this order such that $(b_0, ..., b_{n-1})$, which is called *little-endian*. Either format is fine, as long as the reader is aware of the notation. We will use big-endian in this text, which follows the ATMEL ATmega328P reference manual as well.

Example 1.11 Suppose you are given $\{N\}_2 = (00100110)$ but not told the order. The meaning of the bit string could be either $N_{\text{big-endian}} = 32$ or $N_{\text{little-endian}} = 100$. Clearly, it is important to know the bit-endianness.

1.5.1.2 Signed Integers

Sometimes, there is a need for the ability to represent negative integer values. To do so, we need to use one of the bits in the string as a flag to indicate when a number is meant as a positive or a negative integer. This bit is called the *sign bit,* and it is usually the MSB. Let $b_{n-1} = s$, and then if $s = 0$ then $N > 0$ or if $s = 1$ then $N < 0$.

Example 1.12 Let $n = 8$. The representation for 77 and -77 are

		sign	magnitude
77	=	0	1001101
-77	=	1	1001101

Here the sign of the number is determined by the MSB sign bit, and the magnitude is calculated by Eq. (1.1).

In this code, out of n-bits, we have $n - 1$ bits for the magnitude and 1 bit for the sign indication. For n-bit bytes, we are able to represent any integer in the range $\{-2^{n-1} + 1, ..., -0, 0, ..., 2^{n-1} - 1\}$.

Example 1.13 A typical byte has $n = 8$, which implies the range of values representable is $\{-127, -0, 0, ..., 127\}$.

This signed-integer code is called *Sign and Magnitude*. Its two problems include two versions of zero, i.e., (10...0) and (00...0) are both mathematically equal to zero in base-10, and the hardware required to perform arithmetic in the ALU is complex, large, slow, etc. Addition of two positive or negative numbers in sign and magnitude is performed by adding the magnitude bits and using the sign bit of both numbers, as they are the same. Addition of a positive and negative number is a little more complicated.

1. Compare the magnitudes. The number with the largest magnitude is placed on top.
2. Perform binary subtraction, i.e., borrowing from the next bit position, etc.
3. The resulting sign bit is the same as the top number.

So, the hardware necessary in an ALU would include a magnitude comparator, an unsigned adder and an unsigned subtractor. These issues have led to more complicated codes for humans to understand directly.

First, consider the code called *Ones' Complement* in which the magnitude is interpreted as a function of the sign bit. In other words, the base-10 value is found as follows. As in the previous code, let $b_{n-1} = s$ then

$$\{N\}_{10} = \begin{cases} \sum_{i=0}^{n-1} b_i \cdot 2^i & \text{if } s = 0 \\ -\sum_{i=0}^{n-1} \bar{b_i} \cdot 2^i & \text{if } s = 1 \end{cases} \tag{1.2}$$

So, the magnitude of the number is calculated after the sign-bit is checked. If the number is positive, the magnitude is as in Eq. (1.1). If the number is negative, the bits are all inverted before the magnitude is calculated with the same equation as before.

Example 1.14 Let $n = 8$. The representation for 77 is (01001101), just like before. To get -77 in the ones' complement code, we negate the number, which means we simply invert all the bits, so (10110010).

Example 1.15 Let $n = 8$. What do the bit strings (00110111) and (11100110) mean if we interpret them using ones' complement? First, look at the sign bit. The first string has a '0', so we know it is a positive number. To calculate its magnitude, we just interpret the bits like we did by Eq. (1.1). So, (00110111) = 55. The second string has a '1', so we know it is a negative number. To calculate its magnitude by Eq. (1.2), we need to invert the bits before we add the powers-of-2. So,

$$(11100110) \Rightarrow -(00011001) \Rightarrow -25$$

which means (11100110) = −25 in ones' complement.

In this code, out of n-bits, we have $n - 1$ bits for the magnitude and 1 bit for the sign indication. For n-bit bytes, we are able to represent any integer in the range $\{-2^{n-1} + 1, ..., -0, 0, ..., 2^{n-1} - 1\}$.

Example 1.16 A typical byte has $n = 8$, which implies the range of values representable is $\{-127, -0, 0, ..., 127\}$.

It turns out the ones' complement code has solved the issue of hardware complexity in the ALU. In particular, performing addition and subtraction on ones' complement bit strings can be done via unsigned adder hardware. What this means is that the ALU of a microcontroller can implement a single piece of adder/subtractor hardware and still be able to solve addition and subtraction on both unsigned and signed integers. Addition of any two numbers in ones' complement is as follows:

1. Perform binary addition of all bits, including the sign bit.
2. If a '1' is carried out, add it back to the result.

Subtraction of any two numbers in ones' complement is handled by negating one of the numbers and performing addition.

You may have noticed that ones' complement still suffers from having two versions of zero. To correct this, consider the code called *Two's Complement* in which, just like the ones' complement code, the magnitude is interpreted as a function of the sign bit. In other words, the base-10 value is found as in the previous codes, let $b_{n-1} = s$ and

$$\{N\}_{10} = \begin{cases} \sum_{i=0}^{n-1} b_i \cdot 2^i & \text{if } s = 0 \\ -\left[\left(\sum_{i=0}^{n-1} \bar{b}_i \cdot 2^i\right) + 1\right] & \text{if } s = 1 \end{cases} \tag{1.3}$$

So, the magnitude of the number is calculated after the sign-bit is checked. If the number is positive, the magnitude is as in Eq. (1.1). If the number is negative, the bits are all inverted before the magnitude is calculated with Eq. (1.1). The result has 1 added to get the final magnitude.

Example 1.17 Let $n = 8$. The representation for 77 is (01001101), just like in all other cases. To get -77 in the two's complement code, we negate the number, which means we invert all the bits and add 1, so (10110011).

Example 1.18 Let $n = 8$. What do the bit strings (00110111) and (11100110) mean if we interpret them using two's complement? First, look at the sign bit. The first string has a '0', so we know it is a positive number. To calculate its magnitude, we just interpret the bits like we did by Eq. (1.1). So, (00110111) = 55. The second string has a '1', so we know it is a negative number. To calculate its magnitude by Eq. (1.3), we need to invert the bits before we add the powers-of-2. Also, we need to add 1 before applying the minus sign. So,

$$(11100110) \Rightarrow -[(00011001) + 1] \Rightarrow -26$$

which means (11100110) = −26 in two's complement.

In this code, out of n-bits we have $n - 1$ bits for the magnitude and 1 bit for the sign indication. For n-bit bytes, we are able to represent any integer in the range $\{-2^{n-1}, ..., -1, 0, ..., 2^{n-1} - 1\}$.

Example 1.19 A typical byte has $n = 8$ which implies the range of values representable is $\{-128, ..., 127\}$.

Addition of any two numbers in two's complement is as follows:

1. Perform binary addition of all bits including the sign bit.
2. If a '1' is carried out, discard it.

Subtraction of any two numbers in two's complement is handled by negating one of the numbers and performing addition.

Note that **two's complement is the only signed integer code that is ever used in practice**. The other two codes are always presented in order to show the progression from taking an unsigned bit string and turning it into an implementable signed-integer representation. The two's complement code is always assumed in computer architecture for the same reason ones' complement was a good improvement over sign and magnitude; that is, the two's complement code allows for addition and subtraction treatment of bit strings as though they were unsigned integers. The code was designed to allow for efficient hardware in an ALU, i.e., the code does not care if human engineering students are unable to decipher the meaning of a bit vectors for a homework assignment.

Note that we must be careful of *overflow* conditions which occur when not enough bits are available to hold the correct meaning of a calculation.

Example 1.20 Consider using an unsigned adder sub-circuit to perform addition on some bit vectors. We will only care about the unsigned and two's complement contexts. In the following table, each unsigned hardware operation adds the binary strings, and any carry that is output from the MSB stage is discarded. Depending on the context of signed or unsigned, an overflow indication may be made. Note that overflow is not the same thing as carry-out of the MSB.

$$\begin{array}{r} 0100 \\ 1101 \\ \hline (1)0001 \end{array}$$

In the two's complement context, this is 4 - 3 = 1, which is correct, and there is no arithmetic overflow in spite of the fact that a '1' was carried out of the adder's MSB position. However, in the context of unsigned integers, the fact that there is carry out of the MSB is an indication that overflow has occurred, as $4 + 13 \neq 1$. If we had a fifth bit available for the output of the operation, then we would not have had overflow since 4 + 13 = 17 which is 10001 in base-2.

Unsigned Add in Hardware	Unsigned			Two's Complement	
	Bit String	Value	Overflow	Value	Overflow
(0100 + 0011)	(0111)	4 + 3 = 7		4 + 3 = 7	
(0100 + 0101)	(1001)	4 + 5 = 9		$4 + 5 \neq -7$	✓
(1100 + 1101)	(1001)	$12 + 13 \neq 9$	✓	-4 + -3 = -7	
(1100 + 1011)	(0111)	$12 + 11 \neq 7$	✓	$-4 + -5 \neq 7$	✓

1.5.1.3 Floating Point

Unlike integers, in which hardware is pretty much all the same across all architectures, floating-point codes are not standard. However, the same basic concepts are generally present. In particular, out of an n-bit number, one bit is required for the sign of the number, m bits are required for the mantissa, and the remaining e bits are used to represent the exponent. The IEEE floating point formats are shown in Fig. 1.4.

It turns out that floating point arithmetic is always very cycle expensive unless the processor has a floating point unit built into the ALU, which is hardware expensive. To determine the value of x given a bit string in IEEE single precision, let s be the sign bit, e the eight exponent bits and m the 23-bit mantissa. Then

$$x = (-1)^s \times \left(2^{e-127}\right) \times 1.m \tag{1.4}$$

Similarly, the value of x given an IEEE double precision bit string is

$$x = (-1)^s \times \left(2^{e-1023}\right) \times 1.m \tag{1.5}$$

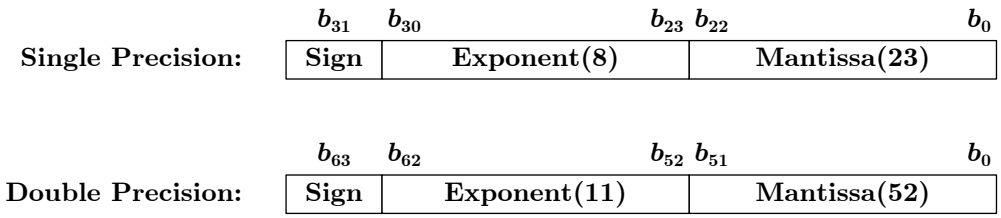


Figure 1.4: The bit fields of single and double precision IEEE floating point representations.

The s in Eq. (1.4) and Eq. (1.5) controls the sign, just like in all the signed integer codes. The exponent e moves the binary point, the base-2 equivalent to the base-10 decimal point, to either the right or left and the mantissa m contains the numerical information, or precision.

Example 1.21 To understand the binary point, consider the 4-bit vector (1111). In all of the integer codes, we have a binary point that is always implied to the right of the LSB. In particular, 1111.0 which is determined to be $2^3 + 2^2 + 2^1 + 2^0 = 15$. Now suppose the binary point is shifted to the left, we have

$$
\begin{aligned}
1111. &= 2^3 + 2^2 + 2^1 + 2^0 = 15 \\
111.1 &= 2^2 + 2^1 + 2^0 + 2^{-1} = 7.5 \\
11.11 &= 2^1 + 2^0 + 2^{-1} + 2^{-2} = 3.75 \\
1.111 &= 2^0 + 2^{-1} + 2^{-2} + 2^{-3} = 1.875 \\
.1111 &= 2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} = 0.9375
\end{aligned}
$$

Notice that because all numbers other than 0 always contain at least one 1, floating point numbers are usually interpreted such that the leading 1 is assumed, with the mantissa containing the entire fractional part.

Example 1.22 Consider the following IEEE single precision representations. Notice that the leading 1 is always assumed in Eq. (1.4). As a result, there is a special check if all bits are 0 to represent 0.0.

Base-10 Number	Sign	Exponent	Mantissa	Value
1.0	0	01111111	00000000000000000000000	1.0
0.5	0	01111110	00000000000000000000000	0.5
3.5	0	10000000	11000000000000000000000	3.5
largest	0	11111111	11111111111111111111111	6.80564e+38
smallest	0	00000000	00000000000000000000001	5.87747e-39
0.0	0	00000000	00000000000000000000000	0.0
2.44	0	10000000	00111000010100011110101	2.43999
-12.16	1	10000010	10000101000111101011100	-12.15999
-9.72	1	10000010	00110111000010100011110	-9.71999

We perform addition just like in base-10.

1. Align the binary points before performing the addition or subtraction. To avoid losing significant digits, the larger number is left alone, and the smaller number has zeros padded in order to move the binary point into position. In this way, if any precision is lost, it is dropped from the smaller number to reduce overall error in the calculation.
2. Addition or subtraction is performed on the aligned magnitudes. This includes the sign of each number.
3. The result will have its binary point adjusted to account for any increase or decrease in order of magnitude.

$$z = \begin{cases} \left[(-1^{s_x} \times 1.m_x) + \left(-1^{s_y} \times 1.m_y \times 2^{-(e_x - e_y)}\right)\right] \times 2^{(e_x - 127)} & \text{if } |x| \geq |y| \\ \left[\left(-1^{s_y} \times 1.m_y\right) + \left(-1^{s_x} \times 1.m_x \times 2^{-(e_y - e_x)}\right)\right] \times 2^{(e_y - 127)} & \text{if } |x| \leq |y| \end{cases} \tag{1.6}$$

Example 1.23 Let $x = 2.44$ and $y = -12.16$. These numbers can be broken down into their components, so that

$$\begin{aligned} x &= 2.44 \\ &= -1^0 \times 2^{(128-127)} \times 1.22 \\ &= (0, 10000000, 00111000010100011110101) \end{aligned}$$

and

$$\begin{aligned} y &= -12.16 \\ &= -1^1 \times 2^{(130-127)} \times 1.52 \\ &= (1, 10000010, 10000101000111101011100) \end{aligned}$$

Note that the leading 1 is always assumed, so the mantissa represent 0.22 and 0.52. Now, since $|y| > |x|$, we calculate the sum as

$$\begin{aligned} z &= \left[\left(-1^1 \times 1.52\right) + \left(-1^0 \times 1.22 \times 2^{-(130-128)}\right)\right] \times 2^{(130-127)} \\ &= \left[(-1 \times 1.52) + \left(1.22 \times 2^{-2}\right)\right] \times 2^3 \\ &= [(-1 \times 1.52) + (0.305)] \times 2^3 \\ &= [-1 \times 1.215] \times 2^3 \\ &= -9.72 \\ &= -1^1 \times 2^{(130-127)} \times 1.215 \\ &= (1, 10000010, 00110111000010100011110) \end{aligned}$$

Clearly there are more steps involved in performing floating point addition as compared to integer addition.

Single precision floating point multiplication is performed via the equation

$$z = \left(-1^{s_x} \times 1.m_x\right) \times \left(-1^{s_y} \times 1.m_y\right) \times 2^{(e_x+e_y-254)} \tag{1.7}$$

Example 1.24 Let $x = 2.44$ and $y = -12.16$, as in Ex. 1.23. We calculate the product as

$$\begin{aligned} z &= \left(-1^1 \times 1.52\right) \times \left(-1^0 \times 1.22\right) \times 2^{(130+128-254)} \\ &= -1 \times 1.8544 \times 2^4 \\ &= -29.6704 \\ &= -1^1 \times 2^{(131-127)} \times 1.8544 \\ &= (1, 10000011, 11011010101110011111010) \end{aligned}$$

The result you should take away from this section is that floating point arithmetic can be implemented either in hardware or in software on a fixed point processor. For most microprocessors, the ALU is fixed-point only, so the program written needs to implement all the binary point adjustments similar to Ex. 1.23 and Ex. 1.24. This is considered cycle expensive since it takes a lot of instructions to tell the processor how to perform a single arithmetic operation. So, most embedded applications should try to stick with integer arithmetic whenever possible.

1.5.1.4 Binary Coded Decimal

Binary Coded Decimal (*BCD*) is a special code used to allow direct conversion between a nibble of bits and a positive decimal value. The binary to BCD conversion is presented in Table 1.2.

Example 1.25 The number $\{53\}_{10}$ is represented as the bit vector $\{01010011\}_{\text{BCD}}$ in BCD.

Table 1.2: Binary to BCD Conversions

Base-2	BCD	Base-2	BCD	Base-2	BCD	Base-2	BCD
0000	0	0100	4	1000	8	1100	-
0001	1	0101	5	1001	9	1101	-
0010	2	0110	6	1010	-	1110	-
0011	3	0111	7	1011	-	1111	-

BCD is a code that allows for a quick conversion between base-2 and base-10. However, it is not very compact because each nibble is only able to code for ten digits instead of 16.

Example 1.26 Recall from Ex. 1.10 that the unsigned integer range of a byte is {0, ..., 255}. The range of a byte in BCD is reduced to {0, ..., 99}. Similarly, where the normal unsigned integer ranges for $n = 16$ $n = 32$ are {0, ..., 65, 535} and {0, ..., 4, 294, 967, 295}, respectively, the BCD ranges are {0, ..., 9, 999} and {0, ..., 99, 999, 999}.

BCD is more of a historic code, and generally computer architectures are not designed to manage BCD values directly. BCD is still useful for embedded applications when interacting with seven-segment displays, as will be seen in subsequent chapter assignments.

1.5.2 TEXT

The American Standard Code for Information Interchange (*ASCII*) is a binary code in which 7-bit vectors are used to represent many alphabetic, numeric and symbolic characters. The standard was originally designed to include control characters meant to manage peripherals such as printers, and so some of the symbols do not see a lot of use any more. The decoder is presented in Table 1.3. The entire first column contains all of the control characters.

Example 1.27 Given the typical 8-bit byte, the bit vector (01010011) represents the printable character 'S' based on the ASCII code.

More recent versions of the code include extended ASCII, which uses the 8th bit in the standard byte to include 128 other characters. Unicode versions include ASCII as well as variable-length codewords depending on the context.

SUMMARY

In electrical and computer engineering, there are always problems that require a designed solution. Historically, each solution would be a specific hardware implementation completely unique to the application at hand. Modern day solutions involve as much intellectual reuse as possible, beginning with the system-level controlling mechanism – the microcontroller.

Table 1.3: Hexadecimal to ASCII Text Conversions

Base-16	ASCII	Base-16	ASCII	Base-16	ASCII	Base-16	ASCII
00	NULL	20	Space	40	@	60	‘
01	SOH	21	!	41	A	61	a
02	STX	22	"	42	B	62	b
03	ETX	23	#	43	C	63	c
04	EOT	24	$	44	D	64	d
05	ENQ	25	%	45	E	65	e
06	ACK	26	&	46	F	66	f
07	BEL	27	’	47	G	67	g
08	BS	28	(	48	H	68	h
09	HT	29	)	49	I	69	i
0A	LF	2A	*	4A	J	6A	j
0B	VT	2B	+	4B	K	6B	k
0C	FF	2C	,	4C	L	6C	l
0D	CR	2D	-	4D	M	6D	m
0E	SO	2E	.	4E	N	6E	n
0F	SI	2F	/	4F	O	6F	o
10	DLE	30	0	50	P	70	p
11	DC1	31	1	51	Q	71	q
12	DC2	32	2	52	R	72	r
13	DC3	33	3	53	S	73	s
14	DC4	34	4	54	T	74	t
15	NAK	35	5	55	U	75	u
16	SYN	36	6	56	V	76	v
17	ETB	37	7	57	W	77	w
18	CAN	38	8	58	X	78	x
19	EM	39	9	59	Y	79	y
1A	SUB	3A	:	5A	Z	7A	z
1B	ESC	3B	;	5B	[	7B	{
1C	FS	3C	<	5C	\	7C	|
1D	GS	3D	=	5D	]	7D	}
1E	RS	3E	>	5E	^	7E	~
1F	US	3F	?	5F	_	7F	DEL

With an embedded processor at the heart of each system, both electrical engineering and computer science concepts are essential to creating a sound design. We began with the concept of using analog voltage levels to represent discrete binary values '0' and '1', which are also the symbols of the binary number system. As a result, all the concepts present in Boolean Algebra are also available in hardware and software development in the form of digital logic. A few fundamental logic operations provide the basis for building large circuits of logic.

Because base-2 is all we can implement in hardware directly, we introduce several coding schemes used to represent other number systems given multidimensional vectors of 0'a and 1's. Among these are the systems for representing unsigned integers, signed integers, floating-point numbers (i.e., the reals), and printable text characters. We complete this chapter with one final example to convey the following important concept: **the only real thing in a microcontroller are the voltages used to represent individual bits; how we interpret those strings of bits is completely based on a higher-level context.**

Example 1.28 Suppose we want to store the information $\{123\}_{10}$. We can do so using the following bit strings when interpreted with the associated context. Unsigned is 01111011, two's complement is 01111011, BCD is 0001, 0010, 0011 and ASCII is 00110001, 00110010, 00110011. Next, consider the information $\{-12\}_{10}$. Neither unsigned nor BCD codes are able to store this information. Two's complement is 11110100 and ASCII is 00101101, 00110001, 00110010.

The importance of this example is that any string of bits may exist in our processor. The meaning of the bits is dependent on the context being used to interpret the bit vector.

PROBLEMS

1.1 Given a digital system with $V_{DD} = 5\text{V}$, what analog-to-digital threshold T should be selected to maximize the noise margins. You should assume an ideal model where $x < T \Rightarrow 0$ and $x > T \Rightarrow 1$.

1.2 Given a digital system with $V_{DD} = 1.8\text{V}$, and an analog-to-digital threshold $T = 1.0\text{V}$ such that $x < T \Rightarrow 0$ and $x > T \Rightarrow 1$. Sketch the input/output relationship for $0 \leq V_{\text{in}} \leq 1.8\text{V}$ where $V_{\text{out}} = 0\text{V}$ for '0' and $V_{\text{out}} = 1.8\text{V}$ for '1'.

1.3 Given the system in problem 1.2, state the bit that is mapped to from each voltage.

(a) 0.1V

(b) 1.73267589V

(c) 0.99V

(d) 1.01V

1.4 Given the system in problem 1.2, state how much analog voltage noise can be tolerated on both the low and high logic levels.

1.5 Given $\{01110110110100011001101111110000\}_2$, determine the equivalent number in each base.

(a) Decimal

(b) Hexadecimal

1.6 Given $\{24687531\}_{16}$, determine the equivalent number in each base.

(a) Decimal

(b) Binary

1.7 Given $\{24687531\}_{10}$, determine the equivalent number in each base.

(a) Hexadecimal

(b) Binary

1.8 Using 8-bit bytes, show how to represent 123. Clearly state the byte values using hexadecimal, and the number of bytes required for each context. Simply indicate the case if the code is not able to represent the information.

(a) Unsigned integer

(b) Two's complement

(c) BCD

(d) ASCII

1.9 Using 8-bit bytes, show how to represent -123. Clearly state the byte values using hexadecimal, and the number of bytes required for each context. Simply indicate the case if the code is not able to represent the information.

(a) Unsigned integer

(b) Two's complement

(c) BCD

(d) ASCII

1.10 Using 8-bit bytes, show how to represent 56,789. Clearly state the byte values using hexadecimal, and the number of bytes required for each context. Simply indicate the case if the code is not able to represent the information.

(a) Unsigned integer

(b) Two's complement

(c) BCD

(d) ASCII

(e) IEEE single precision

1.11 Using 8-bit bytes, show how to represent -6,543. Clearly state the byte values using hexadecimal, and the number of bytes required for each context. Simply indicate the case if the code is not able to represent the information.

(a) Unsigned integer

(b) Two's complement

(c) BCD

(d) ASCII

(e) IEEE single precision

1.12 Using 8-bit bytes, show how to represent 1.23456. Clearly state the byte values using hexadecimal, and the actual number represented with 7 fractional digits.

(a) IEEE single precision

(b) IEEE double precision

1.13 Using 8-bit bytes, show how to represent -65.43210. Clearly state the byte values using hexadecimal, and the actual number represented with 5 fractional digits.

(a) IEEE single precision

(b) IEEE double precision

CHAPTER 2

ANSI C

2.1 INTRODUCTION

One of the most valuable tools an embedded engineer can have at their disposal is knowledge of the C programming language. So many aspects of the technological industry change within a very short time. As a result, it can seem as though many of the industry-standard tools learned in the classroom today are obsolete within a few years of obtaining an engineering position. Interestingly, this is not true of the C programming language, which was introduced in the 1970's, and is still the most often-used high-level language in embedded systems. This means that even after 40 years, engineers are still able to develop systems using the same reliable tool. This book is really born from the fact that I believe C is a cornerstone for every undergraduate computer and electrical engineer. Once you know the material in this chapter, you will be employable regardless of the latest microprocessor lifecycle.

Most information presented within this chapter comes from notes taken from Kernighan and Ritchie (1989). All additional material is provided from personal work experience (14 years and counting).

2.1.1 BACKGROUND

C, influenced from BCPL and B in 1970, is considered a "low-level" high-level language, meaning you can access anything using C that you can in the platform's native assembly language. In general, use of a high-level language reduces the amount of knowledge required about the underlying hardware architecture.

One of the original uses of C was in the development of the UNIX operating system and various compilers. In 1983, the American National Standards Institute (ANSI) established a definition of C, which eventually led to *ANSI C* defined in 1988. As a result, any compiler that claims ANSI compliance must pass a series of tests. ANSI C is 100% portable code meaning that no matter what the target hardware platform is, if the software is written in pure ANSI C then it can be used to create an executable program.

Example 2.1 Question: Why can't Microsoft take Word and just re-compile it to run on MacOSX or Linux? Answer: Aside from various marketing reasons, the software is not pure ANSI C, and so it is not directly portable from one processor to another (or one operating system to another, for that matter).

In fact, ANSI C defines a pretty small set of rules, governing syntax, types, function calls, methods to include external libraries, and not much else. However, ANSI C is always at the core of any C compiler. If we write software using **only ANSI C**, then it will build to run on any target platform that has an ANSI C compiler available.

This is a nice improvement over software written using machine-specific assembly language. As the name implies, every microprocessor has its own unique set of *machine instructions*, which are bytes used to tell the processor which functions to perform. To help human software developers, manufacturers create an *assembly language* which uses meaningful words to represent the machine-understandable codes. The benefit of writing software in assembly language is that an engineer has direct control over every aspect of the processor. The drawback is that any program written for a processor must be translated by hand to run on any other processor since all assembly languages are unique (although they are similar enough to allow porting). Thus, any software written entirely in ANSI C can be executed on any processor with no extra work required.

Unfortunately, not all code can be 100% ANSI C. For embedded processors, there must be a layer of processor specific code in either C or assembly. It is good design practice to place all of this kind of code into a conceptual interface layer such as in Board Support Packages (BSPs), device drivers, etc. As shown in Fig. 2.1, one of the goals of a software project should be to place as much code into the *Application Layer* as possible to reduce the amount of effort necessary for porting code to other platforms. The application layer needs to be written in some high-level language. Because

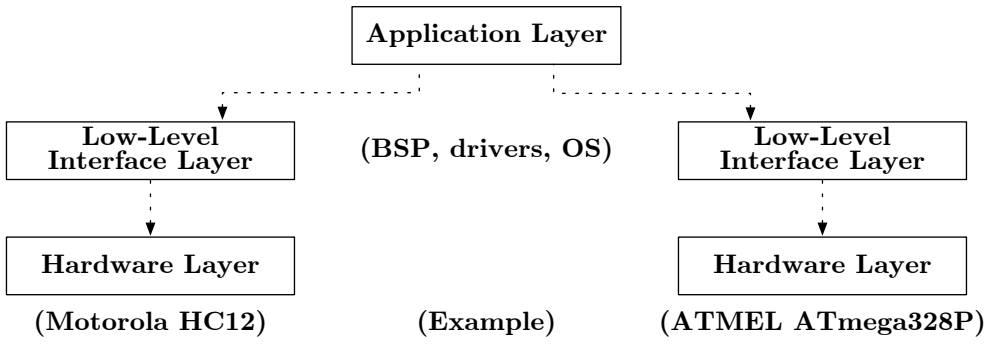


Figure 2.1: Conceptual software layers.

C provides good control and access to low-level functionality, it is very often used. The *Low-Level Interface Layer* generally contains drivers necessary to access platform-specific features. There might be a great deal of processor-specific assembly in this layer, although C is often able to generate efficient and maintainable code and so will be used here as well.

2.2 ESSENTIAL ELEMENTS OF THE LANGUAGE

Consider the following example.

Example 2.2

```
#include <stdio.h>

void main (void)
{
   printf ("Hello, World!\n");
}
```

A C program consists of *functions*, i.e., methods, routines, procedures, etc., and *variables*. Functions contain *statements* that specify the computing operations to be done. Variables store values used in the computations. In Ex. 2.2, `main()` and `printf()` are both functions. Note that typically, `main()` is always the starting point for a program, but, ultimately, the entry point is controlled by the linker program.

The line `#include <stdio.h>` tells the compiler to include the header file `stdio.h` prior to compiling the program. The contents of all included files are copied into the present source file at the location of the inclusion by the preprocessor, then the compiler operates on the result.

One method of communicating data between functions is for the calling function to pass a list of *arguments* to the function it calls. The arguments are listed via the comma-delimited items within the function parenthesis "()".

Example 2.3

```
SomeFunction (a, b, c, "Hi", d, e);
```

In Ex. 2.3 six arguments, including five previously declared variables `a` through `e`, are passed into the function named `SomeFunction`. The fourth argument in the list is the address of a constant null-terminated string of bytes "Hi". In Ex. 2.2, `main (void)` declares to the compiler that `main()` does not expect any arguments. Note that while it is poor C practice, you might see `SomeFunction()`, which is equivalent to `SomeFunction(void)`; however, this seems to be fairly common C++ practice.

The statements of a function are all within a *block* of code. Blocks are all defined by curly braces `{}`. Referring to Ex. 2.2, `main()` has only one statement, a function call to `printf()` with a single argument, the address of a string of bytes. The function `printf()` was not written by us; it is a *library function* that is defined in the header file `stdio.h`, and ultimately linked in by the linker. Hence, there must be an object library or object file somewhere that contains the object code for `printf()`.

2.3 FORMATTED OUTPUT

Often in embedded environments, we have to rely completely on `printf()` in order to look at variable values, which is a fundamental component in the debugging process. The `fprintf()` function provides formatted output conversion.

Example 2.4

```
int fprintf (FILE *stream, const char *format, ...)
```

This function converts and writes output to `stream` under the control of `format`. The return value is the number of characters written, or negative if an error occurred. The format string contains two types of objects.

1. Ordinary characters, which are copied to the output stream, and
2. *conversion specifications*, each of which causes conversion and printing of the next successive argument to `fprintf()`.

Each conversion specification begins with the character % and ends with a conversion character. Between the % and the conversion character there may be, **in the following order**:

- Flags (in any order), which modify the specification, are listed in Table 2.1.

Table 2.1: Formatted Printing Conversion Specification Flags

Flag	Modification
-	specifies left adjustment of the converted argument in its field
+	specifies that the number will always be printed with a sign
space	if the first character is not a sign, a space will be prefixed
0	for numeric conversion, specifies padding to the field width with leading zeros
#	specifies an alternative output form (e.g., for `x`, `0x` will be prefixed to a non-zero result)

- A number specifying a minimum field width. The converted argument will be printed in a field at least this wide, and wider if necessary. If the converted argument has fewer characters than the field width, it will be padded on the left (or right, if left adjustment has been requested) to make up the field width. The padding character is normally space, but `0` if the zero padding flag is present.
- A period, which separates the field width from the precision.

- A number, the precision, that specifies the maximum number of characters to be printed from a string, or the number of digits to be printed after the decimal point for `e` or `f` conversions, or the number of significant digits for `g` conversion, or the minimum number of digits to be printed for an integer (leading `0`s will be added to make up the necessary width).

- A length modifier `h`, `l` or `L`. `h` indicates that the corresponding argument is to be printed as a `short` or `unsigned short`; `l` indicates that the argument is a `long` or `unsigned long`; `L` indicates that the argument is a `long double`.

Width or precision or both may be specified as `*`; in which case, the value is computed by converting the next argument(s), which must be `int`.

Example 2.5

```
printf ("%.*s", max, theString);
```

will output at most `max` characters from `theString`.

The conversion characters and their interpretations are listed in Table 2.2.

Table 2.2: Formatted Printing Conversion Characters

Char	Type	Interpretation
`d,i`	`int`	signed decimal notation
`o`	`int`	unsigned octal notation (without leading `0`)
`x`	`int`	unsigned hexadecimal notation (without leading `0x`, uses `abcdef`)
`X`	`int`	unsigned hexadecimal notation (without leading `0x`, uses `ABCDEF`)
`u`	`int`	unsigned decimal notation
`c`	`int`	single character, after conversion to `unsigned char`)
`s`	`char *`	characters from the string until a `\0` or precision is reached)
`f`	`double`	decimal notation of the form $[-]mmm.ddd$; precision controls ds
`e,E`	`double`	decimal notation of the form $[-]m.ddd$`e`$\pm xx$; precision controls ds
`g,G`	`double`	selects the best choice between `%e` and `%f`
`p`	`void *`	print as a pointer (implementation-dependent representation)
`n`	`int *`	the number of characters output so far via this `printf()` is copied into the argument; no argument is converted
`%`		print a `%`; no argument is converted

If the character after the `%` is not a conversion character, the behavior is undefined. The character constants and their meanings are listed in Table 2.3.

Other forms of `fprintf()` include the functions in Ex. 2.6-Ex.2.8.

Table 2.3: Formatted Printing Character Constants

Control Char	Output	Description
\n	NL (LF)	newline
\t	HT	horizontal tab
\v	VT	vertical tab
\b	BS	backspace
\r	CR	carriage return
\f	FF	formfeed
\a	BEL	audible alert
\\	\	backslash
\?	?	question mark
\'	'	single quote
\"	"	double quote
\o*ooo*	*ooo*	octal number
\x*hh*	*hh*	hexadecimal number

Example 2.6

```
int printf (const char *format, ...)
```

is equivalent to

```
int fprintf (stdout, ...)
```

Example 2.7

```
int sprintf (char *s, const char *format, ...)
```

is the same as `printf()` except the output is written into the memory defined by the string `s`, terminated with a `\0`. Note `s` must be big enough to hold the result or memory will be corrupted.

Example 2.8

```
int snprintf (char *s, int length, const char *format, ...)
```

is the same as `sprintf()` except the number of characters written to the string is limited to at most `length`. Note this is not a function in the original ANSI C, but it is usually available and a nice choice for embedded systems to help prevent overflowing an array.

A simple example usage of `printf()` is presented in Ex. 2.9.

Example 2.9

```
int x = 78;
unsigned long y = 93;
float z = 12.34;

printf("variables x = %d, y = %lu and z = %.2f\nnote x = %#2.2x in
    hex\n", x, y, z, x);
```

this code would result in the output

```
variables x = 78, y = 93 and z = 12.34
note x = 0x4e in hex
```

The format string is managed in the following way.

- Characters are simply written out until a % conversion indicator is encountered;

  `printf("variables x = %d, y = %lu and z = %.2f\nnote x = %#2.2x in hex\n", x, y, z, x);`

- When the %d is seen, the first variable after the format string is interpreted as a signed decimal value because of the d character conversion code. So 78 is printed out due to the first occurrence of x;

  `printf("variables x = %d, y = %lu and z = %.2f\nnote x = %#2.2x in hex\n", x, y, z, x);`

- The next characters, y =, are output

  `printf("variables x = %d, y = %lu and z = %.2f\nnote x = %#2.2x in hex\n", x, y, z, x);`

- Until %lu is seen, at which point, the next variable in the argument list is interpreted as a long unsigned decimal value, and 93 is output;

  `printf("variables x = %d, y = %lu and z = %.2f\nnote x = %#2.2x in hex\n", x, y, z, x);`

- This is followed by the output of and z =

  `printf("variables x = %d, y = %lu and z = %.2f\nnote x = %#2.2x in hex\n", x, y, z, x);`

- Until %.2f is seen, at which time, the next variable in the argument list is interpreted as a double value, and the additional precision restricts the output to 12.34;

  `printf("variables x = %d, y = %lu and z = %.2f\nnote x = %#2.2x in hex\n", x, y, z, x);`

- The next character parsed is the constant \n, which moves to a new line;

  `printf("variables x = %d, y = %lu and z = %.2f\nnote x = %#2.2x in hex\n", x, y, z, x);`

- Which is followed by the next set of characters note x =

  `printf("variables x = %d, y = %lu and z = %.2f\nnote x = %#2.2x in hex\n", x, y, z, x);`

- Until %#2.2x is seen, where the final variable in the list is interpreted as hexadecimal with a width and precision of 2 characters, so 0x4e is output. Note the 0x is printed as a result of the # flag in the conversion code;

```
printf("variables x = %d, y = %lu and z = %.2f\nnote x = %#2.2x in hex\n", x, y, z, x);
```

- The final part of the format string finishes the example by outputting in hex and a final new line character.

```
printf("variables x = %d, y = %lu and z = %.2f\nnote x = %#2.2x in hex\n", x, y, z, x);
```

2.4 VARIABLES AND ARITHMETIC EXPRESSIONS

Consider Ex 2.10.

Example 2.10

```
void main (void)
{
   int farenheit;
   int celsius;
   int lower;
   int upper;
   int step;

   lower = 0;   /* lower limit */
   upper = 300;
   step = 20;
   farenheit = lower;
   while (farenheit <= upper)
   {
      celsius = 5 * (farenheit - 32) / 9;
      farenheit = farenheit + step;
      printf ("%d F = %d C\n", farenheit, celsius);
   }
}
```

Note that anything between /* and */ are *comments*, and ignored by the compiler. Additionally, C++ allows for *inline comments* where anything after // is ignored by the compiler until a new line character. When used properly, comments are useful for generating maintainable software. Comments can contain any prose desired, which allows a developer the ability to document interesting aspects of the program. All comments are discarded by the compiler, so their presence is strictly used for documentation purposes.

In C, all variables must be declared before they are used. Within functions, this is always the first set of statements. A *declaration* announces the properties of variables using the syntax *type* name followed by a comma-delimited list ending with a `;`.

Example 2.11 Note that the compiler allows for the following three-variable declaration.

```
int x, y, z;
```

But this is bad coding practice. For maintainable code listings, use the one-variable-per-line rule instead.

```
int x;
int y;
int z;
```

The possible standard types are listed in Table 2.4. Note that the `char` type is actually an integer type that happens to be useful when managing english text. It is a misconception that `char` variables are used for ASCII text only.

Table 2.4: Standard C Types

Type	Size Info	Range
`char`	(usually 8-bits)	{−128, ..., 127}
`short`	(usually 16-bits)	{−32, 768, ..., 32, 767}
`long`	(usually 32-bits)	{−2, 147, 483, 648, ..., 2, 147, 483, 647}
`int`	integer	?
`float`	single-precision floating point	?
`double`	double-precision floating point	?

Each compiler is free to choose appropriate sizes for its own hardware, but it is required for ANSI that

$$\begin{aligned} \texttt{short},\texttt{int} &\geq 16\text{bits}, \\ \texttt{long} &\geq 32\text{bits}, \\ \texttt{short} &\leq \texttt{int}, \\ \texttt{int} &\leq \texttt{long}. \end{aligned}$$

Example 2.12 On the Texas-Instruments C55x DSP family, a `char` is 16-bits, i.e., there are no 8-bit units on that architecture. Usually, compilers will generate 32-bit `int`s for Intel microprocessors. Some microcontrollers are small enough that 16-bit `int`s will be used instead. Software should be developed without assuming certain type sizes when possible.

Table 2.5: Standard Unsigned Integer C Types

Type	Size Info	Range
`unsigned char`	(usually 8-bits)	{0, ..., 255}
`unsigned short`	(usually 16-bits)	{0, ..., 65, 535}
`unsigned long`	(usually 32-bits)	{0, ..., 4, 294, 967, 295}
`unsigned int`	integer	?

In addition to the standard types, the `unsigned` qualifier may be added to integer types as listed in Table 2.5.

The remaining ANSI C types available are compound from the previously listed basic types and each other pointers and arrays, structures and unions. They will be discussed in Sec. 2.8, Sec. 2.11 and Sec. 2.12.

It is easy to be lazy when declaring variables. This is especially true for software developers that are already used to writing source code for personal computer applications. Because of the seemingly unlimited amount of memory available, programmers can fall into the bad habit of declaring every variable as an `int`. However, doing so might be 4-times wasteful, such as if a 32-bit `int` is used as a small loop counter where an 8-bit `char` could have been used. Be aware of type sizes, and always use the smallest one you can, especially when working with embedded microcontrollers, which are resource-limited.

Returning to Ex. 2.10, after the declarations are *assignment statements*.

```
lower = 0;   /* lower limit */
upper = 300;
step = 20;
farenheit = lower;
```

Here, = is not a mathematical statement of equality. Rather, it is an assignment that evaluates the expression on the right-hand-side (RHS) and places the resulting value into the variable on the left-hand-side (LHS). All statements are terminated by a semicolon `;`.

Finally, after the assignment statements is a *while loop*, which will be discussed in Sec. 2.5.4.2.

2.4.1 VARIABLE NAMES

Legal variable labels are composed of alphanumeric characters and _. Additionally, the first character must be an alphabetic letter or _. Labels in C are case sensitive, so `Done` and `done` are two different variables. Also, you can not use keywords as variable names.

Example 2.13

```
int g_interruptCounter1; /* valid and self-documenting */
int 2lines;              /* not valid */
int _x;                  /* valid, but not very meaningful */
```

```
int int;                    /* not valid */
int float;                  /* not valid */
int if;                     /* not valid */
int while;                  /* not valid */
```

As an aside, there are varying degrees of quality regarding source-code listings. Consider the labels used to represent variables and functions. Because the labels can be nearly any combination of alphanumeric letters, we can generate *self-documenting code* by using meaningful names. In fact, by using overly descriptive labels, there is no need for comments that tend to fall out-of-sync with the instructions they attempt to describe.

2.4.2 TYPE CONVERSIONS

C allows for assigning variables to each other even of different types. But what about the storage difference?

Example 2.14

```
char c;
short s;

/* suppose s is assigned some value prior to this assignment */
c = s;

/* suppose c is assigned some value prior to this assignment */
s = c;
```

Suppose that in Ex. 2.14 a `char` variable is 8 bits and a `short` variable is 16 bits. Then the assignment `s = c` poses no problem; the extra bits are simply *sign-extended* as shown in Fig. 2.2. To understand why sign-extension works (or is needed), first consider the case when $c = 1$ which is

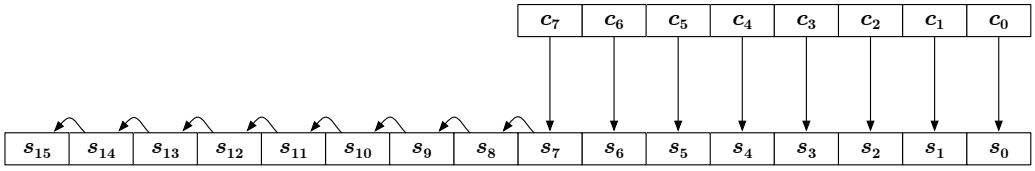


Figure 2.2: An 8-bit signed integer value is loaded into a 16-bit storage location. The correct two's complement value is generated by performing sign extension.

stored as

$$\{c\}_2 = (0, 0, 0, 0, 0, 0, 0, 1).$$

As the figure implies, the first step is copying the bits from c into the lower 8 bits of s,

$$\{s\}_2 = (x, x, x, x, x, x, x, x, 0, 0, 0, 0, 0, 0, 0, 1).$$

Then, for s to hold the correct value of $s = 1$, the sign-bit of $c_7 = 0$ is copied into the upper 8 bits of s resulting in

$$\{s\}_2 = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1),$$

or $s = 1$ as required. Now consider the case when $c = -1$ which is stored in two's complement as

$$\{c\}_2 = (1, 1, 1, 1, 1, 1, 1, 1)$$

(remember, two's complement negation is found via the "invert and add 1" process). Following the same process in the figure, the bits are copied from c into the lower 8 bits of s,

$$\{s\}_2 = (x, x, x, x, x, x, x, x, 1, 1, 1, 1, 1, 1, 1, 1).$$

Now, for s to hold the correct value of $s = -1$, the sign-bit of $c_7 = 1$ is copied into the upper 8 bits of s resulting in

$$\{s\}_2 = (1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)$$

which is the correct two's complement representation for $s = -1$.

Alternatively, the assignment `c = s` as shown in Fig. 2.3. In this case, the destination doesn't

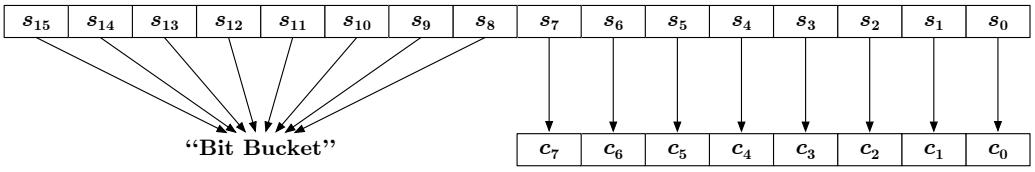


Figure 2.3: A 16-bit signed integer value is loaded into an 8-bit storage location. The final value is forced to ignore the upper 8-bit values of the source.

have enough room to hold the entire source value. As a result, the lower 8 bits of s are copied into c and the higher 8 bits of s are discarded into the "bit bucket" virtual trash can. If any of the upper 8 bits contain information necessary for the correct meaning of the value stored in the variable, then *overflow* (OV) occurs. Overflow happens any time the destination variable does not have enough room to contain the true meaning of the information being loaded.

Example 2.15 Consider the following assignments between an 8-bit, signed integer `char c` and a 16-bit, signed integer `short s`.

	Source		Destination		
Assignment	Base-10	Base-2	Base-10	Base-2	OV
`c = s`	123	(0000, 0000, 0111, 1011)	123	(0111, 1011)	
`s = c`	123	(0111, 1011)	123	(0000, 0000, 0111, 1011)	
`c = s`	145	(0000, 0000, 1001, 0001)	-111	(1001, 0001)	✓
`s = c`	-111	(1001, 0001)	-111	(1111, 1111, 1001, 0001)	
`c = s`	9780	(0010, 0110, 0011, 0100)	52	(0011, 0100)	✓

Next, consider how unsigned integer variables are assigned between different sized types.

Example 2.16

```
unsigned char c;
unsigned short s;

/* suppose s is assigned some value prior to this assignment */
c = s;

/* suppose c is assigned some value prior to this assignment */
s = c;
```

Suppose that variables in Ex. 2.16 use the same number of bits as those from Ex. 2.14. As in the signed case, the assignment `s = c` poses no problem; the extra bits are cleared to zero as shown in Fig. 2.4. By clearing the upper 8 bits, it is guaranteed that the destination will always be a positive

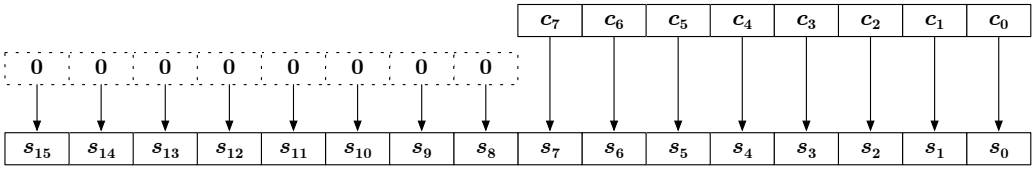


Figure 2.4: An 8-bit unsigned integer value is loaded into a 16-bit storage location. The correct unsigned value is generated by clearing the most significant bits to zero.

value.

The assignment `c = s` is the same as the signed case shown in Fig. 2.3.

Example 2.17 Consider the following assignments between an 8-bit, unsigned integer `char c` and a 16-bit, unsigned integer `short s`.

	Source		Destination		
Assignment	Base-10	Base-2	Base-10	Base-2	OV
c = s	123	(0000, 0000, 0111, 1011)	123	(0111, 1011)	
s = c	123	(0111, 1011)	123	(0000, 0000, 0111, 1011)	
c = s	145	(0000, 0000, 1001, 0001)	145	(1001, 0001)	
s = c	145	(1001, 0001)	145	(0000, 0000, 1001, 0001)	
c = s	9780	(0010, 0110, 0011, 0100)	52	(0011, 0100)	✓

C is considered a *weakly-typed* language, in that you can write source code that assigns variables from one type to another, and the compiler must do its best to figure out what you meant to have happen. By comparison, there are high-level languages, such as Ada, that are *strongly-typed* languages where the compiler will never allow these conversions to occur. Depending on who you ask, this is either an asset or a liability of the language. Generally, it allows more flexibility, but it can also allow for unintentional behavior. The best thing you can do is keep in mind how the information exists in variables and understand the result of assigning one type to another.

When assigning one type to another, the C compiler will usually only give a warning about loss of precision, although many C++ compilers will go so far as generating an error condition that will halt compilation. Assuming you know what you are doing, especially when the compiler is giving errors, you can force the contents of one variable into another with a *type cast*. A type cast is an explicit directive to the compiler that you are interpreting the bits stored at a memory location in a different context. In this way, you can convert any type into any other type. Type casts are stated by placing the target type in parenthesis on the left of the source variable. They tell the compiler to either expand or contract the space used for that variable. Additionally, in the case of converting between floating-point types and integer types, the bits may be re-interpreted all together. In all cases, the original variable is not changed; instead, temporary space is used while evaluating the expression.

Example 2.18

```
char c;
short s;

/* suppose s is assigned some value prior to this assignment */
c = (char) s;

/* suppose c is assigned some value prior to this assignment */
s = (short) c;
```

Consider Ex. 2.19, which presents a typical problem suited for type casts that arises, especially in Digital Signal Processing (DSP).

Example 2.19

```
short tap;
short sample;
short filteredSample;

/* skipping code where variables are loaded with values */

filteredSample = tap * sample;
```

PROBLEM: all variables are 16-bit. If `tap == 0xFFFF` and `sample == 0x7654`, we would require a 32-bit result. But when filtering a set of 16-bit values, it is likely we want 16-bit results (for maybe more filtering, storage space requirements, etc.) So, we only want the most-significant 16-bit result. But, another problem with this statement is that `tap` and `sample` are both 16-bit variables. So, the result of the multiplication is the 16-bit overflowed result. One solution is via extra variables.

```
#define EFFICIENT_DIVIDE_BY_65536 16

short tap;
short sample;
short filteredSample;
long longTap;
long longSample;
long longResult;

/* skipping code where variables are loaded with values */

longTap = tap;
longSample = sample;
longResult = longTap * longSample;
longResult = longResult >> EFFICIENT_DIVIDE_BY_65536;

filteredSample = longResult;
```

But here we have to explicitly define extra memory and do a bunch of work. An easier solution is to use type casts.

```
#define EFFICIENT_DIVIDE_BY_65536 16

short tap;
short sample;
short filteredSample;

/* skipping code where variables are loaded with values */

filteredSample = (short) ((((long) tap) * ((long) sample)) >>
    EFFICIENT_DIVIDE_BY_65536);
```

2.4.3 CONSTANTS

Several methods are available for specifying *constant* values in the C programming language. The first method is immediate placement of the desired value.

Example 2.20 Decimal constant values include the following statements.

```
1234   /* is an int */
1234l  /* is a long */
1234L  /* is a long */
1234u  /* is an unsigned int */
1234U  /* is an unsigned int */
1234ul /* is an unsigned long */
1234UL /* is an unsigned long */
```

Note that usually you don't need to bother with the `L`'s or `U`'s, as the compiler manages this automatically. A leading `0` on a constant means octal (base-8). A leading `0x` on a constant means hexadecimal (base-16). It is a misconception that a leading `0b` on a constant means binary (base-2), but this is **not true for ANSI C**. However, some compilers do support some variations of binary constants.

Example 2.21

Base-10	Base-8	Base-16	Base-16
31	037	0x1f	0x1F
5349	012345	0x14e5	0x14E5

It is considered good coding practice to define all constants (except `0` and `1`, where meaning is obvious) in a header file or at the top of the current source file. This is done by creating a *macro* with the preprocessor directive statement `#define`. Before compilation occurs, the preprocessor replaces all occurrences of the macro label with its defined value. The compiler parses the result, just like the `#include` directives. Using constant macros is analogous to using meaningful variable labels and function names. Doing so leads to self-documenting code and, ideally, the near-elimination of all comments.

Example 2.22

```
#define SPECIAL_MAGIC_NUMBER   3

/* skipping many lines */

   int primeNumber;

/* skipping many lines */

      primeNumber = SPECIAL_MAGIC_NUMBER;
```

Another way to define constants is via an enumeration. In Ex. 2.23, the first statement defines the constant FALSE, assigns it the default starting value of 0, and then increments the value by one for each successive element in the list; in this case, TRUE is defined as 1. Similarly, the constant JAN is defined with an explicit value of 1, and then increments the value by one for each successive element in its list. Thus DEC is defined as 12.

Example 2.23

```
enum boolean {FALSE, TRUE};
enum months {JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV,
    DEC};
```

It is considered good coding practice to create a new type of an enumeration using typedef. Doing so gives the compiler the ability to warn you if variable types are used incorrectly. The code in Ex. 2.24 begins by creating a new type, which can be used just like int, char, etc.

Example 2.24

```
typedef enum
{
   FALSE,
   TRUE
} Boolean;

/* skipping many lines */

void SomeFunction (void)
{
   int i;
   Boolean answer;
   char j;

   /* skipping many lines */

   answer = FALSE;
   while (answer != TRUE)
   {
      /* skipping many lines */
   }
}
```

Note that in C++, the typedef is not really required any more. Just declaring the enum will automatically create the type.

Yet another constant method is achieved via the const qualifier. The code in Ex. 2.25 declares two variables, both of which are constants and so neither can be modified (without cheating). The

variable `c_table` is a block of 10 consecutive `shorts`, i.e., an array, and it is not able to be modified in the code via the assignment operator. Similarly, the variable `c_pi` is assigned the constant value `3.1416` and can not be assigned anything else (using the assignment operator).

Example 2.25

```
#define NUMBER_OF_TABLE_ENTRIES 10
const short c_table[NUMBER_OF_TABLE_ENTRIES] =
{
   1, 2, 3, 4, 5, 6, 7, 8, 9, 0
};
const float c_pi = 3.1416;
```

2.4.4 ARITHMETIC OPERATORS

Consider the list of possible arithmetic operators presented in Table 2.6.

Table 2.6: Arithmetic Operators

Operator	Operation
+	addition
-	subtraction
*	multiplication
/	division (for non-floats, quotient is returned)
%	modulo (for non-floats, remainder is returned)

When the operands are integer types, the division (`q = b / a;`) and modulo (`r = b % a;`) operations return q and r as in

$$b \div a \Rightarrow b = aq + r.$$

ANSI C does define an order of precedence regarding operators. But you should **never** rely on it; it is **much** better coding practice to force the precedence you intend by using parenthesis.

Example 2.26

```
/* Hope you know precedence */
if (year % 4 == 0 && year % 100 != 0)
/*        1    |     |        1      |     */
/*             2     |              2     */
/*                   3                    */

/* Here the order is obvious */
if (((year % 4) == 0) && ((year % 100) != 0))
```

2.4.5 RELATIONAL AND LOGICAL OPERATORS

Consider the list of possible relational and logical operators presented in Table 2.7.

Table 2.7: Relational and Logical Operators

Operator	Operation
`>`	greater-than
`>=`	greater-than or equal-to
`<`	less-than
`<=`	less-than or equal-to
`==`	equal-to
`!=`	not equal-to
`&&`	and
`||`	or
`!`	unary negation (non-zero → `0`, `0` → `1`)

2.4.6 INCREMENT AND DECREMENT OPERATORS

Consider the list of possible increment and decrement operators presented in Table 2.8.

Table 2.8: Increment and Decrement Operators

Operator	Operation
`++`	increment value by 1; either before or after the variable is used
`--`	decrement value by 1; either before or after the variable is used

If the operator is placed on the left-hand-side of the variable (e.g., `++x`), then the variable is changed before it is used in the rest of the expression. If the operator is placed on the right-hand-side of the variable (e.g., `x--`), then the variable is changed after it is used in the rest of the expression.

Example 2.27

Statement	x Before	n After	x After
`n = x++;`	10	10	11
`n = ++x;`	10	11	11
`n = x--;`	10	10	9
`n = --x;`	10	9	9

Table 2.9: Bitwise Operators

Operator	Operation
`&`	AND (boolean intersection)
`|`	OR (boolean union)
`^`	XOR (boolean exclusive-or)
`<<`	left shift
`>>`	right shift
`~`	NOT (boolean negation, i.e., ones' complement)

2.4.7 BITWISE OPERATORS

Consider the list of possible bitwise operators presented in Table 2.9.

Bitwise operations occur on a per-bit level; Fig. 2.5 shows the operation `d = c & b`, assuming all three variables are declared as `unsigned char` (assumed to be 8-bit elements).

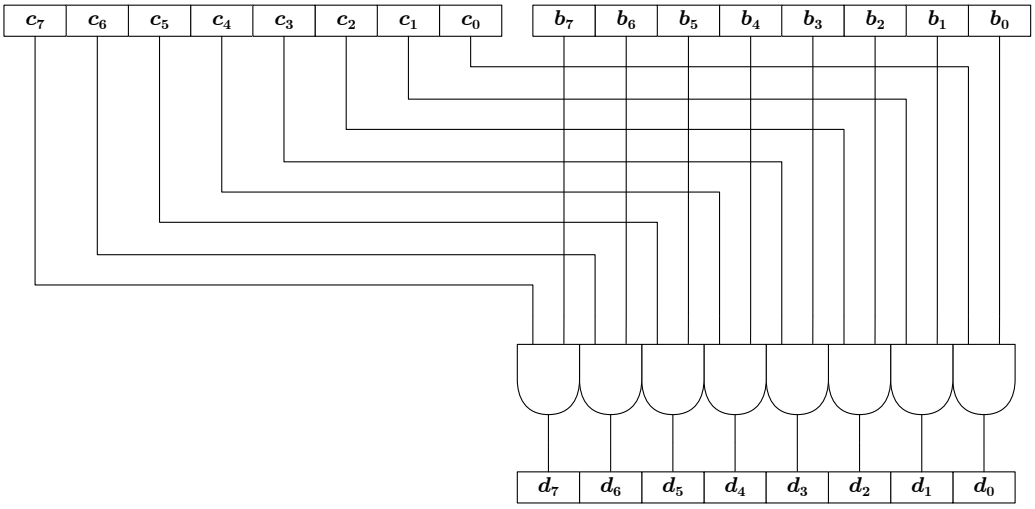


Figure 2.5: A bitwise logical AND operation on two 8-bit unsigned integer locations. The result the boolean intersection between each bit position is stored in the respective location of the target memory location.

Example 2.28

Statement	c	mask	d	Embedded usefulness
`d = (c & mask);`	0x55	0x0F	0x05	Clear bits that are 0 in the mask
`d = (c | mask);`	0x55	0x0F	0x5F	Set bits that are 1 in the mask
`d = (c ^ mask);`	0x55	0x0F	0x5A	Invert bits that are 1 in the mask
`d = (c << 3);`	0x55		0xA8	Multiply by a power of 2
`d = (c >> 2);`	0x55		0x15	Divide by a power of 2
`d = ~c;`	0x55		0xAA	Invert all bits

Note that there is a difference between logical and bitwise operations. Care must be taken to achieve the desired results. All statements in Ex. 2.29 are perfectly legal, so the compiler will not indicate an error (or even a warning).

Example 2.29

Statement	x	y	z After	Operation
`z = (x & y);`	1	2	0	Bitwise AND
`z = (x && y);`	1	2	1	Logical AND
`z = (x | y);`	1	2	3	Bitwise OR
`z = (x || y);`	1	2	1	Logical OR

2.4.8 ASSIGNMENT OPERATORS

Consider the list of possible assignment operators presented in Table 2.10.

Table 2.10: Assignment Operators

Operator	Syntax	Equivalent Operation
`+=`	`i += j;`	`i = (i + j);`
`-=`	`i -= j;`	`i = (i - j);`
`*=`	`i *= j;`	`i = (i * j);`
`/=`	`i /= j;`	`i = (i / j);`
`%=`	`i %= j;`	`i = (i % j);`
`&=`	`i &= j;`	`i = (i & j);`
`|=`	`i |= j;`	`i = (i | j);`
`^=`	`i ^= j;`	`i = (i ^ j);`
`<<=`	`i <<= j;`	`i = (i << j);`
`>>=`	`i >>= j;`	`i = (i >> j);`

2.4.9 CONDITIONAL EXPRESSION

The conditional expression,

$$((expr)\ ?\ trueValue\ :\ falseValue),$$

is a kind-of operator that evaluates any expression and returns a different result based on if the expression is true (i.e., non-zero) or false (i.e., 0).

Example 2.30 The following two examples result in identical results.

```
/* This conditional expression... */
z = (a > b) ? c : d;

/* ...is the same as the following code. */
if (a > b)
{
   z = c;
}
else
{
   z = d;
}
```

Direct usage of this operator is not advised due to its cryptic appearance; the only benefit is compressing source code (not machine code), so it does not help anything (other than job-security seeking engineers). One good application of this expression is via macro definitions such as in Ex. 2.31.

Example 2.31

```
#define MAX(a,b) ((a > b) ? a : b)

/* z will be assigned the maximum of the two values. */
z = MAX(x,y);
```

2.5 CONTROL FLOW

All structural programming languages (even assembly languages) have constructs that allow for an algorithm to execute different instructions depending on various conditions. In general, an expression is executed and compared to zero providing a true or false result. These boolean conditions are one of the cornerstones of microprocessor execution. They can be combined (sometimes this is automatic and hidden) in order to create any number of complex conditions.

2.5.1 IF-ELSE

Consider Ex 2.32.

Example 2.32

```
if (expression)
   /* Performed when expression != 0 */
   statement1
else
   /* Performed when expression == 0 */
   statement2
```

Note that $statement_i$ may be a block of statements surrounded by `{ }`. In fact, it is considered good practice to always use `{ }`, even for single statements. Also note that the `else` is optional.

2.5.2 ELSE-IF

Consider Ex 2.33.

Example 2.33

```
if (expression1)
   /* Performed when expression1 != 0 */
   statement1
else if (expression2)
   /* Performed when (expression1 == 0) and (expression2 != 0) */
   statement2
else
   /* Performed when all previous expressions are 0 */
   statement3
```

Again, all $statement_i$ should be blocks surrounded by `{ }`. There can be as many `else if` blocks as desired. They are evaluated in order from top to bottom, implying priority is at the top.

2.5.3 SWITCH

Consider Ex 2.34.

Example 2.34

```
switch (expression)
{
   case constant-expression1:
      /* Performed when expression == constant-expression1 */
      statements
      break;
```

```
   case constant-expression2:
      /* Performed when expression == constant-expression2 */
      statements
      break;

/* skipping other cases */

   default:
      /* Performed when expression != any constant expression */
      statements
      break;
}
```

The *statements* following the first $constant - expression_i$ label equal to *expression* are executed. Note that `break` is used to prevent the fall-through condition. That is, without the `break` statement, the code in the subsequent `case` will be executed. This is allowed by the compiler, but terrible coding practice. It is much better coding practice to "always" place a `break` at the end of each `case`. If no `case`s equal the *expression*, the statements after the `default` are executed. If no `case`s equal the *expression* and no `default` is listed, nothing will happen.

2.5.4 LOOPS

2.5.4.1 For

Example 2.35

```
for (expression1; expression2; expression3)
   statement
```

2.5.4.2 While

Example 2.36

```
expression1;
while (expression2)
{
   statement
   expression3;
}
```

The two loop styles presented in Ex. 2.35 and Ex. 2.36 are equivalent. Referring to the two loops, usually the following are true.

- $expression_1$ is an initialization assignment (e.g., `i = 0`).
- $expression_2$ is some ending condition (e.g., `i < MAGIC_NUMBER`).
- $expression_3$ is an increment (e.g., `i += MAGIC_INCREMENT`, or `i++`).

2.5.4.3 Do-While

The do-while loop presented in Ex. 2.37 repeats *statement* as long as *expression* `!= 0`. A nice feature of this loop style is that *statement* is always executed at least once prior to the *expression* evaluation.

Example 2.37

```
do
   statement
while (expression);
```

2.5.5 INFINITE LOOPS

Loops that repeat execution "forever" are called *infinite loops*. You might be wondering why such a construct exists, as once the algorithm enters an infinite loop it doesn't exit (until execution is halted by outside force). However, an infinite loop is a very important and fundamental concept necessary to many applications. For example, consider running a web-browser on your computer. You wouldn't want the browser to display some information and then quit after it was done; that would make the application useless. In fact, you want web-browsers (and most applications) to stay running until you issue a command to quit. Thus, at the very lowest level, the web-browser is sitting in an infinite loop, repeatedly waiting for user input and responding accordingly. Embedded applications are fundamentally the same. Imagine how useless an alarm clock that didn't sit in an infinite loop would be.

Example 2.38

```
for (;;)
   statement
```

Example 2.39

```
while (1)
   statement
```

Example 2.40

```
do
   statement
while (1);
```

Especially in an embedded platform, often the outer-most layer of the program is an infinite loop, so that the embedded device will continue to function until some reset condition occurs or power is removed. For example, the Arduino Integrated Development Environment (IDE) hides the `main()` function from the engineer. Instead, they present the two functions `setup()` and `loop()`, which allow the developer to initialize various conditions, and then loop forever. For example, the text in Ex.2.41 is an Arduino-based "Hello world" program as viewed through the IDE.

Example 2.41

```
#define DEBUG_OUTPUT_MESSAGE_MAX_LENGTH 80

void setup()
{
   char message[DEBUG_OUTPUT_MESSAGE_MAX_LENGTH];

   snprintf(message, DEBUG_OUTPUT_MESSAGE_MAX_LENGTH, "Hello, World!\n");

   Serial.begin(9600);
   Serial.write(message);
}

void loop()
{
}
```

The actual C++ source code in Ex.2.42 is generated when the user tries to download the program to an embedded target.

Example 2.42

```
#define DEBUG_OUTPUT_MESSAGE_MAX_LENGTH 80

#include "WProgram.h"
void setup();
void loop();

void setup()
{
   char message[DEBUG_OUTPUT_MESSAGE_MAX_LENGTH];
```

```
    snprintf(message, DEBUG_OUTPUT_MESSAGE_MAX_LENGTH, "Hello, World!\n");

    Serial.begin(9600);
    Serial.write(message);
}

void loop()
{
}

int main(void)
{
    init();

    setup();

    for (;;)
        loop();

    return 0;
}
```

Notice the general structure inside the `main()` function begins by initializing various peripherals, memory, etc., before entering an infinite loop.

2.5.6 MISCELLANEOUS (PLEASE DON'T USE)

The following are a part of ANSI C but should "never" be used for the sake of maintainable code.

2.5.6.1 Break

A `break` allows for the early exit from a loop. It is useful (and actually required for maintainable code) within a `switch` statement so the code in subsequent `case`s is not executed.

Example 2.43 After the loop executes in this example, `i == 5` and `x == 4`.

```
for (i = 0; i < 10; i++)
{
    if (i >= 5)
    {
        /* This will cause the loop to quit when i == 5. */
        break;
    }
    x = i;
}
```

2.5.6.2 Continue

A `continue` is similar to `break` in that when executed the remaining code in the loop block is skipped. However, it differs from `break` in that the loop continues to execute.

Example 2.44 After the loop executes in this example, `i == 10` and `x == 4`.

```
for (i = 0; i < 10; i++)
{
   if (i >= 5)
   {
      /* This will cause a jump to the loop start when i >= 5. */
      continue;
   }
   x = i;
}
```

2.5.6.3 Goto and Labels

Worst of all, `goto` and labels should be avoided at all costs (and can be). However, one place that is riddled with `goto` usage is within Linux device drivers, where a number of conditions can cause an error to occur resulting in the necessary clean-up of any previous allocations. The reality is that `goto`s in these situations could have been avoided, but it is already very established and somewhat common practice. As a result, it is actually better to remain consistent, in spite of the horrible inclusion of `goto`s.

Example 2.45

```
int DeviceDriverProbe(void)
{
/* allocate resource 1 */
   if ( /* a bad condition occurs */ )
   {
      /* This will cause a direct jump to the label below */
      goto errorHandlerResource1;
   }

/* allocate resource 2 */
   if ( /* a bad condition occurs */ )
   {
      /* This will cause a direct jump to the label below */
      goto errorHandlerResource2;
   }

/* many more allocations and other things */

   return (0);
```

```
errorHandlerResource2:
   /* some code to clean up resource 2 */

errorHandlerResource1:
   /* some code to clean up resource 1 */

   return (-1);
}
```

2.6 FUNCTIONS AND PROGRAM STRUCTURES

Functions are isolated blocks of statements that are executed when the function label is called from elsewhere in the program. When the block of code defining the function finishes, program execution returns back to the point from which it was called. Function constructs are familiar in many different languages under a variety of names including routines and procedures.

Good coding practice uses functions to modularize source code, assuming the 7 ± 2 rule is followed. In other words, functions should not be too big nor too small. Of course, there are always exceptions to the rule. Each function definition has the form presented in Ex. 2.46.

Example 2.46

```
return_type functionName (argument declarations)
{
   declarations

   statements
}
```

Note that various parts are optional and may be absent.

Example 2.47 A minimal (and useless) function.

```
smallFunction () {}
```

The function in Ex. 2.47 does nothing and returns nothing. However, it does absorb program space and CPU cycles if called (unless the compiler is "smart" enough to optimize it out).

If the *return_type* is omitted, `int` is assumed. For any non-void *return_type*, the final exiting statement from a function is given in Ex. 2.48.

Example 2.48

```
   return expression;
```

Use *function prototypes* to declare to the compiler the exact *return_type* and argument declarations. Note that this is not required, the compiler can resolve everything without them. However, especially for complex projects with many source files, it is good practice to always specify functions via prototypes, such as in Ex. 2.49.

Example 2.49

```
/* Prototype */
return_type functionName (argument declarations);

/* skip lots of code */

/* actual function */
return_type functionName (argument declarations)
{
   /* skip code */
}
```

2.7 SCOPE RULES

The *scope* of a label is the part of the program within which it can be used. Variables may be used only in the block in which they are declared. This includes any sub-blocks also declared within the block. Any block may contain variable declarations. Variables are not visible to any code outside of the defining block. Global variables are available to all blocks of code.

Use of the `extern` keyword makes labels in one source file available to another source file.

Example 2.50 Let `fileA.c` contain the following code

```
extern int x;
extern void functionName (void);

void main (void)
{
   x = 0;
   functionName();
}
```

Let `fileB.c` contain the following code

```
/* x exists everywhere in this program, including fileA.c */
int x;
void functionName (void);

void functionName (void)
{
   /* i only exists inside this function (including the loop) */
   unsigned char i;

   for (i = 0; i < 10; i++)
   {
      /* j only exists inside this for loop block */
      int j = 23;

      x += j + i;
   }
}
```

It is considered good coding practice to make global functions and variables available via header files, as in Ex. 2.51.

Example 2.51 Let `fileC.h` contain the following code

```
#ifdef _FILE_B
#define FILE_C_EXTERN
#else
#define FILE_C_EXTERN extern
#endif

FILE_C_EXTERN int x;
void functionName (void);
```

Then the files from Ex. 2.50 would use this header file to communicate the global information to each other. Let `fileA.c` contain the following code

```
#include "fileC.h"

void main (void)
{
   x = 0;
   functionName();
}
```

Let `fileB.c` contain the following code

```
#define _FILE_B
#include "fileC.h"
```

```
void functionName (void)
{
   unsigned char i;

   for (i = 0; i < 10; i++)
   {
      int j = 23;

      x += j + i;
   }
}
```

By placing the `#define _FILE_B` in `fileB.c` before the header file is included, it causes the preprocessor to define a blank macro. This causes the compiler to allocate the necessary memory for variable `x`. `fileA.c` simply includes the header file without defining anything, and so `x` is declared as an external variable within that file.

The keyword `static` limits the scope of a global variable to just within the file declaring it.

Example 2.52 Here, `y` is still a global variable, but the compiler will not allow any other files in the build process to have access to it because of the `static` keyword.

```
#define _FILE_B
#include "fileC.h"

/* y is global only within this file */
static int y;

void functionName (void)
{
}
```

Additionally, when `static` is used inside a local block, it makes a variable persistent. However, the variable is still only accessible within that block.

Example 2.53 The first call to the `test()` function initializes `i == 0`. On the next call, `i` is not re-initialized to zero; instead, it contains the value it had when the function exited previously (i.e., `i == 1`).

```
void test (void)
{
   static int i = 0;
   i += 1;
}
```

Note that there are compiler rules regarding scope when two variables are named the same thing in nested blocks.

Example 2.54 Here there are two versions of the variable `i`. They do not refer to the same memory locations – they are different variables with the same name.

```
int i;

void functionName (void)
{
   unsigned char i;

   /* which i is valid in here? */
}
```

To avoid the confusing problem posed in Ex. 2.54, a consistent coding convention is useful for the sake of maintainable code. An example coding convention might include the rules presented in Table 2.11. A set of coding convention rules from an embedded software company is presented in Apx. D.

Table 2.11: Example Coding Convention Rules

Name type	Convention
constant	`CONSTANT`
global	`g_name`
static global	`m_name`
local	`name`

Example 2.55 Consider Ex. 2.54 rewritten using the convention presented in Table 2.11.

```
int g_i;

void functionName (void)
{
   unsigned char i;

   /* no question about i now */
}
```

The keyword `register` advises the compiler that the variable will be accessed a lot and should be placed in a machine register. In ANSI C, the compiler is free to ignore this directive (and often does).

Example 2.56

```
register int x;
```

Some compilers provide the keyword `volatile`, which advises the compiler that the variable may be changed externally, i.e., from outside the CPU, such that the processor can't assume a cached value. Because most programs are completely deterministic, the compiler can analyze the source code and determine situations where it can optimize out instructions and still achieve the intended behavior. Part of this optimization process assumes that values stored in memory locations can only change if the CPU writes to them. Thus, the compiler can remove statements if values haven't been written to explicitly. However, there are many situations in embedded applications when the program needs to access memory-mapped hardware that may have its value change due to external stimuli. If the compiler is not informed about this behavior, it can unintentionally optimize out certain instructions. The result is that real-time hardware values may not be updated by the program, even though C instructions were written to do so. By using the `volatile` keyword on a variable, the compiler is told not assume the memory location hasn't changed. Note that the `volatile` keyword is not part of the ANSI C standard.

Example 2.57

```
volatile int x;
```

As a result of the `volatile` keyword, the compiler is unable to optimize out certain statements due to memory caching. Thus, sometimes it is helpful to use the keyword on a regular variable to trick the compiler into preserving some statements that you don't want removed.

2.8 POINTERS AND ARRAYS

A *pointer* is a variable that contains an address, usually of another variable, but it can be anything in the addressable space. Memory, in general, is just a block of addressable bits that may be manipulated in various-sized groups, but the bits are all physically the same. Generally, the sizes listed in Table 2.12 represent the contiguous memory locations, usually along address boundaries.

Table 2.12: Typical Contiguous Memory Sizes

Type	Number of 8-bit Units	Total Contiguous Bits
`char`	1	8
`short`	2	16
`long`	4	32

Then, a pointer is a group of cells (usually 2 or 4 8-bit cells) that holds an address.

Example 2.58 Suppose the following variables are allocated in the specified order. Let `r` be a pointer that points to `l`, `q` be a pointer that points to `s`, and `p` be a pointer that points to `c`. Also, let `s` be a 16-bit `short`, `l` be a 32-bit `long`, and `c` be an 8-bit `char`. Note that in this hypothetical example, pointers are 32-bit variables meaning the processor has 32-bit addressable space.

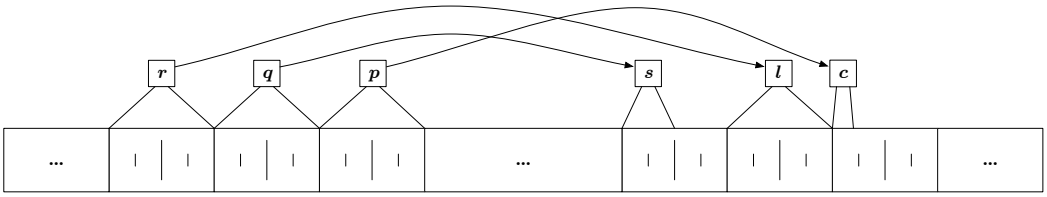


Figure 2.6: A schematic view of memory. The smallest division represents one 8-bit memory location. The height of the division line represents the byte-, short-, and long-address boundaries.

As indicated by Ex. 2.58, the amount of space reserved for any pointer is the same, no matter to what type the pointer is pointing. In the example, 32-bits are assumed for the addressable space; this is based on the architecture of the CPU. One way to make the pointer point to some variable is via the unary operator `&`, which gives the address of the label on its right-hand-side.

Example 2.59

```
p = &c;
q = &s;
r = &l;
```

Note that `&p` is the address of the pointer variable.

One way to access the contents of a variable using a pointer is via the unary operator `*`, which is called the *dereferencing* operator.

Example 2.60 At the end of these three statements, variable `c` is loaded with the value 10 via the dereference of its address in pointer `p`.

```
p = &c;
c = 0;
*p = 10;

/* now it is true that (c == 10) */
```

To declare a pointer, just add the * symbol to the left of the variable name.

Example 2.61

```
char *p;
short *q;
long *r;
```

This syntax is intended as a mnemonic. Using Ex. 2.61, the notation implies that the expression `*p` is a `char`, `*q` is a `short` and `*r` is a `long`. Note that as seen before, the space allocated to hold `p`, `q` and `r` is all the same (usually 32-bits on modern microprocessors), but what they point to is different. This matters to the compiler when pointer indexing is used.

You can create pointers to **any** type, for example:

- pointers to other pointers;
- pointers to functions;
- pointers to any type of array; and
- pointers to structure types.

The unary operators * and & have a very high precedence. However, the unary operators ++ and -- have the same level of operator precedence. When the compiler parses a line of source code, it resolves operators with the same precedence from right-to-left. Thus, the statement `*p++;` will have a very different result compared to `(*p)++;`. The former case will increment the address stored in `p` first, then dereference the result. The latter case will read the dereferenced address first and increment the resulting value without changing the address stored in `p`. Several examples of using pointer indexing are listed in Table 2.13, assuming that `char c = 5;`, `char *p;`, and `p = &c;`. Notice that all but the final statement are equivalent.

Table 2.13: Pointer Indexing Operations

	Before			After			
Instruction	&c = 100	101	p	&c = 100	101	p	*p
`c = *p + 1;`	5	0	100	6	0	100	6
`*p += 1;`	5	0	100	6	0	100	6
`++*p;`	5	0	100	6	0	100	6
`(*p)++;`	5	0	100	6	0	100	6
`*p++;`	5	0	100	5	0	101	0

Arrays are blocks of consecutive types. An array variable is similar to a pointer of that type that has been initialized to the address of the first entry of the block. That is, pointers are similar

to uninitialized array variables. An array is declared via the second statement presented in Ex. 2.62. Then, `a[i]` refers to the *i*th element of the array, beginning with 0.

Example 2.62 This example declares a pointer and an array of length 10. The array declaration causes the compiler to reserve a block of 10 consecutive 16-bit cells.

```
short *p;
short a[10];

p = &(a[2]);

/* The following expressions are true. */
*(p) == a[2];
*(p+1) == a[3];
```

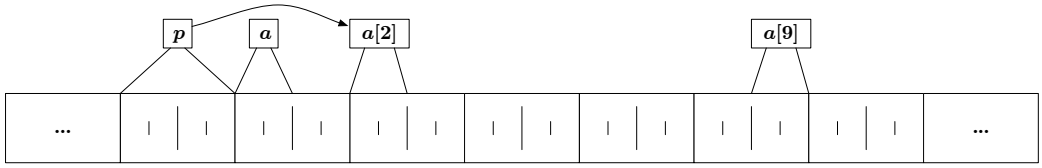


Figure 2.7: A schematic view of memory containing a pointer and an array.

Referring to Ex. 2.62, notice that adding 1 to `p` is equivalent to adding 1 to the array index. That is, assuming a `short` is 16 bits, the compiler knows `p` points to a 16-bit memory elements, so when altering the address which `p` points to, the compiler adjusts based on the type.

Example 2.63 This example declares a pointer to a `short` type and an array of 5 `long`s. The pointer is loaded with the address of the base of the array.

```
short *p;
long a[5];
char c;

p = (short *) (&(a[0]));
```

Many powerful and dangerous tricks can be performed using pointers. In fact, there are engineers in the industry who will purposely use pointers in cryptic and malicious ways. Often their goal is to create job security for themselves. My very first consulting money was earned by reverse engineering some overly cryptic code, using multiple levels of function pointers, written by

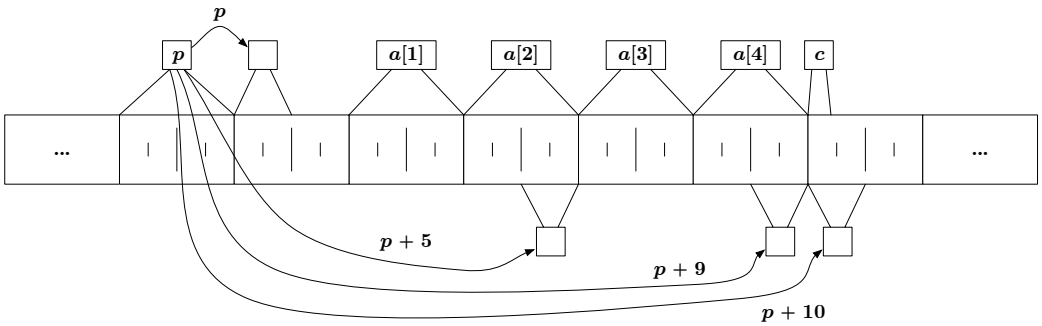


Figure 2.8: A schematic view of memory containing a pointer and an array. Here the pointer type is different from the array type.

an unstable "engineer" who was fired. Sadly, even if you try to write maintainable code, you will probably be forced to work on terrible code, which often involves poor use of pointers. For example, referring to Ex. 2.63, the compiler knows that `p` is supposed to point to types of 16-bit `shorts`, and it will do so. The code simply initializes the address to which `p` points. After that, there is nothing to prevent the code from adding any number to the address and dereferencing the result. Thus, it is very easy to overwrite memory locations by accident (or on purpose – `consts` may not be so constant after all).

WARNING: `p` can point to any address. If you do not initialize `p`, it still points to something.

As you may have guessed, pointers are the most common source of difficult problems in code. Often, this has to do with the pointer notation which can be intimidating. As a nice alternative, we can use array notation on pointers as in the next example.

Example 2.64

```
/* these are equivalent ways of accessing the fifth element away from p
   */
*(p+5) == p[5];
```

It is always better to opt away from pointer notation. Pointers are powerful but generally a little cryptic. You should always write code as if you are working on a team, and assume someone else will have to understand your code. Array notation is more likely to be better understood by all members of the team.

You might be wondering why we need pointers, especially if they seem to cause so much trouble. Three good reasons for using pointers include:

1. Passing information in and out of function calls.
2. Dynamic memory allocation.
3. Especially in embedded devices, we can use pointers to access memory-mapped registers in order to manage various peripheral devices. We will be using pointers for this extensively later on.

2.8.1 PASSING BY REFERENCE

Consider the following code which does not function as intended.

Example 2.65

```
void main (void)
{
   short a = 10;
   short b = 13;

   swap (a,b);

   /* a == ?, b == ? */
}

void swap (short x, short y)
{
   short temp;

   temp = x;
   x = y;
   y = temp;
}
```

The code in Ex. 2.65 will not do what we want because the values loaded into `a` and `b` are passed into the function; that is, the values are copied onto the stack or into a register so the function can access them. Fig. 2.9 shows how the stack pointer (SP) is used to temporarily store the return address so the program counter knows where to return to, and then the local variables `x` (which holds a copy of the `a` value), `y` (which holds a copy of the `b` value) and `temp`. The values below the "memory" represent how the values in the local memory change after each instruction of the function is executed. At the very end of the function, the stack pointer "pops" all the memory by moving back from where it started, so the changed memory is lost forever. Notice the original values are still safely stored in `a` and `b`.

To correct the problem from Ex. 2.65, consider using pointers.

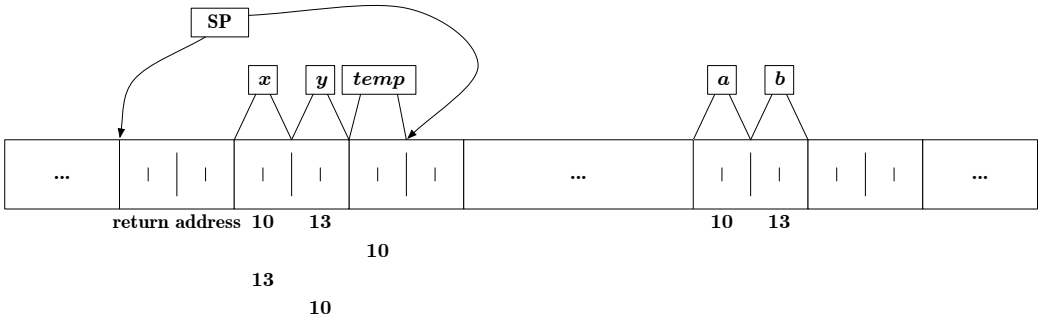


Figure 2.9: The memory contents while the code in Ex. 2.65 is executed.

Example 2.66

```
void main (void)
{
   short a = 10;
   short b = 13;

   /* Now we pass the address of the variables we want to change. */
   swap (&a,&b);

   /* a == ?, b == ? */
}

void swap (short *x, short *y)
{
   short temp;

   temp = *x;
   *x = *y;
   *y = temp;
}
```

Now, the code in Ex. 2.66 does what we intended because the addresses of `a` and `b` are passed into the function. The function accesses their values by indirect reference via the pointers `x` and `y`. The memory contents change as indicated in Fig. 2.10.

Another reason to use pointers is for passing large pieces of memory into a function. For example, suppose we wanted to pass an array of 10,000 `long`s into a function. If you tried passing by value, the stack would need to hold all 80,000 bytes. However, if we used a pointer to the base of the array, the stack only needs to hold the 4-byte base address.

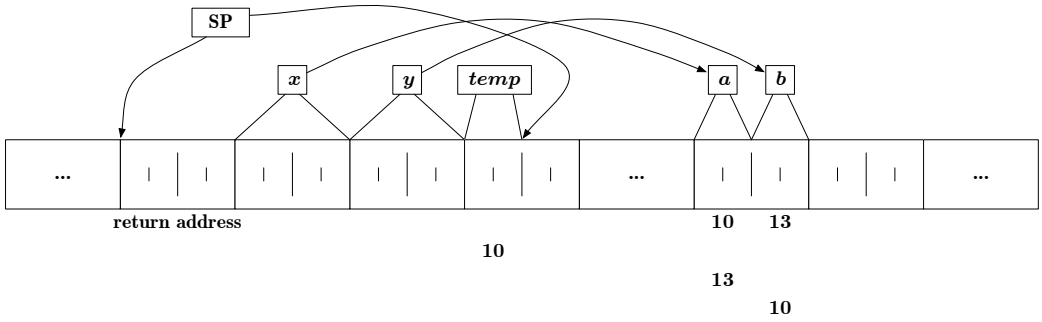


Figure 2.10: The memory contents while the code in Ex. 2.66 is executed.

2.8.2 DYNAMIC MEMORY ALLOCATION

You are able to dynamically allocate memory at run-time and have the base be referred via a pointer. What this means is that the compiler doesn't reserve the consecutive bytes of memory as for an array declaration. Instead, the CPU is directed to find a block of consecutive bytes in memory that are not being used and return the base address.

WARNING: it is easy to lose memory if a function allocates memory but never frees it. This is a *memory leak*, and, eventually, repeated calls to the function will consume all of the available memory, causing the program to crash.

WARNING: run-time allocation is not a great idea for embedded programs. Memory leaks in PC applications are difficult enough to track down using all of the powerful debugging tools available on a host system. Many embedded system tools are extremely limited, and so debugging an embedded memory issue tends to be exceptionally difficult.

Example 2.67

```
#define NUMBER_OF_SHORTS_TO_ALLOCATE 10

short *p;

p = (short *) malloc (sizeof (short) * NUMBER_OF_SHORTS_TO_ALLOCATE);
if (p == NULL)
{
   /* error -- need to tell the user and stop execution */
}

/* skipping code */

free (p);
```

The C operator macro `sizeof()` returns the number of bytes used by the given argument. The function `malloc()` returns the base to a block of requested bytes. Referring to the code in Ex. 2.67, the processor must locate 10 consecutive unused `short` cells and return the base address. If they cannot be found, the special "invalid address" `NULL` is returned. Once the code is finished using the allocated memory, it needs to be returned to the state of unused memory via a call to `free()`.

Example 2.68 An alternative form for dynamic memory allocation.

```
#define NUMBER_OF_SHORTS_TO_ALLOCATE 10

short *p;

p = (short *) calloc (NUMBER_OF_SHORTS_TO_ALLOCATE, sizeof (short));
if (p == NULL)
{
   /* error -- need to tell the user and stop execution */
}

/* skipping code */

free (p);
```

Example 2.69 Code can adjust the size of the allocated memory with the following.

```
#define NUMBER_OF_SHORTS_TO_ALLOCATE 10
#define NUMBER_OF_SHORTS_TO_REALLOCATE 12

short *p;

p = (short *) calloc (NUMBER_OF_SHORTS_TO_ALLOCATE, sizeof (short));
if (p == NULL)
{
   /* error -- need to tell the user and stop execution */
}

/* skipping code */

p = (short *) realloc (p, sizeof (short) *
    NUMBER_OF_SHORTS_TO_REALLOCATE);
if (p == NULL)
{
   /* error -- need to tell the user and stop execution */
}

free (p);
```

2.9 MULTI-DIMENSIONAL ARRAYS

Consider the code in Ex. 2.70, in which multi-dimensional arrays are declared. Note that the `const` qualifier is not required for the array definition. However, many times large arrays are used for various lookup tables and so are meant to be fixed conceptually.

Example 2.70

```
#define MAX_ROWS 2
#define MAX_COLS 5

const short m_table[MAX_ROWS][MAX_COLS] =
{
   {1, 2, 3, 4, 5},
   {6, 7, 8, 9, 10}
};

/* Then the following are true. */
m_table[0][1] == 2;
m_table[1][4] == 10;
```

Remember the first entry in every dimension is 0. This may be annoying to people already familiar with other high-level languages. For example, MATLAB begins all vectors at index 1.

Especially for embedded applications, it is of interest to understand how the compiler organizes individual elements in a multi-dimensional array. Consider the table presented in Ex. 2.71. Note that the rightmost array index varies fastest as elements are accessed in storage order.

Example 2.71 Here index `MAX_DIM2` will vary most frequently.

```
#define MAX_DIM0 3
#define MAX_DIM1 2
#define MAX_DIM2 5

const short m_table[MAX_DIM0][MAX_DIM1][MAX_DIM2] =
{
   {
      {1, 2, 3, 4, 5},
      {6, 7, 8, 9, 10}
   },
   {
      {11, 12, 13, 14, 15},
      {16, 17, 18, 19, 20}
   },
   {
      {21, 22, 23, 24, 25},
      {26, 27, 28, 29, 30}
```

```
    },
};

/* Then the following are true. */
m_table[0][1][2] == 8;
m_table[2][1][3] == 29;
```

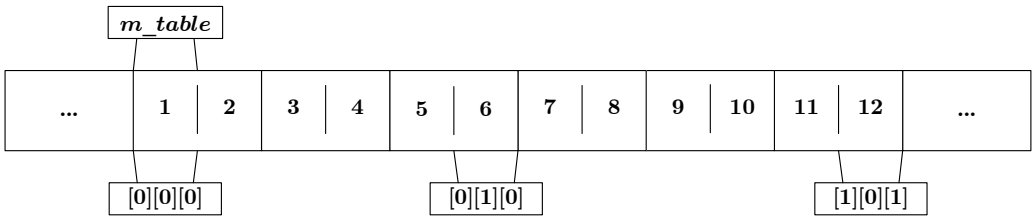


Figure 2.11: The memory contents for the multi-dimensional array of the code listing.

Regarding pointers, [] have higher precedence than unary *.

Example 2.72 The following chart lists some of the different declarations possible using array and pointer notations.

Declaration	Primary Number	Each Primary Element Type	Total Allocation at Compile Time
`short a[10][20];`	10	Array of 20 `shorts`	$10 \times 20 = 200$ `shorts`.
`short *b[10];`	10	Pointer to 1 `short`	10 pointers to `shorts`. All 10 pointers are uninitialized.
`short (*c)[20];`	1	Pointer to array of 20 `shorts`	1 uninitialized pointer.

Now suppose we had the following code.

```
typedef short (*PointerToArrayType) [20];

short i;

for (i = 0; i < 10; i++)
{
   b[i] = (short *) malloc (sizeof (short) * 20);
}
c = (PointerToArrayType) malloc (sizeof (short) * 20 * 10);
```

In a sense, a, b and c are all equivalent, as in the following code.

```
short i;
short j;

for (i = 0; i < 10; i++)
{
   for (j = 0; j < 20; j++)
   {
      a[i][j] = SOME_MAGIC_NUMBER;
      (b[i])[j] = SOME_MAGIC_NUMBER;
      c[i][j] = SOME_MAGIC_NUMBER;
   }
}
```

So, in all cases, we are able to read/write 200 short values. Also, we are able to use array notation in all cases. What's the difference?

	a	b	c
Compile-Time Memory	200 consecutive short	10 consecutive short *	1 short (*)[20]
Run-Time Memory		10 non-consecutive blocks of 20 consecutive short	200 consecutive short
Comments	Always available; no memory management; fixed array size.	Most memory usage; fragmented memory; variable array size.	Pointer required; memory management; fixed array size.
Bytes Required	200 × sizeof(short)	(10 × sizeof(short *))+ (10 × 20 × sizeof(short))	(1 × sizeof(short *))+ (200 × sizeof(short))

Based on the differences, c is nearly the same as a with the exception that it requires a pointer and dynamic memory allocation. As a result, typically either method a or b is used.

2.10 FUNCTION POINTERS

While a function is not a variable, it is a label and still has an address. As a result, it is possible to define *function pointer*s, which can be assigned and treated as any other pointer variable. For example, they can be passed into other functions, in particular, callbacks into Real-Time Operating Systems (RTOSes) or hooks in an Interrupt Service Routine (ISR) vector table. We will see more information about ISRs on the Arduino in Ch. 9. It turns out that the Wiring Arduino software library uses an array of function pointers in order to allow casual users the ability to dynamically set arbitrary functions as ISRs. This material is beyond the scope of this book.

Notation to declare a function pointer is

```
return_type (* variableName)(argument_list);
```

Example 2.73

```
int (* comp) (void *, void *);
char * (* weird) (void);
```

Variable `comp` is a pointer to a function that returns an `int` and takes two generic pointers as arguments. Variable `weird` is a pointer to a function that returns a `char` pointer and takes no arguments.

Note that the best way to use function pointers is by defining a new type as in Ex. 2.74 that shows a method for implementing a state machine in software.

Example 2.74

```
typedef void (* PointerToStateFunction) (int);

typedef enum
{
   STATE_ONE,
   STATE_TWO
} States;

typedef struct
{
   States theState;
   PointerToStateFunction theStateFunction;
} StateEntry;

void StateOne (int Message);
void StateTwo (int Message);

static States m_state;
const static StateEntry m_stateTable[] =
{
   {STATE_ONE, StateOne},
   {STATE_TWO, StateTwo},
};

m_state = STATE_ONE;

void ProcessMessages (void)
{
   int message;
```

```
    /* wait until some asynchronous event happens and set message */

    m_stateTable[m_state].theStateFunction(message);
}
```

Notice at the very end of the code listing in Ex. 2.74, the variable `m_stateTable` (which is a structure type, see Sec. 2.11) is used to call a function (either `StateOne()` or `StateTwo()`).

Function pointers are very tricky, but often used in embedded programming for various reasons including:

- callbacks into RTOSes;
- ISR handling;
- I/O port interfacing to higher level;
- as shown in Ex. 2.74 to reduce code size at expense of a complexity.

Be aware of operator precedence as in the next example.

Example 2.75

```
/* function returning pointer to int */
int *f (void);

/* pointer to function returning int */
int (*f) (void);
```

2.11 STRUCTURES

A *structure* is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. Similar constructs are available in most high-level languages such as records in Pascal and classes in C++.

Notation for defining a structure type is as presented in Ex. 2.76.

Example 2.76

```
struct TagPoint
{
    int x;
    int y;
};
```

Referring to Ex. 2.76, `struct` is a keyword introducing the structure declaration, which is the list in braces `{ }`. `TagPoint` is an **optional** structure tag. It names this kind of structure and may be used as shorthand for the part of the declaration in braces. The labels `x` and `y` are called *members* of the structure. They are "within the scope" of this structure.

To see how we can declare structure variables, consider the code listing in Ex. 2.77.

Example 2.77 The following

```
struct TagPoint x, y, z;
```

is analogous to

```
int x, y, z;
```

Both cases declare `x`, `y` and `z` as variables of the same type.

Note that a structure declaration followed by no variables does not allocate any space; it just defines the template for later use.

Example 2.78

```
struct TagPoint point;
struct TagPoint maxPoint = {320, 200};
```

The code presented in Ex. 2.78 will use and allocate the space for the structure definition previously stated. The second variable `maxPoint` will declare a variable and initialize all members.

The `.` operator connects the structure variable name and the member name. So, for the proceeding example, the following are true.

Example 2.79

```
maxPoint.x == 320;
maxPoint.y == 200;
```

Now consider building structures using other types such as other structures.

Example 2.80

```
struct TagRectangle
{
   struct TagPoint upperLeftCorner;
   struct TagPoint lowerRightCorner;
};

struct TagRectangle polygon;

polygon.upperLeftCorner.x = 320;
```

Legal operations on structures include assignment, copy, passing type into functions, returning type from functions, get address via &, and access members. Notice the list does not include operations like comparison, increment, decrement, etc.

Example 2.81

```
struct TagPoint add (struct TagPoint p1, struct TagPoint p2)
{
   struct TagPoint temp;

   temp = p1;

   temp.x += p2.x;
   temp.y += p2.y;

   return (temp);
}
```

The code in Ex. 2.81 works fine, but consider all of the memory getting copied to/from the stack; especially in cases when the structure has several members. A more efficient alternative is to pass the address of the structure variables, and then use pointers within the function. Pointers to structures are just like any other pointer variables. However, pointers to structures are so common that a special operator -> is used to dereference the member of a structure via a pointer.

Example 2.82

```
struct TagPoint *pp;
struct TagPoint p3;

pp = &p3;

/* The following are equivalent. */
(*pp).x == pp->x;
```

Be careful of operator precedence.

Example 2.83

```
(*pp).x == pp->x;
(*pp).x != *(pp.x);
*pp.x == *(pp.x);
```

That is, since . has higher precedence than *, we would need (). The better choice is to use ->. Note that both . and -> associate from **left-to-right**.

Example 2.84

```
struct TagRectangle r;
struct TagRectangle *rp;

rp = &r;
r.upperLeftCorner.x = 5;

/* The following are all equivalent. */
rp->upperLeftCorner.x == 5;
(r.upperLeftCorner).x == 5;
(rp->upperLeftCorner).x == 5;
```

Arrays of structures are just as before using the type of structure. Note that you shouldn't assume the size of a structure is the sum of the sizes of members. Address boundary alignment requirements will force "holes" in a structure. Consider the following example.

Example 2.85 Compare the memory of the following two variables.

```
struct
{
   char c;
   long l;
} x;

struct
{
   long l;
   char c;
} y;
```

Here x requires 8 bytes, while y only uses 5 bytes, as indicated in Fig. 2.12.

Referring to Ex. 2.85, the memory holes occur as a result of the byte-boundaries necessary in computer architecture addressing where masking-off the least significant address bits is used to address larger units. The `sizeof()` operator returns the proper value, so `sizeof(x)== 8` and `sizeof(y)== 5`. Note that these amounts are assuming the compiler does not perform any kind of optimization. However, even with optimizations, there are still situations where structure memory will contain unused bytes.

Recursion using pointers is allowed, as indicated in the code listing of Ex. 2.86.

Example 2.86

```
struct TagOfNode
{
   char *word;
```

```
   int count;
   struct TagOfNode *left;
   struct TagOfNode *right;
};
```

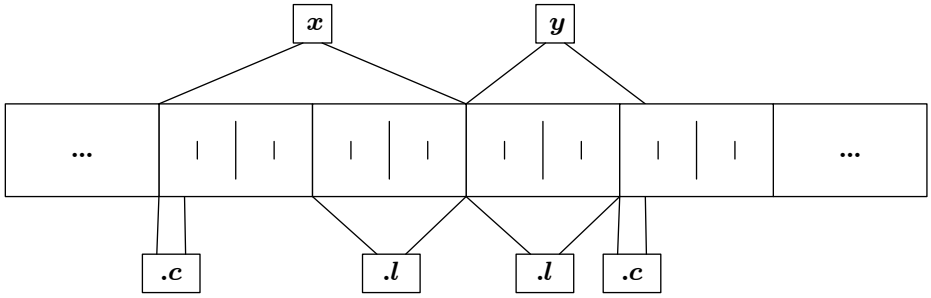


Figure 2.12: The memory contents for the two structure variables of the code listing.

The last two members of the proceeding code listing are allowed as pointers, but `struct TagOfNode x` would not be allowed (yet another important reason for pointers). To understand why, imagine that the compiler needs to reserve space for each member. Because of the recursion, there is no way for the compiler to save the correct amount of structure space for a structure that is still being parsed. However, because all pointers are the same size regardless of the type to which they point, the compiler can reserve the proper amount of space without knowing how big the structure is going to be.

2.11.1 TYPEDEF

The keyword `typedef` creates a new data type name. It is really useful regarding `struct` types, as indicated in the following example.

Example 2.87

```
typedef struct TagPoint
{
   int x;
   int y;
} Point;
```

```
Point p1;
Point p2;

p1.x = 1;
```

Note that C++ automatically does this for `enum`s and `struct`s, so `typedef` is less useful than it used to be.

2.12 UNIONS

A `union` is a variable that may hold, at different times, objects of different types and sizes. The compiler keeps track of size and alignment requirements. The following shows an example union variable declaration containing three members.

Example 2.88 Here u is a union variable that only uses 4 bytes, assuming the typical byte sizes.

```
union optionalTag
{
   short s;
   char c;
   long l;
} u;
```

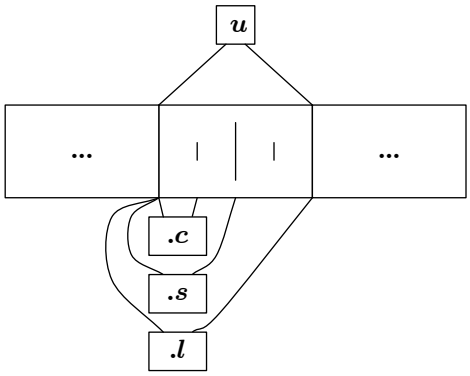


Figure 2.13: The memory contents for the union variable of the code listing.

Variables declared as `union`s are treated similar to structures. Use `.` and `->` to access members.

2.13 BIT-FIELDS

Especially in embedded applications, memory resources are valuable and often scarce. Yet product requirements can often call for extensive features. As a result, we often need to pay attention to how information is being stored in memory. One interesting aspect of C that does not receive a lot of attention is the ability to pack bits when storage is at a premium. One method for packing bits is to use bit masks such as in the following example.

Example 2.89

```
#define BIT_MUTE_AUDIO  0x01
#define BIT_BACKLIGHT   0x02

unsigned int flags;

flags = (BIT_MUTE_AUDIO | BIT_BACKLIGHT);
```

An alternative method uses bit-fields, as in the following example.

Example 2.90

```
struct
{
   unsigned int mute : 1;
   unsigned int backlight : 1;
   unsigned int unused : 14;
} flags;

flags.mute = 1;
flags.backlight = 1;
```

Bit-fields may be declared only as `int`s, and the individual fields are not directly addressable. **WARNING:** while ANSI C specifies the syntax for bit-fields, their implementation is **totally** machine dependent. Example issues include word boundary alignment and bit ordering. As a result, bit-field usage may cause trouble when porting software from one target platform to another. For example, consider Ex. 2.91.

Example 2.91 The following listing is a nice way to use unions and bit-fields, especially for saving parameter "files" into non-volatile memory such as an EEPROM. But, it is not up to ANSI C to define the bit ordering. So, either `x.word == 0x8000` or `x.word == 0x0001` might be true – it depends on the compiler for the target platform.

```
typedef union
{
   struct
   {
      unsigned int mute : 1;
      unsigned int backlight : 1;
      unsigned int unused : 14;
   } bits;
   unsigned int word;
} Type;

Type x;

x.word = 0x0000;
x.bits.mute = 1;
```

2.14 VARIABLE-LENGTH ARGUMENT LISTS

The declaration ... means the number and types of arguments may vary. It appears only at the end of a function's argument list. Then, the type `va_list` and the macros `va_start`, `va_arg` and `va_end` are used on `va_list` to access the arguments.

Example 2.92

```
#include <stdarg.h>

void someFunction (int numArgs, ...)
{
   int temp;
   va_list ap;
   int i;

   va_start (ap, numArgs);

   for (i = 0; i < numArgs; i++)
   {
      /* assuming all types passed in are int */
      temp = va_arg (ap, int);
   }

   va_end (ap);
}
```

This is a very rare aspect of C and is only included for completeness.

SUMMARY

This chapter has been a brief, yet complete presentation of the ANSI C high-level programming language. It truly is one of the most valuable tools that an electrical or computer engineer can have in their toolset going into the industry. It is a foundational language at the core of many recent derivative languages including C++, Microsoft's C# and Apple's Objective C. More relevant is the pervasiveness of C in so many embedded systems ranging from simple 8-bit microcontrollers, such as the ATMEL ATmega328P in an Arduino development board, to highly complex 32-bit microprocessors like the Texas Instruments OMAP dual core running embedded Linux Real Time Operating Systems, which are written in C. C has been relevant for over 40 years and shows no real sign of disappearing.

CHAPTER 3

Introduction to Arduino

3.1 BACKGROUND

The next topic we need to cover is our tool suite. In particular, we need a means for writing programs that run on an embedded processor. This brings us to an *Arduino*, which refers to a circuit board containing an ATMEL ATmega microcontroller preloaded with a *boot loader* program. A boot loader is a tiny program (e.g., 2 Kbytes on the Arduino) stored in non-volatile memory at a location such that when power is applied or the reset button is pressed the CPU jumps into it and executes its instructions. The system is designed such that when the board is reset, the boot loader configures the microcontroller serial port to send and receive information to and from your computer, often referred to as the host. Then, when you create an *application*, an embedded program you create to do something meaningful, you can send it to the target processor by uploading it. When you start the upload process, a special signal on the Arduino hardware causes the microcontroller to reset which forces the boot loader to run. At that time, the boot loader is written such that it doesn't assume anything about the current state of the processor. So, it performs several steps to configure the controller to listen for incoming data on its serial communications port. Because the host computer initiated an upload of the application program which caused the reset, there is data being sent via the serial port. So the boot loader sees this data and responds by saving it to the non-volatile memory at a special location. When finished, the boot loader jumps to that application location in the flash and starts executing program instructions. In the event that the Arduino is manually reset or when power is first applied, the boot loader will not see any data on the serial port, and it will just jump to the application location in flash. In either case, the boot loader jumps into your program which does whatever it is you meant it to do.

A significant concept behind the Arduino embedded development platform is its intent of being as open as possible. This means the hardware schematics of the circuit board (see Apx. A) are freely available in the public domain such that end users and, in particular, engineering students can create their own design beginning with an inexpensive functional platform. Additionally, the Arduino Integrated Development Environment (*IDE*), a program that runs on your computer in order to write and compile source-code into ATmega executable programs, is written in Java. The benefit of a Java-based IDE is that any computer with a Java interpreter is able to run the Arduino development software. In particular, students with Windows, MacOSX or even Linux will be able to write programs for their embedded hardware.

The final component of Arduino is a library of functions referred to as *Wiring*. Any program developed in the Arduino IDE is called a *sketch*. The concepts of Wiring and sketches are to benefit the

end user by concealing all of the low-level details required to control the embedded microcontroller. While this is a meant to be a nice feature of using an Arduino, it is considered a negative in this book. The primary purpose of this text is to learn how to manipulate generic microcontrollers. We happen to be using the ATMEL ATmega328P microcontroller, which is the controller on an Arduino development board. It is not useful to memorize the various Wiring functions, as they only pertain to the ATmega processor as it sits on the Arduino circuit board. The significance of this is that if you write a program using the Wiring library functions to do something interesting on the Arduino hardware, it will **not** function on any other hardware platform. While it might be satisfying to write software that causes the Arduino to do something, if it is not portable to another platform, then you have really limited your own usefulness in the engineering industry.

To correct this, we will avoid Wiring functions at all costs. Instead, we are going to use the Arduino hardware and the IDE in order to learn how to develop ANSI C programs that will configure and control our embedded microcontroller. The methods introduced in subsequent chapters are meant to be universal and generic whenever possible. We will use a few Wiring functions early on out of necessity. In particular, the `delay()` and `Serial` functions will be used in many applications until much later when timers, interrupts and serial communication topics are all studied. Beyond this, the ANSI C methods presented are 100% portable, and the various topics covered are as generic as possible. Once learned, the end user only needs to refer to a microcontroller reference manual to determine specific control over their target processor.

3.2 EXPERIMENTS USING THE ARDUINO DUEMILANOVE DEVELOPMENT BOARD

All of the material presented in this book will involve analyzing and developing embedded source code that is run on the ATMEL ATmega328P microcontroller that sits on the Arduino Duemilanove (i.e., "2009" in Italian) development board as depicted in the modified photograph of Fig. 3.1. Many times, the C and assembly source code will be provided to allow readers to experiment with embedded programming and applications.

The Arduino Duemilanove development board has an ATMEL ATmega328P 8-bit processor operating at 16 MHz, with 2 Kbytes of SRAM, 32 Kbytes of Flash memory (2 Kbytes are already used by the built-in boot loader, so only 30 Kbytes are available for your applications), 1 Kbyte of Electrically-Erasable/Programmable Read-Only Memory (*EEPROM*), three user accessible LEDs, and 20 input/output (I/O) pins; six of which provide Pulse-Width Modulation (PWM) analog output and six other provide Analog-to-Digital Converted (ADC) input. Many of these features are depicted on the modified photograph in Fig. 3.1.

The development board also provides a USB interface that allows for communication between the ATmega328P and the Java-based IDE. So, for the lab work, we will always communicate with the ATmega328P via the USB connector, at which time the development board also receives its 5 V regulated power. However, note that there is also a 2.1 mm center-positive barrel power interface with associated voltage regulator. Once a program is stored in the flash memory of the microcontroller, a

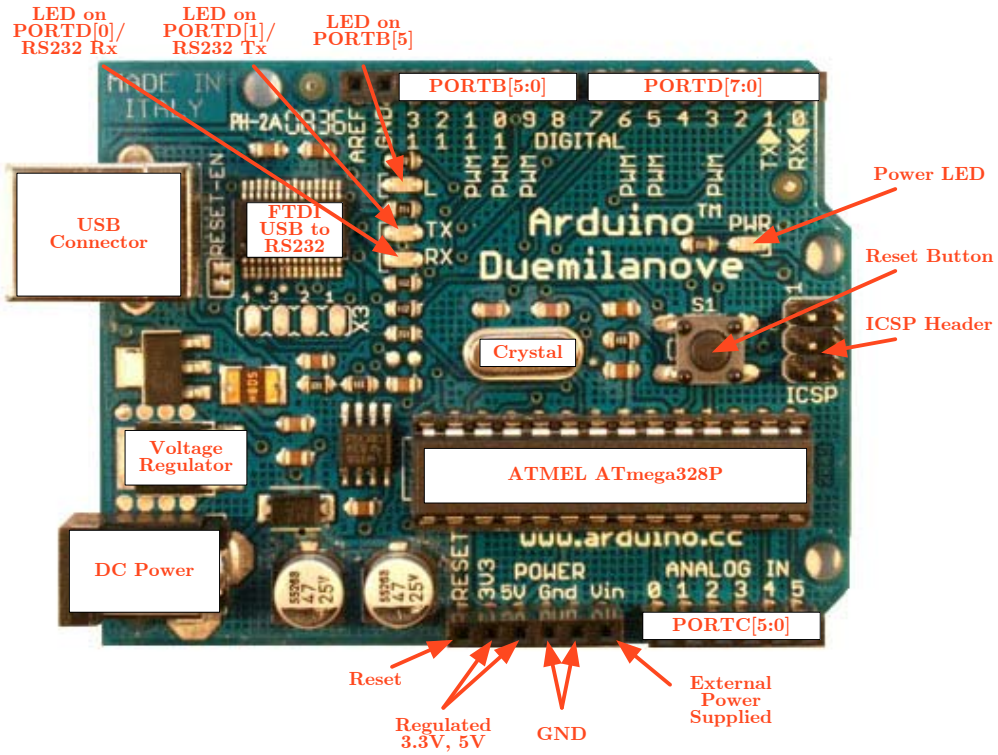


Figure 3.1: Photograph of the Arduino Duemilanove development board, including key component identifications. Adapted from Arduino (2010).

separate DC power supply can be used to supply power to the board, at which time the built-in boot loader will run, ultimately jumping into whatever program is present. The Arduino Duemilanove circuit board is designed to automatically switch to the correct power supply. If desired, power may also be supplied to the GND and V_{in} pins on the **POWER** header. The development board can safely operate on an external supply of 7 V to 12 V.

3.3 ARDUINO TOOLS TUTORIAL

The purpose of this section is to ensure your embedded development environment is installed and ready to go.

1. Download the latest Arduino IDE from `http://arduino.cc/en/Main/Software`. This is the Java program used to create, write, compile and upload the embedded software.

2. Download the latest Serial-to-USB drivers from http://www.ftdichip.com/Drivers/VCP.htm. These drivers allow for RS-232 serial communications via your computer's USB port. Once installed, they create a virtual serial port, so the IDE is able to connect to the Arduino hardware via a physical USB cable. The driver converts USB communications to and from a serial protocol, so the IDE sends and received RS-232 data as though you had a serial port on your computer (i.e., many modern computer manufacturers no longer provide serial ports). The Arduino circuit board includes an FTDI chip that performs a similar conversion between USB communications and RS-232 serial communications for the ATmega328P microcontroller.

3. Then:

   - MacOSX: Install the USB drivers. Then connect the development board via the USB cable.
   - Windows: Connect the development board via the USB cable. When prompted, install the USB drivers.

4. The green power LED should light up. Additionally, if you have a new board, the yellow LED (connected to PortB[5]) should blink due to the pre-loaded application stored in the flash.

5. Then:

   - MacOSX: Copy the Arduino IDE application to the Applications directory. Launch the Arduino IDE.
   - Windows: Open the Arduino folder and launch the Arduino IDE.

6. You should see a windows similar to that depicted in Fig. 3.2.

7. Open the example "Blink" program by selecting `File→Examples→Digital→Blink`.

8. You should have the following source code listing in an editor pane. Note that text in between `/* */` is considered a comment for humans to document some important information regarding the program; that is, it is not actually part of the code used by the processor, and so it is omitted here. Also, all text on the same line following `//` is considered a comment and so is omitted here.

```
int ledPin = 13;

void setup()
{
   pinMode(ledPin, OUTPUT);
}

void loop()
{
   digitalWrite(ledPin, HIGH);
```

```
    delay(1000);
    digitalWrite(ledPin, LOW);
    delay(1000);
}
```

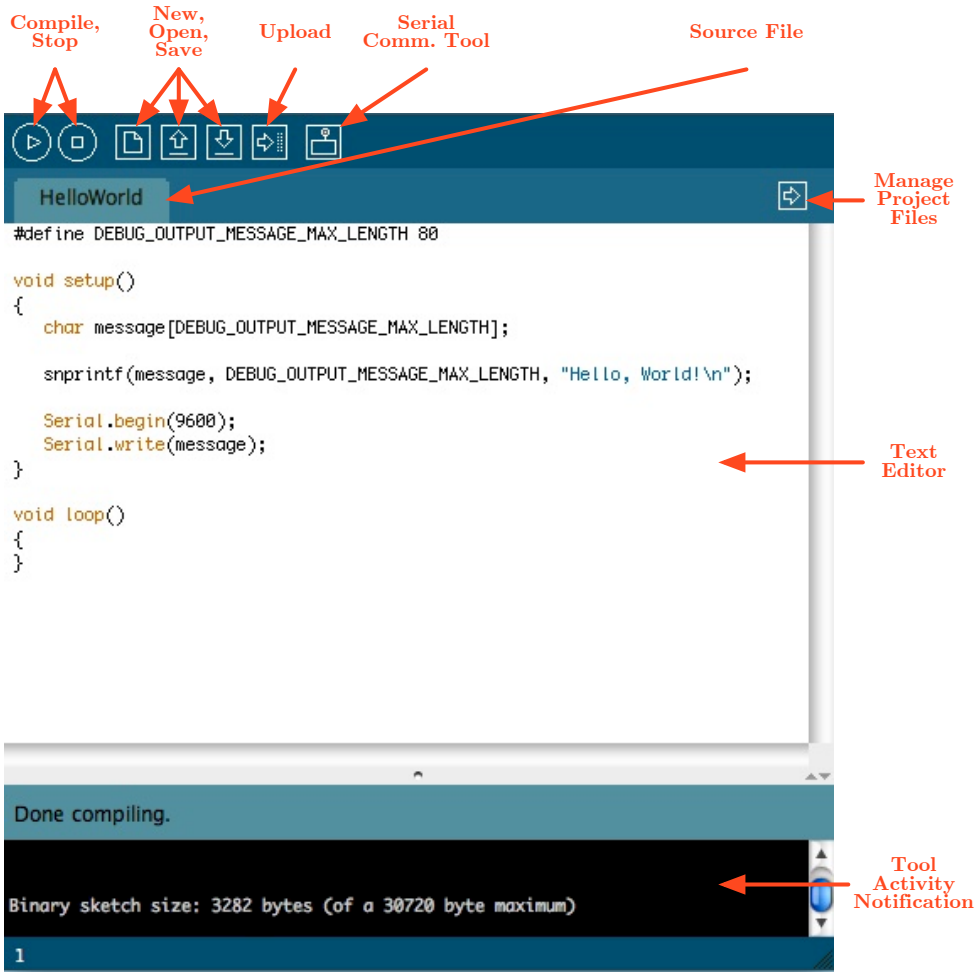


Figure 3.2: Screen shot of the Arduino IDE. All of the interesting regions and icons are explicitly identified.

Regarding this simple example, all programs built in the Arduino IDE include the two functions `setup()` and `loop()`. As the labels are meant to imply, the `setup()` function is run once

when the program begins, so various initialization would take place here, while the `loop()` function executes over and over until the processor is reset. Imagine a race-car on an oval race-track; the car just keeps driving around the same loop until the race is finished. This example includes a named piece of memory called a global variable, with the label `ledPin`. When the program is compiled on the host computer, this particular location in the target memory is initialzied with the value 13 because the Arduino circuit board's 13th I/O pin has the yellow LED attached to it; see Apx. A. When `setup()` is run at the start of the program, a single library call is made to make the 13th I/O pin, referenced via that named variable, as an output pin, so that the processor can drive the line running to the LED either high or low, hence turning the LED on or off. Pins could be made to be input, so that a sensor or button can be read in, instead. After `setup()` is finished, `loop()` is called over and over. Within the `loop()` function, the LED is turned on, and then a fixed delay of 1000 milliseconds is executed before the LED is turned off, and another delay.

Notice that none of the functions presented in this example are either assembly nor are they ANSI C; instead, they are all part of the provided Wiring library of functions.

9. With the "Blink" program in the editor, press the "play" icon in order to compile the program (the tool refers to this as "Verify").

10. To upload the program to the embedded processor, first make sure you have the correct development board by checking under `Tools→Board`. It is most likely set to the correct value of Arduino Duemilanove or Nano w/ ATmega328. If you have a different board, you should adjust accordingly.

11. Next, connect to the proper serial port under `Tools→Serial Port`

    - MacOSX: /dev/tty.usbserial- (something)
    - Windows: USB Serial (COMx); check device manager

    In either case, the driver you installed earlier is allowing the development environment to communicate to the embedded processor with RS-232 serial communications via the USB cable.

12. Press the penultimate icon, the tool refers to this as "Upload". You should see the Rx and Tx yellow LEDs flash as the program is sent to the target flash memory. When it finishes, you should see the LED blink at one second rate.

13. Now that you have compiled and uploaded your first program, let's convert it from the easy-to-use-yet-non-standard library calls to easy-once-you-know-them-and-portable ANSI C instructions. Change the "Blink" program to the following.

```
void MyDelay(unsigned long mSecondsApx);
```

```
void setup()
{
   unsigned char *portDDRB;

   portDDRB = (unsigned char *) 0x24;

   *portDDRB |= 0x20;
}

void loop()
{
   unsigned char *portB;

   portB = (unsigned char *) 0x25;

   *portB |= 0x20;
   MyDelay(1000);
   *portB &= 0xDF;
   MyDelay(1000);
}

void MyDelay(unsigned long mSecondsApx)
{
   volatile unsigned long i;
   unsigned long endTime = 1000 * mSecondsApx;

   for (i = 0; i < endTime; i++);
}
```

This code is provided without much explanation, as you will learn what this code is doing later in the book. To summarize, we looked at the reference manual for the ATmega328P ATMEL (2009) component to find the physical address of the data direction register (DDR) and port register for port B (see Apx. B), that connects to pin 13 on the circuit board. We then accessed the registers directly by using a C pointer that allows such access. We then set the desired bit in the register using bit-wise OR (the vertical bar), and clear the same bit using bit-wise AND (the ampersand). Finally, we created our own approximate delay function; however, it is not accurate without using straight assembly. The most important point here is that we used standard ANSI C in conjunction with the microprocessor's data sheet to complete the task.

PROBLEMS

3.1 Modify the provided example Blink program to cause the LED to blink in a more interesting pattern. Turn in the source code listing of your modified program. Demonstrate your program to the instructor.

3.2 State the total number of bytes required by your program in (3.1). Hint: this number is easily obtained after you compile your program in the bottom window of the editor.

3.3 Modify the ANSI C version of the Blink program to cause the LED to blink in a more interesting pattern. Turn in the source code listing of your modified program.

3.4 State the total number of bytes required by your program in (3.3). Hint: this number is easily obtained after you compile your program in the bottom window of the editor.

CHAPTER 4

Embedded Debugging

4.1 INTRODUCTION

In many ways, software development is universal among all levels of programs. From high-level host applications to low-level embedded routines. In particular, by using a high-level programming language, engineers gain a great deal of portability that may lead to code re-use, and they gain a high-degree of maintainability through the proper use of self-documenting code constructs. All software development platforms, i.e., the CAD tools, include a compiler for the high-level language, an assembler for the architecture-specific language, and a linker to map software to physical or virtual addresses within the target environment. Additionally, most software development platforms include a "design entry" tool that allows the engineer the ability to create the source code text files. This tool turns out to be de-coupled from the CAD suite, as often engineers will discover a particular editor program they like and will tend to use it separate from the actual compilation tool-chain.

Another very important aspect to the software development process is the ability for an engineer to test the correctness of their software design. It turns out there are several stages within the development process in which problems may occur with the design. The two broadest categories for software defects include *compile-time errors*, in which the compiler is unable to successfully parse the input text file, and *run-time errors*, in which an executable was created but the desired behavior is incorrect. The term *bug*, referring to an incident in the 1940's in which a moth was caught in a computer circuit resulting in a malfunction, is used to describe any unwanted behavior in a software design. As such, the process of attempting to resolve problems in software is called *debugging*. Some software CAD tools include a debugger which is another program in the suite that allows an engineer the ability to:

- halt program execution when a specific line of code is reached (i.e., *breakpoints*);
- once halted, look at the current contents of memory (i.e., *memory watches*);
- once halted, look at the current contents of CPU registers;
- receive notification of critical exceptions with explanations of the offending instruction;
- execute one instruction (both high-level and assembly) and then halting again (i.e., *stepping*);
- resume normal execution;

among many other useful abilities. For anyone who has developed PC application software, it seems like the debugger is always present in the CAD tool-chain. For example, any of the Microsoft Visual

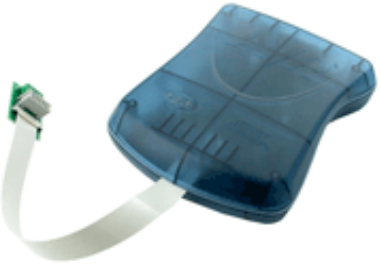
Studio development environments tie their debugger into their editor, so the software developer can make changes to their code, set breakpoints on certain lines, compile and run. The massive benefit occurs when a breakpoint line is reached, execution of the program is halted and the debugger shows the current line in the editor software. At this point, the developer can look at variable values, among other things, and then continue execution in many ways.

The debug tool is extremely useful in the process of software development. The observant reader would have noticed that the debugger is available to the PC application developer because the source code is compiled, assembled and linked for the same processor architecture as that of the CAD tools themselves. This means that the source code is able to include special debug marker information, so the debug tool can follow along with how the application is running. Additionally, the debug tool is able to see the state of registers and memory because the software in question is running on the same processor as the debugger. Unfortunately, this is not possible with cross-compilers, in which software is compiled, assembled and linked into a target architecture other than that of the host computer. In fact, this is the very nature of embedded development – we have to create software on a host computer, and then compile, assemble and link into a target architecture. It is not possible for a debugger on a host to see what is taking place inside a target processor.

As a result, some embedded debugging tools are available, especially for large microprocessors. Often they involve an intermediate device, i.e,. hardware debugger or emulator, such as those depicted in Fig. 4.1, between the host and the target processors.

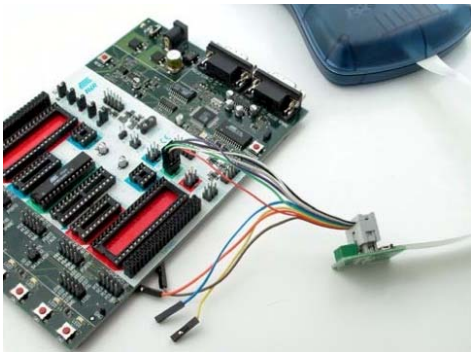
Such a hardware debugger usually has a second microprocessor inside that is able to understand the same machine instructions as that of the target processor. Additionally, the target architecture has to provide the ability to allow the debugger to halt its program counter and look at registers and memory, etc. The host debugging software tool then communicates with the hardware debugger in order to perform the same kind of debugging abilities as that previously stated for the application-level development environment.

While a hardware debugger is a great tool to have, it is not always available for all platforms. In particular, often smaller microcontrollers will not have such devices available (although it never hurts to check on this at the start of a new project!). Additionally, hardware debuggers can add significant cost to a project and may not always fit into a small budget. And so, engineers have to resort to simpler methods for debugging their software. For anyone who has developed PC console application software, the idea is to add temporary output instructions that are removed once the program is working as desired. Typically, the standard C function `printf()` (see Sec. 2.3) is used to print text, possibly containing current memory values, to the console (i.e., output terminal). Again, the observant reader would have noticed that most embedded platforms do not have a convenient standard-output console. Fortunately, many embedded processors do have at least one serial data port, which will often be used during development for which the embedded software can dump information; this is the case for the Arduino development board. In the event that no console is available at all, the most common method for embedded debugging is toggling a general I/O port pin between '1' and '0' which can be observed either via an LED or an oscilloscope. It turns out

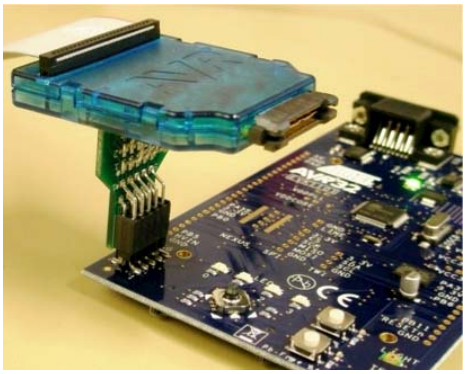


AVR JTAGICE mkII

AVR ONE!



Connected to Target



Connected to Target

Figure 4.1: Photographs of two different hardware debuggers ATMEL (2006, 2010).

this is a very useful method for real-time debugging, in which critical timing can be measured, or to determine if an interrupt is occurring.

4.2 DEBUGGING THE ARDUINO TUTORIAL

The purpose of this section is to introduce the Arduino Wiring `Serial` library function calls as a debugging tool, and to practice some compile-time and run-time debugging.

1. Visit http://arduino.cc/en/Reference/Serial to review the Arduino `Serial` library functions `begin()`, and `write()`.

2. Use the `Serial.begin()` and `Serial.write()` methods to complete the following "Hello, World!" program. Note that the C function `snprintf()` is the same as `printf()`, except it prints a formatted string to a memory buffer instead of the standard output port. So, you can do any sort of printing to our serial console by first printing to a buffer and then writing the buffer out the serial port.

```
#define DEBUG_OUTPUT_MESSAGE_MAX_LENGTH 80

void setup()
{
   char message[DEBUG_OUTPUT_MESSAGE_MAX_LENGTH];

   snprintf(message, DEBUG_OUTPUT_MESSAGE_MAX_LENGTH, "Hello,
       World!\n");
}

void loop()
{
}
```

3. Compile and upload your program to the development board. Open the serial monitor, reset your board and enjoy your first program. Tip: make sure that the baud rate you use in the `Serial.begin()` method is the same as your serial monitor setting. You can change the baud rate to any of the acceptable values; the boot loader will always reset the serial port so you can continue downloading new programs later.

PROBLEMS

4.1 Practice compile-time debugging by fixing all of the syntax errors in the following listing. Do yourself a favor and work through all the bugs using the Arduino IDE compile output information. Turn in the source code listing of your debugged program. Clearly state all of the syntax errors that you found.

```
void MyDelay(unsigned long mSecondsApx);

void setup()
{
   unsigned char *portDDRB;

   portDDRB = (unsigned char *) 0x24;

   *portDDRB |= 0x20;
```

```
}

void loop()
{
   unsigned char *portB;

   portB = (unsigned char *) 0x25;

   *portB |= 0x20;
   MyDelay{1000}
   *portB &= 0xDF:
   MyDelay[1000],
}

void MyDelay(unsigned long mSecondsApx)
{
   volatile unsigned long i;
   unsigned long endTime = 1000 * mSecondsApx;

   for (i = 0; i < endTime; i++);
}
```

4.2 Practice run-time debugging by fixing the following listing, so that the LED will blink like it was meant to. Note: you should be able to do this with two minor changes.

```
void NewDelay(unsigned char mSecondsApx);

void setup()
{
   unsigned char *portDDRB;

   portDDRB = (unsigned char *) 0x24;

   *portDDRB |= 0x20;
}

void loop()
{
   unsigned char *portB;

   portB = (unsigned char *) 0x25;

   *portB |= 0x20;
   NewDelay(100);
   *portB &= 0xDF;
   NewDelay(100);
}
```

```
void NewDelay(unsigned char mSecondsApx)
{
   volatile unsigned char i;
   unsigned long endTime = 1000 * mSecondsApx;

   for (i = 0; i < endTime; i++);
}
```

4.3 Explain why the original program in problem 4.2 was not working, and what you did to fix it.

4.4 Practice run-time debugging by fixing the following listing, so that the LED will blink like it was meant to.

```
void NewDelay(unsigned long mSecondsApx);

void setup()
{
   unsigned char *portDDRB;

   portDDRB = (unsigned char *) 0x24;

   *portDDRB |= 0x20;
}

void loop()
{
   unsigned char *portB;

   portB = (unsigned char *) 0x25;

   *portB |= 0x20;
   NewDelay(100);
   *portB &= 0xDF;
   NewDelay(100);
}

void NewDelay(unsigned long mSecondsApx)
{
   volatile unsigned long i;
   unsigned char j;
   unsigned long k;
   unsigned long endTime = 100 * mSecondsApx;

   for (i = 0; i < endTime; i++)
   {
      j = 10;
      do
      {
```

```
            j = j - 1;
            k = i / j;
        } while (k > 0);
    }
}
```

4.5 Explain why the original program in problem 4.4 was not working, and what you did to fix it.

4.6 Practice run-time debugging by fixing the following listing, so that the LED will blink like it was meant to.

```
void NewDelay(unsigned long mSecondsApx);

void setup()
{
    unsigned char *portDDRB;

    portDDRB = (unsigned char *) 0x24;

    *portDDRB |= 0x20;
}

void loop()
{
    unsigned char *portB;

    portB = (unsigned char *) 0x25;

    *portB |= 0x20;
    NewDelay(100);
    *portB &= 0xDF;
    NewDelay(100);
}

void NewDelay(unsigned long mSecondsApx)
{
    volatile unsigned long i;
    unsigned char j = 0;
    unsigned long endTime = 100 * mSecondsApx;

    i = 0;
    while (j = 0)
    {
        i++;
        if (i = endTime)
        {
            j = 1;
        }
```

```
    }
}
```

4.7 Explain why the original program in problem 4.6 was not working, and what you did to fix it.

CHAPTER 5

ATmega328P Architecture

Before we can learn how to control peripheral devices in general, we need to understand how our specific microcontroller is structured and functions. This is typically referred to as its *architecture*. The schematic of the Arduino in Apx. A shows the ATMEL ATmega328P microcontroller is at the heart of the development board. This brief chapter is meant to introduce the basic structure and operation of the microcontroller with specific emphasis on the AVR CPU. Most information presented within this chapter comes directly from ATMEL (2009).

5.1 OVERVIEW

The ATmega328P is a low-power CMOS 8-bit microcontroller based on the AVR enhanced Reduced Instruction Set Computing (*RISC*) architecture. A simplified block diagram containing the most pertinent components of the ATmega328P microcontroller is presented in Fig. 5.1.

The ATmega328P provides the following significant features: 32 KBytes of In-System Programmable (ISP) flash to store programs, 2 KBytes SRAM to hold run-time variables, 1 KBytes EEPROM to store any data that programs may wish to retain after power is cycled, 23 general purpose input/output (GPIO) lines, 32 general purpose working registers, three Timer/Counters with compare modes, internal and external interrupts, a Universal Synchronous/Asynchronous Receiver/-Transmitter (USART), a 2-wire Serial Interface (TWI) serial port, a Serial Peripheral Interface (SPI) serial port, and a 6-channel 10-bit Analog/Digital Converter (ADC).

All of these peripheral devices are controlled via sets of specific registers, each of which is connected to the 8-bit data bus. Thus, a program interacts with each peripheral by accessing various memory-mapped registers, for example via C pointers. Additionally, each peripheral device accesses the off-chip pins via one of the three available ports B, C or D. Each port is configurable via memory-mapped registers to allow different functions access to external pins; a process called pin-multiplexing.

The on-chip ISP flash allows the program memory to be changed via either a SPI serial interface, a conventional non-volatile memory programmer, or an on-chip boot loader running on the AVR core. Note that the latter method is that used by the Arduino IDE. The boot loader can use any interface to download the application program in the application flash memory. The Arduino boot loader uses the USART configured for RS-232 via the USB-to-serial converter chip on the development board. Software in the boot flash section will continue to run while the application flash section is updated. By combining an 8-bit RISC CPU with ISP flash on a monolithic chip,

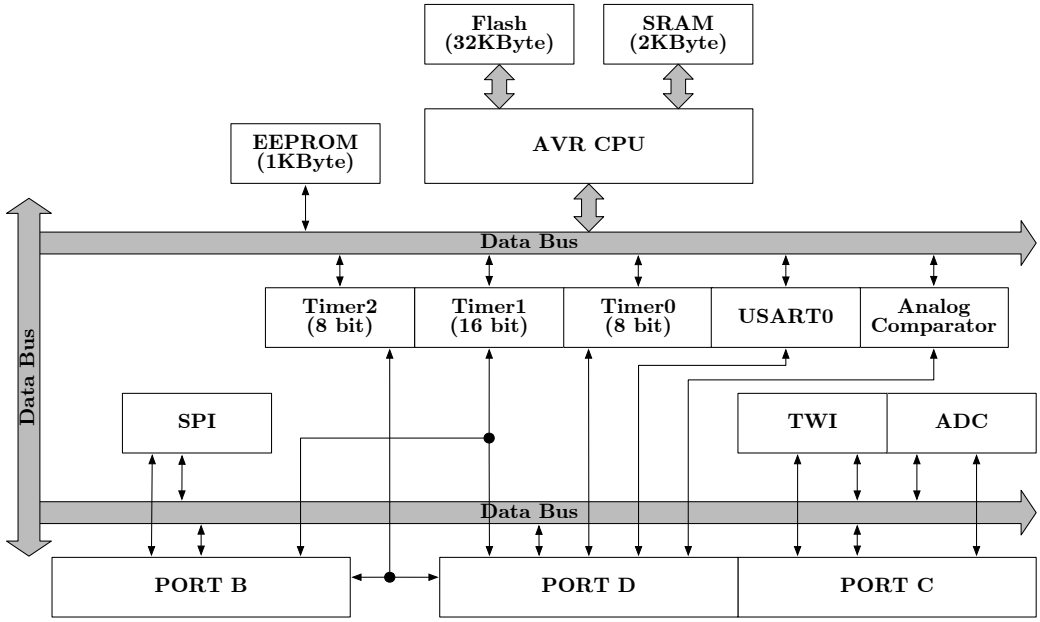


Figure 5.1: ATmega328P block diagram, adapted from ATMEL (2009).

the ATMEL ATmega328P is a powerful microcontroller that provides a highly flexible and cost effective solution to many embedded control applications.

5.2 AVR CPU CORE

At the center of the microcontroller design is the most significant component, the AVR CPU which is further broken down in Fig. 5.2. The main function of the CPU core is to ensure correct program execution. The CPU must therefore be able to access memories, perform calculations, control peripherals, and handle interrupts.

In order to maximize performance and parallelism, the AVR uses a *Harvard architecture* with physically separate storage and buses for program and data. Note that this is in contrast to the *von Neumann architecture* which operates with a single storage structure to hold both program and data. So, for the AVR CPU, while one instruction is being executed, the next instruction is pre-fetched from the program memory. This concept enables instructions to be executed in every clock cycle. The program memory, or code-space, is defined as the entire ISP flash memory.

The 32, 8-bit general purpose working registers are discussed in detail in Apx. C. Notice that they are directly connected to the ALU. The AVR ALU operates in direct connection with

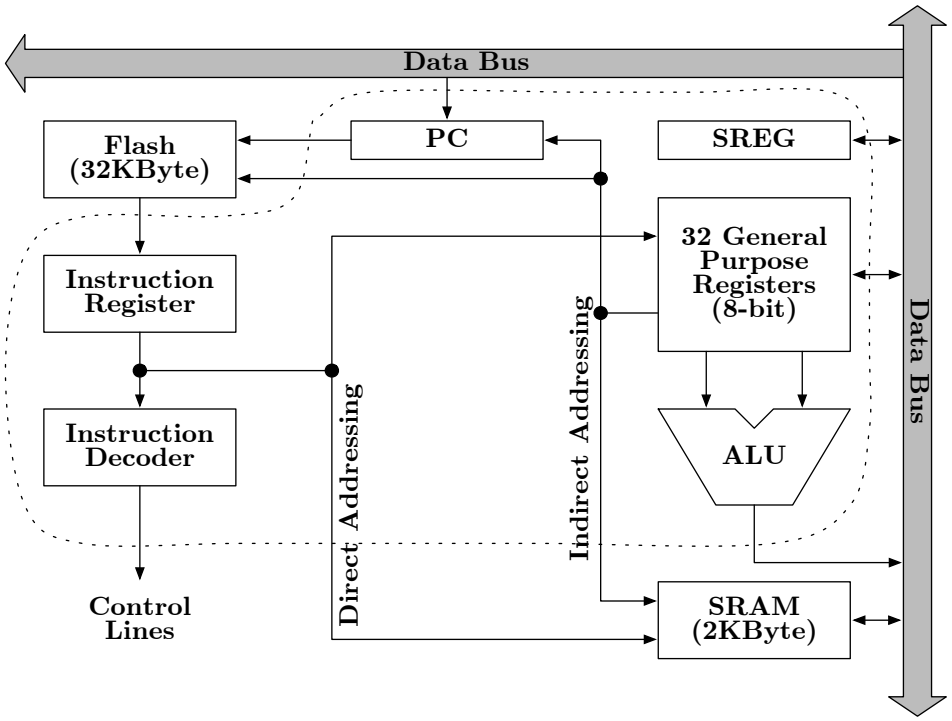


Figure 5.2: AVR CPU block diagram, adapted from ATMEL (2009).

all the 32 general purpose working registers. Within a single clock cycle, arithmetic operations between general purpose registers or between a register and an immediate are executed. The ALU operations are divided into the three main categories arithmetic, logical, and bit-functions. After an arithmetic operation, the status register (SREG) is updated to reflect information about the result of the operation. The SREG is described in full detail in Apx. C.

Six of the 32 registers can be used as three 16-bit indirect address register pointers for data space addressing, enabling efficient address calculations. One of the these address pointers can also be used as an address pointer for look up tables in flash program memory. These added function registers are the 16-bit X-, Y-, and Z-registers, described in more detail in Apx. C.

Program flow is provided by conditional and unconditional jump and call instructions, able to directly address the whole address space. Most AVR instructions have a single 16-bit word format. Every program memory address contains a 16- or 32-bit instruction.

During interrupts and subroutine calls, the return address Program Counter (PC) is stored on the stack. The stack is effectively allocated in the general data SRAM, and, consequently, the

stack size is only limited by the total SRAM size and the usage of the SRAM. All user programs must initialize the SP in the reset routine before subroutines or interrupts are executed. The Stack Pointer (SP) is read/write accessible in the I/O space and is described in greater detail in Apx. C. The data SRAM can easily be accessed through the five different addressing modes supported in the AVR architecture.

SUMMARY

The Arduino development board contains an ATMEL ATmega328P microcontroller as its embedded controlling device. The material presented in this brief chapter was meant to introduce the basic functionality and operation of the processor. In particular, the block diagrams in Fig. 5.1 and Fig. 5.2 will be frequently referred to in subsequent chapters when introducing the various peripheral devices. You will see a recurring pattern when configuring the various peripherals. In particular, each on-chip peripheral device has its own set of registers. Each register is memory-mapped, and so connected to the 8-bit data bus. When the program needs to access a peripheral, all it needs to do is access the registers via their address, then set and clear the bits to achieve the desired configurations.

CHAPTER 6

General-Purpose Input/Output

6.1 OUTPUT

6.1.1 INTRODUCTION

The most commonly used microprocessor component within embedded systems is probably the set of *general-purpose input/output* (GPIO) ports. As the name implies, these hardware components provide a very generic interface to external peripheral devices in the form of individual digital lines. Each GPIO line has a pathway from the microcontroller integrated circuit (IC) to an external pin on the microcontroller package. As a result, GPIO lines can be connected to any device that can source or sink digital signals (e.g., buttons or LEDs), and software can be written to directly control whether the pin outputs a '1' or '0' or reads digital values as an input pin.

6.1.2 BASIC OPERATION

Shown in Fig. 6.1 is a stripped-down schematic of a single port pin on the ATmega328P containing only the Data Direction Register (DDR) bit and the data bit. While this circuitry is specific to the ATmega328P processor, it represents very typical architecture to most microprocessors and microcontrollers. That is, it is very common for a microprocessor to provide bi-directional GPIO lines for generic usage within embedded systems. This schematic details GPIO pin *n* on port *x*. From the simplified figure, there are two D flip-flop memory elements for each GPIO line. The bottom D flip-flop stores whatever value is on line `Port`*x*`_Bit`*n* when the control signal `WritePort`*x*`_Bit`*n* goes from low to high; this will occur by the CPU generating the appropriate signals when the program executes an instruction to write to this particular register. In a similar sense, the top D flip-flop stores the value present on `DDR`*x*`_Bit`*n* when the control signal `WriteDDR`*x*`_Bit`*n* occurs. The three tri-state buffers allow their input signals to pass when enabled; otherwise, they present a high-impedance state to the output (i.e., disconnecting the line from the output).

So, the basic operation to output a digital signal on pin *n* of port *x* is to:

1. write a '1' into `DDR`*x*`_Bit`*n*, then
2. write either a '1' or '0' into `Port`*x*`_Bit`*n*.

The first step results in enabling the tri-state buffer connecting the output of the bottom D flip-flop to the actual pin going off-chip. The second step stores whatever signal is desired which will be passed through the tri-state buffer to the pin. This configuration makes the port pin an output, as the CPU is now free to store '1's and '0's into the bottom D flip-flop, which will then always drive

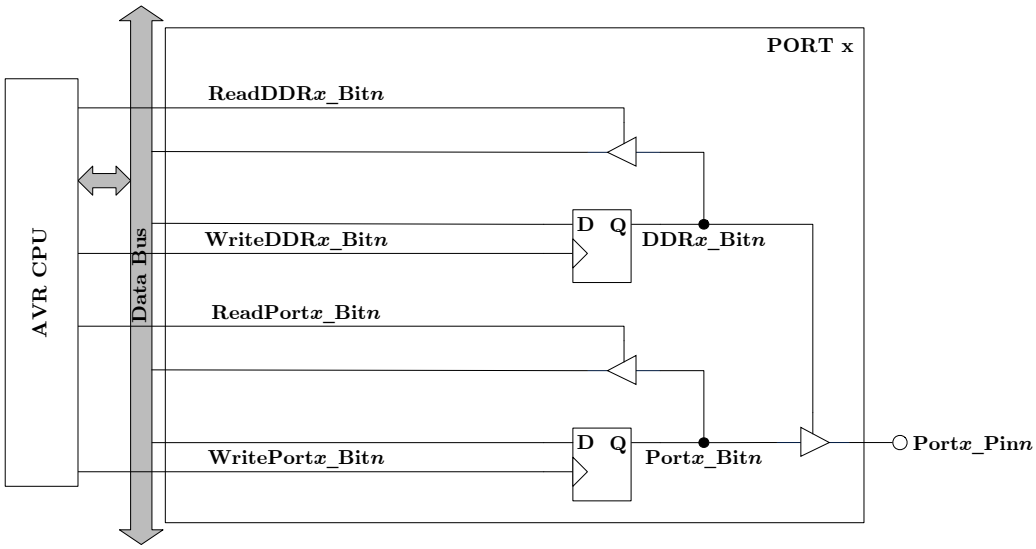


Figure 6.1: A schematic of a single port pin containing only the Data Direction Register (DDR) bit and the Data bit, adapted from ATMEL (2009).

the pin high or low. Note that if a '0' is written into `DDR`*x*`_Bit`*n*, the tri-state buffer is disabled, and the pin is no longer driven high or low; this will be revisited in Sec. 6.2. Finally, note the CPU is able to read the contents of both D flip-flops at any time by enabling the associated `Read`*x*`_Bit`*n* control lines, which will then pass the contents of either D flip-flop onto the data bus.

6.1.3 PIN-MUXING

One complication with GPIO lines typical to most microprocessor architectures is the process called *pin-muxing*. The idea is that microprocessors provide plenty of functionality on their tiny IC, but each sub-system requires its own set of I/O pins necessary to drive the behavior. As a result, if a microprocessor provided a unique pin for each signal possible, the package would grow to undesirable size. Additionally, most applications do not make use of every on-chip peripheral device for their specific embedded system. Thus, manufacturers often if not always use multiplexers on pins in order to share the pin among sub-systems. For example, if an embedded system calls for the processor to interface with peripheral device *y*, a couple of pins can be dedicated to its functionality, and so, they can not be used for GPIO. Usually, specific registers are used to control the pin-muxing configuration of the processor. On the ATmega328P, pin-muxing is as depicted in Fig. 6.2, in which the specific peripheral sub-systems will generate their own override signals in order to obtain control of individual pins.

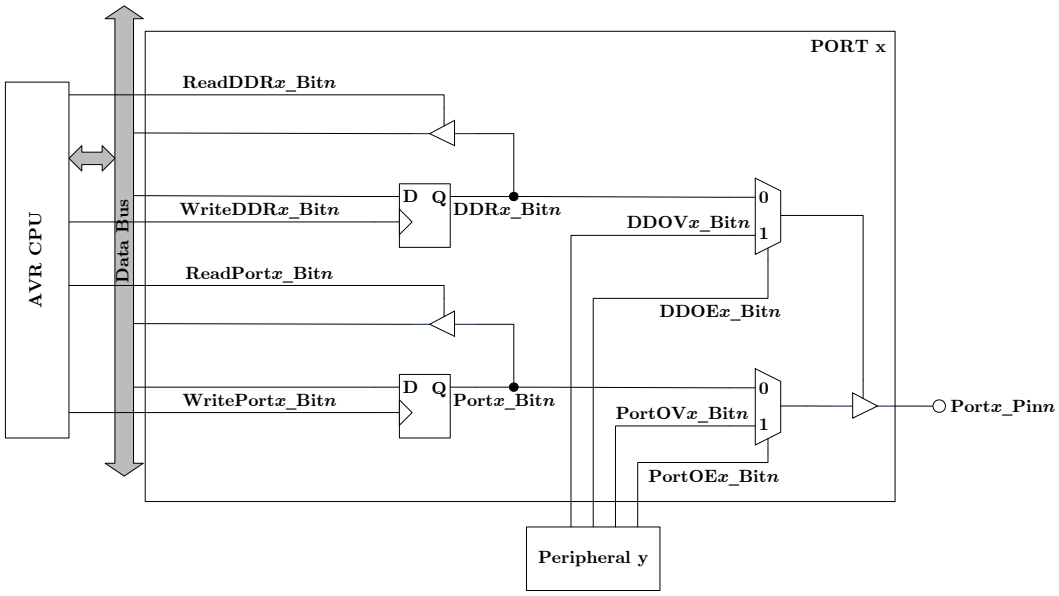


Figure 6.2: A schematic of a single port pin containing the DDR and Data bits with the addition of pin-mux multiplexers, adapted from ATMEL (2009).

New to the schematic are two multiplexers, one for each D flip-flop output. Each multiplexer will pass the D flip-flop value as long as the associated override enable line from peripheral *y* is '0' (i.e., `PortOE`*x*`_Bit`*n* and `DDOE`*x*`_Bit`*n*). When either line goes high, the corresponding D flip-flop value is blocked and replaced with the associated override value line from peripheral *y* (i.e., `PortOV`*x*`_Bit`*n* and `DDOV`*x*`_Bit`*n*). In this way, if peripheral *y*'s sub-circuitry is enabled, it will automatically make use of the associated pins. To make use of GPIO lines, care must be taken to ensure the muxed devices are turned off or disabled.

See Apx. A for a complete table of all GPIO port pins and their alternative functions via pin-muxing. In general, if the peripheral device is enabled, it will override the ability to use a pin as general I/O.

The final step in understanding GPIO ports is handled by combining groups of pins into registers, as depicted in Figure 6.3. This figure details the fact that individual D flip-flops are not addressable directly, but they are addressed in groups of a certain size. In the case of the ATmega328P, port registers are all accessed in eight-bit units. The primary change in the figure is the combination of D flip-flops into 8-bit registers, all of which store their input bit values when the single control line `WriteDDR`*x* or `WritePort`*x* goes from low to high. This is a significant point: **in order to change a single bit in a register, all seven other bits must also be written**. The typical method for handling

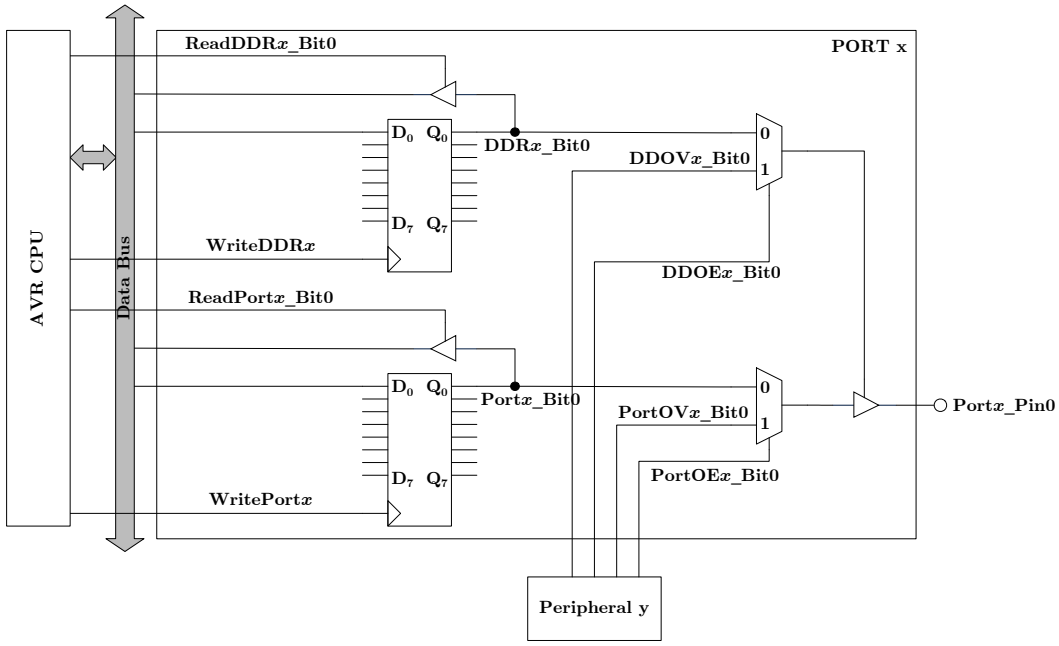


Figure 6.3: A schematic of an eight-bit port showing only a single DDR and Data bit with the addition of pin-mux multiplexers.

this issue is by first reading the current contents of the register, then masking the bit-of-interest, and finally writing the entire register back. While the process reads and writes the entire eight bits, only the bit-of-interest is changed.

Note for simplicity, the figure does not depict all tri-state buffers, multiplexers, nor pins; there are eight instances of each.

6.2 INPUT

6.2.1 INTRODUCTION

This section continues exploring the most commonly used microprocessor component within embedded systems, the set of GPIO ports. By now, it is assumed the reader has already been introduced to GPIO port pins. In particular, the reader should be familiar with outputting logic-high and low values on individual pins in order to drive peripheral devices, for example turning on and off an LED. Generally speaking, the two port registers that are most important for setting a port pin as an output and then driving a value on that pin are the DDR and the Data register. For the ATmega328P, a port pin is configured as an output when a '1' is written into the corresponding DDR bit position,

and then the value driven on the pin is determined by the bit value in the respective data register. As the name implies, the DDR is used to configure a GPIO port pin to be an output when a '1' is loaded; otherwise, it is an input port pin when a '0' is loaded.

Recall that shown in Fig. 6.1 is a stripped-down schematic of a single port pin from the ATmega328P containing only the DDR bit and the data bit. While this circuitry is specific to the ATmega328P microcontroller, it represents very typical architecture to most microprocessors and microcontrollers. This figure should be familiar to the reader from learning about outputting values on GPIO port pins. Of particular interest now is the DDR bit that acts as the enable signal on the tri-state buffer connecting the data bit to the external port pin, `Pinx_Bitn`. Consider what happens when `DDRx_Bitn` contains a '0', which then disables the tri-state buffer. This condition blocks the data bit from driving the external pin, and effectively disconnects the pathway between the data register and the external pin. At this time, it is safe for an external device to drive the line. Now consider the additional circuitry present in Fig. 6.4, which connects the external port pin, `Pinx_Bitn`, to a new register `Pinx_Bitn`.

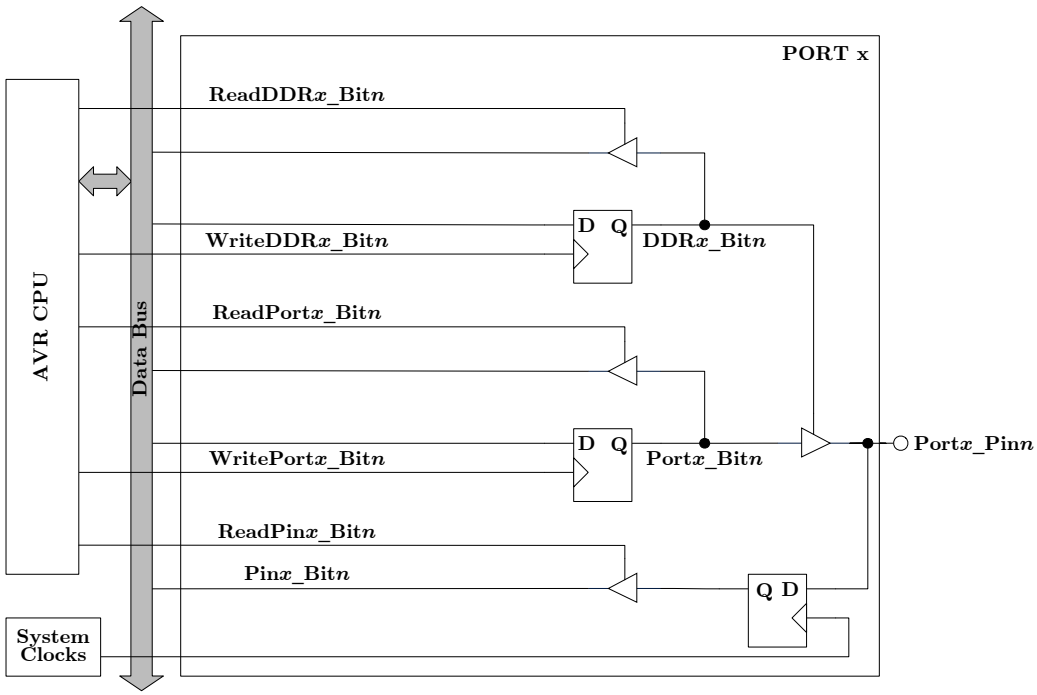


Figure 6.4: A schematic of a single port pin containing only the DDR bit, the Data bit and the input Pin bit, adapted from ATMEL (2009).

As can be seen from the figure, when the DDR contains a '0', the tri-state buffer prevents the Data bit from passing through, safely allowing an external device to drive the pin. The driving signal is latched into the new register, which can then be read by the CPU by simply accessing the appropriate bit out of the port Pin register. Note that this method is specific to the ATmega328P; many processors re-use the data register for both outputting-from as well as inputting-to the microcontroller.

So, the basic operation to input a digital signal on pin *n* of port *x* is to:

1. write a '0' into `DDRx_Bitn`, then
2. read the value out of `Pinx_Bitn`.

The first step results in disabling the tri-state buffer connecting the output of the data D flip-flop to the actual pin going off-chip. The second step reads whatever signal is currently stored in the register whose input is connected directly to the pin. This configuration makes the port pin an input, as the CPU is now able to read '1's and '0's out of the bottom D flip-flop, which is constantly sampling the external pin via the control signal coming from the System Clocks sub-circuitry.

6.2.2 INTERNAL PULL-UP RESISTOR

Generally, when a port pin is not being used, the microcontroller will default the pins to input, in order to save on power. However, if the pin is unused, then it is probably not connected to any external circuitry. As a result, the input pin is *floating*, and the input signal to the `Pinx_Bitn` D flip-flop is unknown, which can lead to strange and unexpected behavior; for example, an unexpected/unwanted interrupt might occur. A very standard method for managing unused input pins is to connect a fairly large resistance between the pin and the power supply. In this way, the port pin is very weakly pulled high, and no unwanted transitions will occur. It turns out that because this method is so standard, most microprocessors will provide internal pull-up resistors on any port pins that may be configured as inputs.

The ATmega328P provides internal pull-up resistors on all port pins, which can be enabled and disabled based on the schematic depicted in Fig. 6.5. New to the schematic is an AND gate that has inputs of the `DDRx_Bitn` output via an inversion and the `Portx_Bitn` output. The output of the AND gate goes to the gate of a transistor, which then allows a channel pathway between the power supply and the pin perhaps through a resistor, although the transistor can be made to act as the resistor itself. As a result, an internal pull-up behavior is enabled via the following procedure:

1. write a '0' into `DDRx_Bitn`, then
2. write a '1' into `Portx_Bitn`.

To keep a port pin configured as an input, but to disable the internal pull-up, a '0' would be written into `Portx_Bitn` instead. Note that this procedure is very specific to the ATmega328P microcontroller; other processors offer their own methods for managing internal pull-up resistors.

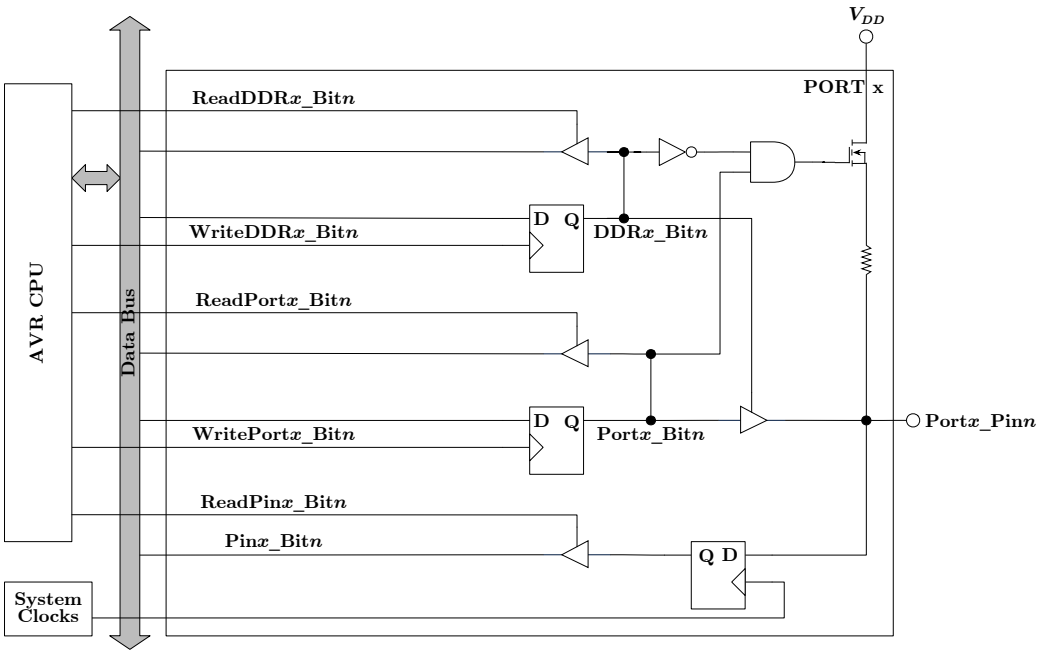


Figure 6.5: A schematic of a single port pin containing the DDR bit, the Data bit and the input Pin bit with the addition of a pull-up resistor, adapted from ATMEL (2009).

6.3 ACCESSING GPIO LINES IN C

6.3.1 MANAGING OUTPUTS

The first step for accessing GPIO lines in software is identifying the addresses at which the DDR and Data registers exist. In the case of the ATmega328P, three ports exist, `PORTB`, `PORTC` and `PORTD` all specified in Sec. 6.4. Note that `PORTC` only has seven bits available. From C, it is possible to read and write the contents of these registers using pointers to eight-bit elements as in the following example.

Example 6.1

```
unsigned char *portDDRB;

portDDRB = (unsigned char *) 0x24;
```

The lines in Ex. 6.1 declares a pointer variable that points to an eight-bit `unsigned char`, and then loads that pointer variable with the hard-coded value of the data-space address for the

desired register. Once the pointer is available, the register can be read-from and written-to by simply dereferencing the pointer. The final trick is in knowing how to access specific bits within the register. Suppose the program needs to clear bit 6 and set bit 5. The following example shows how to form the proper bit-masks and the usage of bitwise AND and OR operations.

Example 6.2

```
#define BIT6_MASK 0x40 // 0100 0000
#define BIT5_MASK 0x20 // 0010 0000

*someRegisterPointer = (*someRegisterPointer) & (~BIT6_MASK);
*someRegisterPointer = (*someRegisterPointer) | BIT5_MASK;
```

The first line makes use of the ones' complement of the bit mask to generate the bit vector 10111111 which is then ANDed bit-per-bit along the contents of the register. The result, which has a '0' forced into the sixth bit, is written back into the register. The second line simply performs a bitwise OR of the contents of the register with the bit vector 00100000, which results in setting a '1' into the fifth bit.

6.3.2 MANAGING INPUTS

There are two methods for managing GPIO input port pins in software, polling and usage of interrupts. Because interrupts are generally more problematic to work with, this topic will be deferred to a later chapter. Instead, the method of polling an input pin will be used to capture when an external signal changes.

6.3.2.1 Polling

Polling is the simple process of repeatedly reading the value of the input pin(s) and comparing the current value to the previously-read value to determine if an external signal event has occurred; i.e., when a digital line switches states from '1' to '0' or '0' to '1'. As was done for outputting on GPIO port pins, the first step for accessing GPIO inputs in software is identifying the addresses at which the DDR, Data and Pin registers exist.

In the case of the ATmega328P, three ports exist, `PORTB`, `PORTC` and `PORTD` all specified in Sec. 6.4. From C, it is possible to configure a port pin as an input using pointers to eight-bit elements as in the following example.

Example 6.3

```
unsigned char *portDDRB;
unsigned char *portDataB;
unsigned char *portPinB;

portDDRB = (unsigned char *) 0x24;
```

```
portDataB = (unsigned char *) 0x25;
portPinB = (unsigned char *) 0x23;
```

The code in the previous example declares a pointer variable that points to an eight-bit `unsigned char`, and then it loads that pointer variable with the hard-coded value of the data-space address for the desired register. Now suppose the program needs to make bit 0 of port B an input with an internal pull-up resistor enabled. The following example demonstrates.

Example 6.4

```
#define BIT0_MASK 0x01 // 0000 0001

*portDDRB = (*portDDRB) & (~BIT0_MASK);  // configure bit 0 as an input
*portDataB = (*portDataB) | BIT0_MASK;   // turn on the bit 0 pull-up
    resistor
```

The final step is to continuously sample the pin-of-interest, waiting for it to change. When a change occurs, the program can "do something". Consider the following code piece that would sit in the `loop()` function of the Arduino sketch.

Example 6.5

```
void loop()
{
   unsigned char previousSample;
   unsigned char currentSample;

   currentSample = (*portPinB) & BIT0_MASK; // only interested in bit 0
   if (currentSample != previousSample)
   {
      // a change has occurred; we should do something
   }

   // store the current sample as the next previous sample
   previousSample = currentSample;
}
```

Notice how this code doesn't do anything interesting until a change is measured on the input port pin. It just keeps comparing the "previous sample" with the "current sample". In this way, the software is sampling the input data and detecting when a signal event occurs.

6.4 PERTINENT REGISTER DESCRIPTIONS

The information presented in this section was taken from ATMEL (2009).

6.4.1 PORTB - THE PORT B DATA REGISTER

Bit	7	6	5	4	3	2	1	0
0x25	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

- PORTB7-0: GPIO data value stored in bit *n*.

6.4.2 DDRB - THE PORT B DATA DIRECTION REGISTER

Bit	7	6	5	4	3	2	1	0
0x24	DDRB7	DDRB6	DDRB5	DDRB4	DDRB3	DDRB2	DDRB1	DDRB0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

- DDRB7-0: selects the direction of pin *n*. If DDRB*n* is written '1', then PORTB*n* is configured as an output pin. If DDRB*n* is written '0', then PORTB*n* is configured as an input pin.

6.4.3 PINB - THE PORT B INPUT PINS ADDRESS

Bit	7	6	5	4	3	2	1	0
0x23	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0
Read/Write	R	R	R	R	R	R	R	R
Default	-	-	-	-	-	-	-	-

- PINB7-0: logic value present on external pin *n*.

6.4.4 PORTC - THE PORT C DATA REGISTER

Bit	7	6	5	4	3	2	1	0
0x28	-	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0
Read/Write	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

- PORTC6-0: GPIO data value stored in bit *n*.

6.4.5 DDRC - THE PORT C DATA DIRECTION REGISTER

Bit	7	6	5	4	3	2	1	0
0x27	-	DDRC6	DDRC5	DDRC4	DDRC3	DDRC2	DDRC1	DDRC0
Read/Write	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

- DDRC6-0: selects the direction of pin *n*. If DDRC*n* is written '1', then PORTC*n* is configured as an output pin. If DDRC*n* is written '0', then PORTC*n* is configured as an input pin.

6.4.6 PINC - THE PORT C INPUT PINS ADDRESS

Bit	7	6	5	4	3	2	1	0
0x26	-	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0
Read/Write	R	R	R	R	R	R	R	R
Default	0	-	-	-	-	-	-	-

- PINC6-0: logic value present on external pin *n*.

6.4.7 PORTD - THE PORT D DATA REGISTER

Bit	7	6	5	4	3	2	1	0
0x2B	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

- PORTD7-0: GPIO data value stored in bit *n*.

6.4.8 DDRD - THE PORT D DATA DIRECTION REGISTER

Bit	7	6	5	4	3	2	1	0
0x2A	DDRD7	DDRD6	DDRD5	DDRD4	DDRD3	DDRD2	DDRD1	DDRD0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

- DDRD7-0: selects the direction of pin *n*. If DDRD*n* is written '1', then PORTD*n* is configured as an output pin. If DDRD*n* is written '0', then PORTD*n* is configured as an input pin.

6.4.9 PIND - THE PORT D INPUT PINS ADDRESS

Bit	7	6	5	4	3	2	1	0
0x29	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0
Read/Write	R	R	R	R	R	R	R	R
Default	-	-	-	-	-	-	-	-

- PIND7-0: logic value present on external pin *n*.

PROBLEMS

6.1 Using the ATmega328P pin configuration table and the Arduino Duemilanove Schematic in Apx. A, state all of the pins on the three ports that are already dedicated to an alternative

function via pin-muxing. Hint: all the pins in the table correspond to pins on the schematic; some of them will go to specific circuitry other than the headers.

6.2 A 7-segment display is just a set of seven LED bars contained in a convenient package, as depicted in Figure 6.6. They are generally provided in two different packages: common-cathode and common-anode. The common-cathode package ties all cathodes together, which then needs to be connected to ground; each LED can be illuminated by applying positive voltage to the individual anodes (i.e., write a '1' to a port pin connected to that LED bar). The common-anode package ties all anodes together, which then needs to be connected to power; each LED can be illuminated by applying ground to the individual cathodes (i.e., write a '0' to a port pin connected to that LED bar). Write a C function that will take in a `char` variable, and output an appropriate 7-segment set of signals to display the associated alphabetic (both upper and lower case) and numeric character.

- Within the `setup()` function, set seven available port pins (i.e., pins other than those already used for specific functionality) to output by setting the respective port pin data direction register bits to '1'.
- Create a function that takes in a `char` argument, then changes the seven port pins to either '1' or '0' in order to turn on or off the respective LED bar of the display in order to output an image of the corresponding argument. For example, if the argument is 'A', then a, b, c, e, f, g should all be '1' and d should be '0', assuming a common-cathode display (reverse the bit values for a common-anode display).
- Within the `loop()` function, create some test code that will exercise your new function by looping through all the characters. Be sure to add enough `delay()` in between each call in order to see the output on the display.

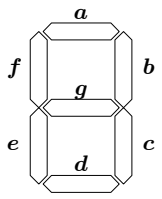


Figure 6.6: A 7-segment display.

Tip: don't try to do the whole thing in one step. Instead, try to get a single LED working. In fact, you already have source code from previous sections that toggle one port pin (PB5 by using J3 pin 6 (labeled 13)).

Tip: be sure you have your LEDs connected correctly. You can do this by first making the LED on the Arduino blink, and then connect the associated port pin (PB5 by using J3 pin 6

(labeled 13)) to an external LED. If you can't make the external LED blink, then you don't have your LED physically connected to your processor correctly.

Tip: Once you have your LEDs connected to the processor, then a good starting point would be to take the "blink" source code and modify it to control one of your LED bars. Once that is working, try to expand your solution to control more than one port pin.

6.3 Because many microcontrollers operate in noisy environments, the first detection of a signal-transition cannot always be trusted (i.e., it might be a glitch on the line). Additionally, many mechanical switches are notorious for *bouncing* when they are first pressed or released. So, you are to write a function that *debounces* an input pin. Your function should be called whenever the first signal-event occurs on an input pin, and it should debounce for a reasonable amount of time (e.g., 20 msec). At the end of the debounce time, the input should be re-sampled and compared to the original state of the line; if they are different, you should declare that a valid transition has occurred.

6.4 Write a program that monitors and debounces an external push-button device. Do something interesting when the user presses the button (e.g., write a message to the terminal using the serial library or use GPIO output pins to light up an LED, etc.). Your program should continuously monitor the push-button switch and respond whenever it is pressed or released after debouncing, of course.

6.5 Consider how we might like to turn on a DC motor in either a forward or reverse direction. This is a simple matter of switching the terminals of the DC motor between power and ground. The circuitry necessary to allow a dynamic connection of terminals between power and ground is called an *H Bridge*, and is shown in Fig. 6.7. The basic idea of the H Bridge is to turn transistors A and D on and B and C off allowing the left terminal a connection to V_{DD} and the right terminal a connection to GND. This will cause the motor to spin in the forward direction. Similarly, when transistors A and D are off and B and C are on, the terminals have the opposite connections, and the motor will turn in the reverse direction.

An important issue with motor control is back electromotive force (back EMF), which is a voltage that occurs in electric motors where there is relative motion between the armature of the motor and the external magnetic field. When the transistors are all turned off to stop the motor from spinning, there is a storage of current in the motor's coil that needs to dissipate. The diodes placed across the source and drain of each transistor allow the current to safely sink out of the motor. If the diodes are missing, the current will still leave the motor, but it does so by increasing the potential enough to force the current back through the transistors. This could cause damage to the transistors, so be sure the diodes are in place.

The following three values should be the only bit patterns written to the four gates of the transistors.

- ABCD = 1100 will turn all four transistors off, and so the motor is off;

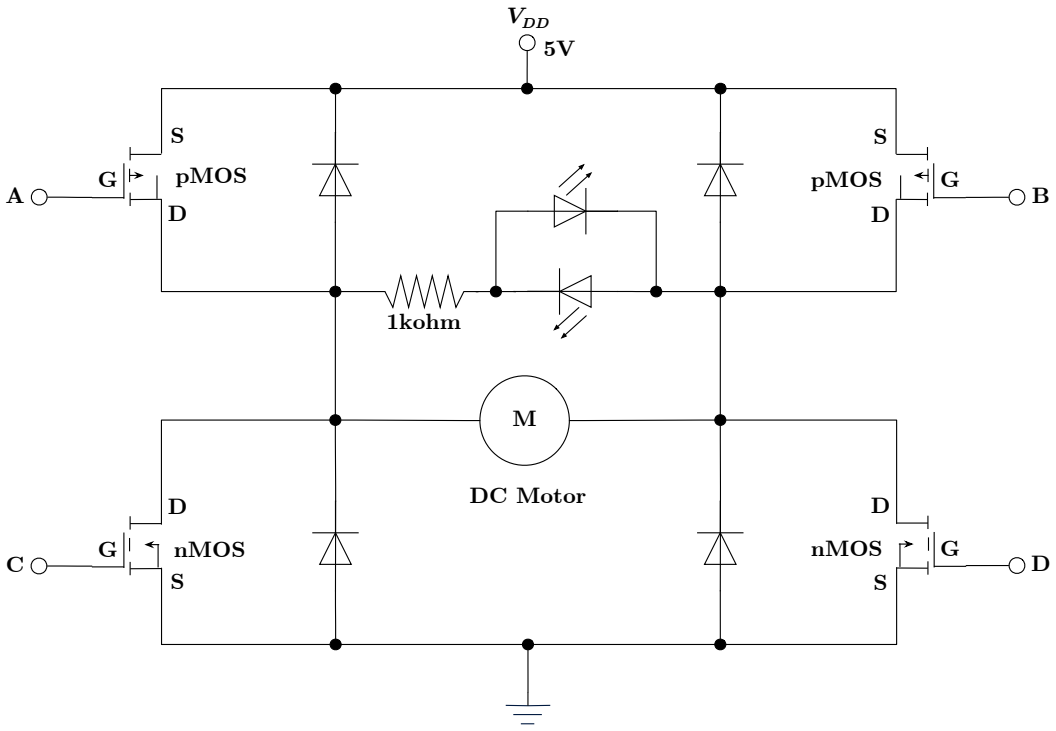


Figure 6.7: A schematic of an H Bridge circuit with back-EMF protection for the use of controlling the direction of a DC motor without damaging the microcontroller.

- ABCD = 0101 will turn A and D on and B and C off, and so the motor is in the forward direction;
- ABCD = 1010 will turn A and D off and B and C on, and so the motor is in the reverse direction.

Warning: if A and C or B and D are ever turned on at the same time, there is a direct connection between power and ground with very little resistance – a transistor will most-certainly blow up.

Build the circuit in Fig. 6.7, and connect four GPIO pins from the Arduino development board to the ABCD gates of the H Bridge. Note that you can leave the motor out of the circuit and see if your program works by using just the LEDs. Once you are confident it is working, you can add the DC motor.

6.6 Referring to problem 6.5, write a function that will turn on the motor in both directions, or turn off the motor based on an input variable.

6.7 Write a program that monitors and debounces an input switch. When the user presses the switch, turn on a DC motor in the forward direction. When the user presses the switch a second time, turn off the motor. You are welcome to modify this step in more interesting ways, but the user should be able to turn on and off the motor via the input push-button switch.

CHAPTER 7

Timer Ports

7.1 PULSE WIDTH MODULATION

7.1.1 INTRODUCTION

Several modulation methods have been developed for applications that require a digital representation of an analog signal. One popular and relevant scheme is *pulse width modulation* (PWM) in which the instantaneous amplitude of an analog signal is represented by the width of periodic square wave. For example, consider the signals depicted in Fig. 7.1. Notice, the PWM version of the signal has a fixed frequency defining the point when a pulse begins. During the period of an individual pulse, the signal remains high for an amount of time proportional to the amplitude of the analog signal.

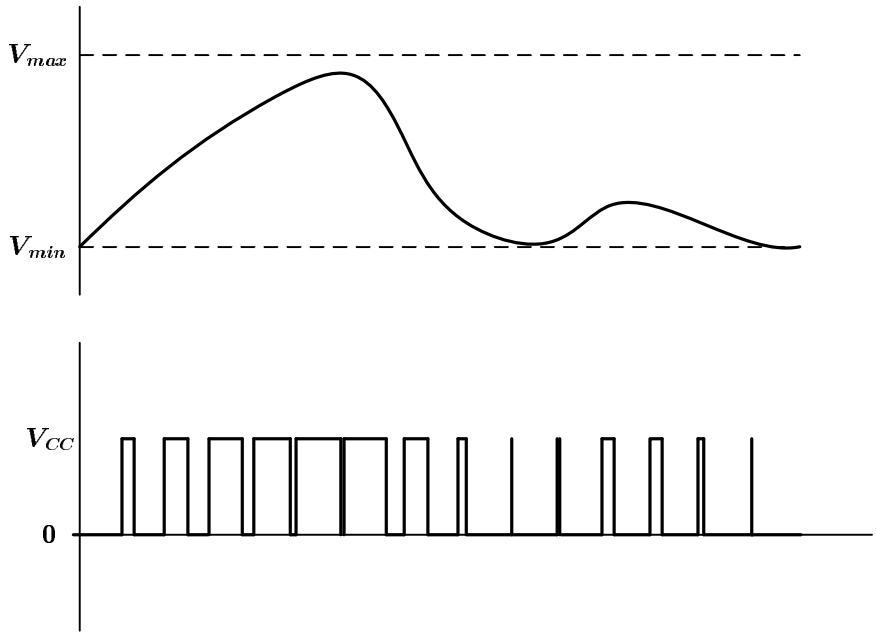


Figure 7.1: An example analog signal and a pulse width modulated representation.

Recall the duty cycle of a square wave is given by

$$D = \frac{\tau}{T}$$

where $0 \leq \tau \leq T$ is the amount of time a signal is non-zero and T is the period of the square wave. Effectively, the pulse duty cycle represents a measure of the analog signal amplitude. Specifically, when the amplitude of a signal is at or below some minimal voltage V_{min}, the PWM pulses have a 0% duty cycle. Similarly, when a signal is at or above some maximal voltage V_{max}, the PWM pulses have a 100% duty cycle. Ideally, when the amplitude of a signal is in between V_{min} and V_{max}, the duty cycle of the PWM pulses is defined by the linear relation

$$D = \frac{V_{sig} - V_{min}}{V_{max} - V_{min}}$$

where V_{sig} is the analog signal.

7.1.2 DEMODULATION

Converting the square-wave signal to the desired analog value is referred to as demodulation, and requires a demodulator circuit to use the pulse widths as a measure for the signal amplitude. Consider the simple low-pass filter (LPF) shown in Fig. 7.2. Recall, the -3 dB (or high-cutoff) frequency is

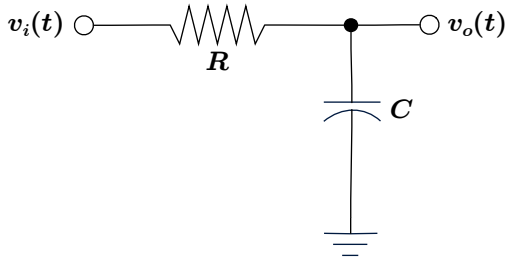


Figure 7.2: A simple low-pass filter.

determined by the choice of R and C using the relation $f_h = \frac{1}{2\pi RC}$. It is a well-known result that if $f_m \ll f_h \ll f_t$, then an approximation of the original signal is recovered. Here, f_m is the maximum frequency of the analog message signal, and f_t is the frequency of the PWM square-wave. Thus, the filter needs to be designed such that the highest frequency of the desired analog signal is less than f_h.

In practice, most systems do not construct an explicit LPF before the load, which is typically either inductive or capacitive. Instead, the natural frequency response of the load is used to demod-

ulate the PWM pulses directly. In either case, effectively the voltage of the original message signal is delivered to the load via averaging over the PWM pulses.

7.1.3 MODULATION

The remaining component to outputting an analog signal from a digital microcontroller is to generate the appropriate PWM waveform, a process referred to as modulation; i.e, an analog message signal modulates the square-wave carrier by altering its duty cycle. Most microcontrollers provide at least one port that has timer sub-circuitry capable of generating PWM signals on a port pin. Typically, one just needs to configure the square-wave frequency and desired duty cycle via a couple of registers. When enabled, the port pin will output a PWM signal that can be demodulated in order to provide an approximation to an analog signal.

In the case of the ATmega328P, there are two 8-bit timers and one 16-bit timer, all of which are capable of generating PWM outputs. The registers are listed in Sec. 7.3.

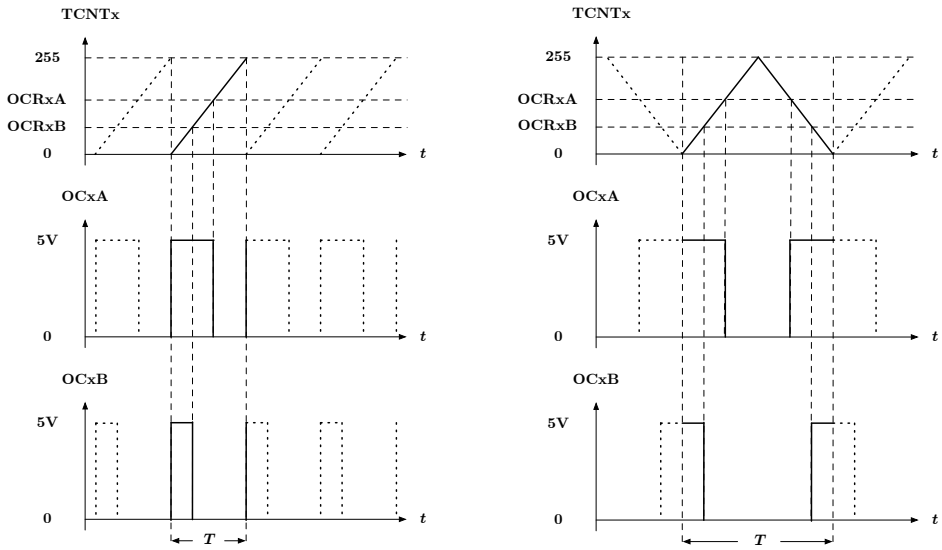
Note, in the `init()` function within the Wiring library, all three timers are pre-initialized with some specific functionality in mind. In particular, the often used `delay()` function relies on a timer 0 interrupt, and so we will avoid using it for the time being.

On the ATmega328P, three waveform generation bits exist within the two timer/counter control registers. Four of the eight possible waveform generation modes involve PWM waveform outputs, two of which are considered fast PWM while the remaining two are called phase-correct PWM. Shown in Fig. 7.3 and Fig. 7.4 are the four different output waveforms given the specified waveform configurations.

In general, the PWM generation circuitry operates based on the 8-bit or 16-bit count register `TCNT` which updates its current value every time there is a clock pulse. As long as the `TCNT` value is below the value stored in the output compare register `OCRA` or `OCRB`, then the associated output pin `OCA` or `OCB` will remain in a specific state, for example, set high. Once the `TCNT` value becomes greater than the compare register value, the output pin will switch to the opposite state, for example clear low. This operation will continue until the timer is disabled.

The first output mode shown in Fig. 7.3(a) represents the waveforms generated given a fast PWM setting where the TOP value is fixed at the maximum 8-bit value of 255. In this mode, two different output compare register values can be set independent of each other, each affecting a different output pin. That is, two separate PWM waveforms may be generated on two different port pins. Additionally, the period of the PWM waveform is $T = 256/f_{clk}$, which is half the period of the phase-correct version, i.e., it is faster.

The second output mode shown in Fig. 7.3(b) represents the waveforms generated given the phase-correct PWM setting where the TOP value is also fixed at the maximum 8-bit value of 255. As in the fast PWM case, two different output compare register values can be set independent of each other, each affecting their own output pin. As can be seen, this mode alters the `TCNT` register behavior in that once the counter reaches the TOP value of 255, it begins counting backwards toward 0. The benefit has to do with the phase of the modulated carrier. In particular, notice the

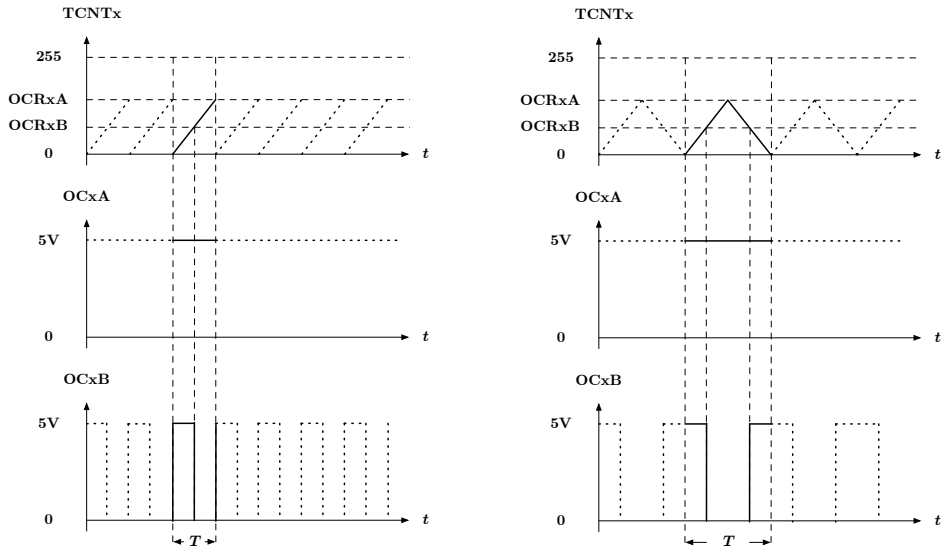


(a) Fast PWM (b) Phase-Correct PWM

Figure 7.3: The two PWM configuration output waveforms with TOP = 255.

narrower pulses of OCB as compared to that of OCA in both Fig. 7.3(a-b). In the fast case, the front edges line up, whereas in the phase-correct case, the center of the pulses line up; that is, the phase of the OCA and OCB waveforms are equivalent. As a result, the period of the PWM waveform is nearly doubled to $T = (2 \times 255)/f_{clk} = 510/f_{clk}$, so less resolution is available. This is not a typo; in the fast PWM case, the counter follows the sequence {0, 1, ..., 254, 255, 0}, which implies there are 256 values in a single period. In the phase-correct case, the counter follows the sequence {0, 1, ..., 254, 255, 254, ..., 1, 0}, which implies there are (255 + 255) values in a single period.

The final two output modes shown in Fig. 7.4 represent the fast and phase-correct PWM waveforms when the TOP value is set to the 8-bit value stored in OCRA. Both of these modes effectively disable the OCA pin functionality at the benefit of increasing the PWM frequency dramatically. In both cases, the TCNT register will count up to the OCRA value, and then either reset to 0 or start counting down toward 0. The only comparison that matters is that to OCRB, which will affect the OCB pin as in the previous cases. The two most significant results are that for a value of N loaded into OCRA, the total number of analog output levels available is reduced from 256 to $N + 1$. However, the periods of the two PWM waveforms are $T = (N + 1)/f_{clk}$ and $T = (2N)/f_{clk}$, respectively.



(a) Fast PWM (b) Phase-Correct PWM

Figure 7.4: The two PWM configuration output waveforms with TOP = `OCRA`.

7.2 INPUT CAPTURE

Many microcontrollers that provide timer ports also include the ability to take a snap-shot of the free-running count value. From ATMEL (2009), we see that the ATmega328P also provides the input capture functionality on one of its timer ports.

The 16-bit timer Timer/Counter1 includes an input capture unit that can capture external events and give them a time-stamp indicating time of occurrence. The external signal indicating an event, or multiple events, can be applied via the `ICP1` pin or alternatively, via the analog-comparator unit. The time-stamps can then be used to calculate frequency, duty-cycle, and other features of the signal applied. Alternatively, the time-stamps can be used for creating a log of the events.

7.3 PERTINENT REGISTER DESCRIPTIONS

The information presented in this section was taken from ATMEL (2009).

7.3.1 TCCR0A - TIMER/COUNTER0 CONTROL REGISTER A

Bit	7	6	5	4	3	2	1	0
0x44	COM0A1	COM0A0	COM0B1	COM0B0	-	-	WGM01	WGM00
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W
Default	0	0	0	0	0	0	0	0

- COM0A1-0: Compare Output Mode for Channel A. These bits control the Output Compare pin (`OC0A = PORTD6`) behavior.

- COM0B1-0: Compare Output Mode for Channel B. These bits control the Output Compare pin (`OC0B = PORTD5`) behavior. If one or both of the `COM0x1:0` bits are set, the `OC0x` output overrides the normal port functionality of the I/O pin it is connected to. However, note that the `DDR` bit corresponding to the `OC0x` pin must be set in order to enable the output driver.

  When `OC0x` is connected to the pin, the function of the `COM0x1:0` bits depends on the `WGM02:0` bit setting. Table 7.1 shows the `COM0x1:0` bit functionality when the `WGM02:0` bits are set to a normal or CTC mode (non-PWM).

Table 7.1: Timer0 Compare Output Mode, non-PWM Mode

COM0x1-0	Description
00	Normal port operation, `OC0x` disconnected
01	Toggle `OC0x` on Compare Match
10	Clear `OC0x` on Compare Match
11	Set `OC0x` on Compare Match

Table 7.2 shows the `COM0x1:0` bit functionality when the `WGM02:0` bits are set to fast PWM mode.

Table 7.2: Timer0 Compare Output Mode, Fast PWM Mode

COM0x1-0	Description
00	Normal port operation, `OC0x` disconnected
01	`WGM02 = 0`: Normal Port Operation, `OC0A` disconnected `OC0B` reserved behavior `WGM02 = 1`: Toggle `OC0A` on Compare Match, `OC0B` reserved behavior
10	Clear `OC0x` on Compare Match, set `OC0x` at 0x00, (non-inverting mode)
11	Set `OC0x` on Compare Match, clear `OC0x` at 0x00, (inverting mode)

Table 7.3 shows the COM0*x*1:0 bit functionality when the WGM02:0 bits are set to phase correct PWM mode.

Table 7.3: Timer0 Compare Output Mode, Phase Correct PWM Mode

COM0x1-0	Description
00	Normal port operation, OC0*x* disconnected
01	WGM02 = 0: Normal Port Operation, OC0A disconnected OC0B reserved behavior WGM02 = 1: Toggle OC0A on Compare Match, OC0B reserved behavior
10	Clear OC0*x* on Compare Match when up-counting Set OC0*x* on Compare Match when down-counting
11	Set OC0*x* on Compare Match when up-counting Clear OC0*x* on Compare Match when down-counting

- WGM01-0: Waveform Generation Mode. Combined with the WGM02 bit found in the TCCR0B Register, these bits control the counting sequence of the counter, the source for maximum (TOP) counter value, and what type of waveform generation to be used, see Table 7.4. Modes of operation supported by the Timer/Counter unit are: Normal mode (counter), Clear Timer on Compare Match (CTC) mode, and two types of PWM modes.

Table 7.4: Timer0 Waveform Generation Mode Bit Description

WGM02-0	Mode	TOP	OCR0x Update	TOV0 Flag Set
000	Normal	0xFF	Immediate	0xFF
001	Phase Correct PWM	0xFF	TOP	0x00
010	CTC	OCR0A	Immediate	0xFF
011	Fast PWM	0xFF	0x00	0xFF
100	Reserved	-	-	-
101	Phase Correct PWM	OCR0A	TOP	0x00
110	Reserved	-	-	-
111	Fast PWM	OCR0A	0x00	TOP

7.3.2 TCCR0B - TIMER/COUNTER0 CONTROL REGISTER B

Bit	7	6	5	4	3	2	1	0
0x45	FOC0A	FOC0B	-	-	WGM02	CS02	CS01	CS00
Read/Write	W	W	R	R	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

- FOC0A: Force Output Compare for Channel A.

- FOC0B: Force Output Compare for Channel B. The FOC0*x* bit is only active when the WGM02:0 bits specify a non-PWM mode.

  However, for ensuring compatibility with future devices, this bit must be set to zero when TCCR0B is written when operating in PWM mode. When writing a logical one to the FOC0*x* bit, an immediate Compare Match is forced on the Waveform Generation unit. The OC0*x* output is changed according to its COM0*x*1:0 bits setting. Note that the FOC0*x* bit is implemented as a strobe. Therefore, it is the value present in the COM0*x*1:0 bits that determines the effect of the forced compare.

  A FOC0*x* strobe will not generate any interrupt nor will it clear the timer in CTC mode using OCR0A as TOP. The FOC0*x* bit is always read as zero.

- WGM02: Waveform Generation Mode. See the WGM01:0 description in register TCCR0A.

- CS02-0: Clock Select. The three Clock Select bits select the clock source to be used by the Timer/Counter as shown in Table 7.5.

  If external pin modes are used for the Timer/Counter0, transitions on the T0 pin will clock the counter even if the pin is configured as an output. This feature allows software control of the counting.

Table 7.5: Timer0 Clock Select Bit Description

CS02-0	Description
000	No clock source (Timer/Counter stopped)
001	$clk_{I/O}$/1 (No pre-scaling)
010	$clk_{I/O}$/8 (From pre-scaler)
011	$clk_{I/O}$/64 (From pre-scaler)
100	$clk_{I/O}$/256 (From pre-scaler)
101	$clk_{I/O}$/1024 (From pre-scaler)
110	External clock source on T0 pin. Clock on falling edge.
111	External clock source on T0 pin. Clock on rising edge.

7.3.3 TCNT0 - TIMER/COUNTER0 REGISTER

Bit	7	6	5	4	3	2	1	0
0x46	TCNT0[7:0]							
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

The Timer/Counter Register gives direct access, both for read and write operations, to the Timer/Counter unit 8-bit counter. Writing to the TCNT0 Register blocks (removes) the compare match on the following timer clock for all compare units. Modifying TCNT0 while the counter is running, introduces a risk of missing a compare match between TCNT0 and the OCR0*x* Registers.

7.3.4 OCR0A - OUTPUT COMPARE0 REGISTER A

Bit	7	6	5	4	3	2	1	0
0x47	OCR0A[7:0]							
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

The Output Compare Register A contains an 8-bit value that is continuously compared with the counter value TCNT0. A match can be used to generate an Output Compare interrupt or to generate a waveform output on the OC0A = PORTD6 pin.

7.3.5 OCR0B - OUTPUT COMPARE0 REGISTER B

Bit	7	6	5	4	3	2	1	0
0x48	OCR0B[7:0]							
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

The Output Compare Register B contains an 8-bit value that is continuously compared with the counter value TCNT0. A match can be used to generate an Output Compare interrupt or to generate a waveform output on the OC0B = PORTD5 pin.

7.3.6 TCCR1A - TIMER/COUNTER1 CONTROL REGISTER A

Bit	7	6	5	4	3	2	1	0
0x80	COM1A1	COM1A0	COM1B1	COM1B0	-	-	WGM11	WGM10
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W
Default	0	0	0	0	0	0	0	0

- COM1A1-0: Compare Output Mode for Channel A. These bits control the Output Compare pin (OC1A = PORTB1) behavior.

- COM1B1-0: Compare Output Mode for Channel B. These bits control the Output Compare pin (OC1B = PORTB2) behavior. If one or both of the COM1*x*1:0 bits are set, the OC1*x* output overrides the normal port functionality of the I/O pin it is connected to. However, note that the DDR bit corresponding to the OC1*x* pin must be set in order to enable the output driver.

  When OC1*x* is connected to the pin, the function of the COM1*x*1:0 bits depends on the WGM13:0 bit setting. Table 7.6 shows the COM1*x*1:0 bit functionality when the WGM13:0 bits are set to a normal or CTC mode (non-PWM).

Table 7.6: Timer1 Compare Output Mode, non-PWM Mode

COM1x1-0	Description
00	Normal port operation, OC1*x* disconnected
01	Toggle OC1*x* on Compare Match
10	Clear OC1*x* on Compare Match
11	Set OC1*x* on Compare Match

  Table 7.7 shows the COM1*x*1:0 bit functionality when the WGM13:0 bits are set to fast PWM mode.

Table 7.7: Timer1 Compare Output Mode, Fast PWM Mode

COM1x1-0	Description
00	Normal port operation, OC1*x* disconnected
01	WGM13:0 $\neq$ 14,15: Normal Port Operation, OC1*x* disconnected WGM13:0 = 14,15: Toggle OC1A on Compare Match, OC1B disconnected
10	Clear OC1*x* on Compare Match, set OC1*x* at 0x0000, (non-inverting mode).
11	Set OC1*x* on Compare Match, clear OC1*x* at 0x0000, (inverting mode).

  Table 7.8 shows the COM1*x*1:0 bit functionality when the WGM13:0 bits are set to phase correct PWM mode.

- WGM11-0: Waveform Generation Mode. Combined with the WGM13:2 bit found in the TCCR1B Register, these bits control the counting sequence of the counter, the source for maximum (TOP) counter value, and what type of waveform generation to be used, see Table 7.9. Modes of operation supported by the Timer/Counter unit are: Normal mode (counter), Clear Timer on Compare Match (CTC) mode, and three types of PWM modes.

Table 7.8: Timer1 Compare Output Mode, Phase Correct PWM Mode

COM1x1-0	Description
00	Normal port operation, OC1*x* disconnected
01	WGM13:0 ≠ 9,11: Normal Port Operation, OC1*x* disconnected WGM13:0 = 9,11: Toggle OC1A on Compare Match, OC1B disconnected
10	Clear OC1*x* on Compare Match when up-counting. Set OC1*x* on Compare Match when down-counting.
11	Set OC1*x* on Compare Match when up-counting. Clear OC1*x* on Compare Match when down-counting.

Table 7.9: Timer1 Waveform Generation Mode Bit Description

WGM13-0	Mode	TOP	OCR1x Update	TOV1 Flag Set
0000	Normal	0xFFFF	Immediate	0xFFFF
0001	Phase Correct 8-bit PWM	0x00FF	TOP	0x0000
0010	Phase Correct 9-bit PWM	0x01FF	TOP	0x0000
0011	Phase Correct 10-bit PWM	0x03FF	TOP	0x0000
0100	CTC	OCR1A	Immediate	0xFFFF
0101	Fast 8-bit PWM	0x00FF	0x0000	TOP
0110	Fast 9-bit PWM	0x01FF	0x0000	TOP
0111	Fast 10-bit PWM	0x03FF	0x0000	TOP
1000	Phase/Frequency Correct PWM	ICR1	0x0000	0x0000
1001	Phase/Frequency Correct PWM	OCR1A	0x0000	0x0000
1010	Phase Correct PWM	ICR1	TOP	0x0000
1011	Phase Correct PWM	OCR1A	TOP	0x0000
1100	CTC	ICR1	Immediate	0xFFFF
1101	Reserved	-	-	-
1110	Fast PWM	ICR1	0x0000	TOP
1111	Fast PWM	OCR1A	0x0000	TOP

7.3.7 TCCR1B - TIMER/COUNTER1 CONTROL REGISTER B

Bit	7	6	5	4	3	2	1	0
0x81	ICNC1	ICES1	-	WGM13	WGM12	CS12	CS11	CS10
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

- ICNC1: Input Capture Noise Canceler. Setting this bit (to one) activates the Input Capture Noise Canceler. When the noise canceler is activated, the input from the Input Capture pin

(ICP1 = PORTB0) is filtered. The filter function requires four successive equal valued samples of the ICP1 pin for changing its output. The Input Capture is therefore delayed by four Oscillator cycles when the noise canceler is enabled.

- ICES1: Input Capture Edge Select. This bit selects which edge on the Input Capture pin (ICP1) that is used to trigger a capture event. When the ICES1 bit is written to zero, a falling edge is used as trigger, and when the ICES1 bit is written to one, a rising edge will trigger the capture.

  When a capture is triggered according to the ICES1 setting, the counter value is copied into the Input Capture Register (ICR1). The event will also set the Input Capture Flag (ICF1), and this can be used to cause an Input Capture Interrupt, if this interrupt is enabled.

  When the ICR1 is used as TOP value, the ICP1 is disconnected and, consequently, the Input Capture function is disabled.

- WGM13-2: Waveform Generation Mode. See the WGM11:0 description in register TCCR1A.

- CS12-0: Clock Select. The three Clock Select bits select the clock source to be used by the Timer/Counter as shown in Table 7.10.

  If external pin modes are used for the Timer/Counter1, transitions on the T1 pin will clock the counter even if the pin is configured as an output. This feature allows software control of the counting.

Table 7.10: Timer1 Clock Select Bit Description

CS12-0	Description
000	No clock source (Timer/Counter stopped)
001	$clk_{I/O}$/1 (No pre-scaling)
010	$clk_{I/O}$/8 (From pre-scaler)
011	$clk_{I/O}$/64 (From pre-scaler)
100	$clk_{I/O}$/256 (From pre-scaler)
101	$clk_{I/O}$/1024 (From pre-scaler)
110	External clock source on T1 pin. Clock on falling edge.
111	External clock source on T1 pin. Clock on rising edge.

7.3.8 TCCR1C - TIMER/COUNTER1 CONTROL REGISTER C

Bit	7	6	5	4	3	2	1	0
0x82	FOC1A	FOC1B	-	-	-	-	-	-
Read/Write	W	W	R	R	R	R	R	R
Default	0	0	0	0	0	0	0	0

- FOC1A: Force Output Compare for Channel A.
- FOC1B: Force Output Compare for Channel B. The FOC1*x* bit is only active when the WGM13:0 bits specify a non-PWM mode. When writing a logical one to the FOC1*x* bit, an immediate Compare Match is forced on the Waveform Generation unit. The OC1*x* output is changed according to its COM1*x*1:0 bits setting. Note that the FOC1*x* bit is implemented as a strobe. Therefore, it is the value present in the COM1*x*1:0 bits that determines the effect of the forced compare.

  A FOC1*x* strobe will not generate any interrupt, nor will it clear the timer in CTC mode using OCR1A as TOP. The FOC1*x* bit is always read as zero.

7.3.9 TCNT1H AND TCNT1L - TIMER/COUNTER1 REGISTER

Bit	7	6	5	4	3	2	1	0
0x85	TCNT1[15:8]							
0x84	TCNT1[7:0]							
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

The two Timer/Counter Registers give direct access, both for read and write operations, to the Timer/Counter unit 16-bit counter. To ensure that both the high and low bytes are read and written simultaneously when the CPU accesses these registers, the access is performed using an 8-bit temporary High Byte Register (TEMP). This temporary register is shared by all the other 16-bit registers.

Writing to the TCNT1 Register blocks (removes) the compare match on the following timer clock for all compare units. Modifying TCNT1 while the counter is running, introduces a risk of missing a Compare Match between TCNT1 and the OCR1*x* Registers.

7.3.10 OCR1AH AND OCR1AL - OUTPUT COMPARE1 REGISTER A

Bit	7	6	5	4	3	2	1	0
0x89	OCR1A[15:8]							
0x88	OCR1A[7:0]							
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

The Output Compare Register A contains a 16-bit value that is continuously compared with the counter value TCNT1. A match can be used to generate an Output Compare interrupt or to generate a waveform output on the OC1A = PORTB1 pin.

The Output Compare Registers are 16-bit in size. To ensure that both the high and low bytes are written simultaneously when the CPU writes to these registers, the access is performed using

an 8-bit temporary High Byte Register (TEMP). This temporary register is shared by all the other 16-bit registers.

7.3.11 OCR1BH AND OCR1BL - OUTPUT COMPARE1 REGISTER B

Bit	7	6	5	4	3	2	1	0
0x8B	OCR1B[15:8]							
0x8A	OCR1B[7:0]							
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

The Output Compare Register B contains a 16-bit value that is continuously compared with the counter value TCNT1. A match can be used to generate an Output Compare interrupt or to generate a waveform output on the OC1B = PORTB2 pin.

The Output Compare Registers are 16-bit in size. To ensure that both the high and low bytes are written simultaneously when the CPU writes to these registers, the access is performed using an 8-bit temporary High Byte Register (TEMP). This temporary register is shared by all the other 16-bit registers.

7.3.12 ICR1H AND ICR1L - INPUT CAPTURE1 REGISTER

Bit	7	6	5	4	3	2	1	0
0x87	ICR1[15:8]							
0x86	ICR1[7:0]							
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

The Input Capture is updated with the counter TCNT1 value each time an event occurs on the ICP1 = PORTB0 pin (or optionally on the Analog Comparator output for Timer/Counter1). The Input Capture can be used for defining the counter TOP value.

The Output Compare Registers are 16-bit in size. To ensure that both the high and low bytes are written simultaneously when the CPU writes to these registers, the access is performed using an 8-bit temporary High Byte Register (TEMP). This temporary register is shared by all the other 16-bit registers.

7.3.13 TCCR2A - TIMER/COUNTER2 CONTROL REGISTER A

Bit	7	6	5	4	3	2	1	0
0xB0	COM2A1	COM2A0	COM2B1	COM2B0	-	-	WGM21	WGM20
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W
Default	0	0	0	0	0	0	0	0

- COM2A1-0: Compare Output Mode for Channel A. These bits control the Output Compare pin (`OC2A = PORTB3`) behavior.

- COM2B1-0: Compare Output Mode for Channel B. These bits control the Output Compare pin (`OC2B = PORTD3`) behavior. If one or both of the `COM2`*x*`1:0` bits are set, the `OC2`*x* output overrides the normal port functionality of the I/O pin it is connected to. However, note that the `DDR` bit corresponding to the `OC2`*x* pin must be set in order to enable the output driver.

  When `OC2`*x* is connected to the pin, the function of the `COM2`*x*`1:0` bits depends on the `WGM22:0` bit setting. Table 7.11 shows the `COM2`*x*`1:0` bit functionality when the `WGM22:0` bits are set to a normal or CTC mode (non-PWM).

Table 7.11: Timer2 Compare Output Mode, non-PWM Mode

COM2x1-0	Description
00	Normal port operation, `OC2`*x* disconnected
01	Toggle `OC2`*x* on Compare Match
10	Clear `OC2`*x* on Compare Match
11	Set `OC2`*x* on Compare Match

Table 7.12 shows the `COM2`*x*`1:0` bit functionality when the `WGM22:0` bits are set to fast PWM mode.

Table 7.12: Timer2 Compare Output Mode, Fast PWM Mode

COM2x1-0	Description
00	Normal port operation, `OC2`*x* disconnected
01	`WGM22 = 0`: Normal Port Operation, `OC2A` disconnected `OC2B` reserved behavior `WGM22 = 1`: Toggle `OC2A` on Compare Match, `OC2B` reserved behavior
10	Clear `OC2`*x* on Compare Match, set `OC2`*x* at 0x00, (non-inverting mode).
11	Set `OC2`*x* on Compare Match, clear `OC2`*x* at 0x00, (inverting mode).

Table 7.13 shows the `COM2`*x*`1:0` bit functionality when the `WGM22:0` bits are set to phase correct PWM mode.

- WGM21-0: Waveform Generation Mode. Combined with the `WGM22` bit found in the `TCCR2B` Register, these bits control the counting sequence of the counter, the source for maximum (TOP) counter value, and what type of waveform generation to be used, see Table 7.14.

Table 7.13: Timer2 Compare Output Mode, Phase Correct PWM Mode

COM2x1-0	Description
00	Normal port operation, OC2*x* disconnected
01	WGM22 = 0: Normal Port Operation, OC2A disconnected OC2B reserved behavior WGM22 = 1: Toggle OC2A on Compare Match, OC2B reserved behavior
10	Clear OC2*x* on Compare Match when up-counting. Set OC2*x* on Compare Match when down-counting.
11	Set OC2*x* on Compare Match when up-counting. Clear OC2*x* on Compare Match when down-counting.

Modes of operation supported by the Timer/Counter unit are: Normal mode (counter), Clear Timer on Compare Match (CTC) mode, and two types of PWM modes.

Table 7.14: Timer2 Waveform Generation Mode Bit Description

WGM22-0	Mode	TOP	OCR2x Update	TOV2 Flag Set
000	Normal	0xFF	Immediate	0xFF
001	Phase Correct PWM	0xFF	TOP	0x00
010	CTC	OCR2A	Immediate	0xFF
011	Fast PWM	0xFF	0x00	0xFF
100	Reserved	-	-	-
101	Phase Correct PWM	OCR2A	TOP	0x00
110	Reserved	-	-	-
111	Fast PWM	OCR2A	0x00	TOP

7.3.14 TCCR2B - TIMER/COUNTER2 CONTROL REGISTER B

Bit	7	6	5	4	3	2	1	0
0xB1	FOC2A	FOC2B	-	-	WGM22	CS22	CS21	CS20
Read/Write	W	W	R	R	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

- FOC2A: Force Output Compare for Channel A.
- FOC2B: Force Output Compare for Channel B. The FOC2*x* bit is only active when the WGM22:0 bits specify a non-PWM mode.

However, for ensuring compatibility with future devices, this bit must be set to zero when TCCR2B is written when operating in PWM mode. When writing a logical one to the FOC2*x* bit, an immediate compare match is forced on the waveform generation unit. The OC2*x* output is changed according to its COM2*x*1:0 bits setting. Note that the FOC2*x* bit is implemented as a strobe. Therefore, it is the value present in the COM2*x*1:0 bits that determines the effect of the forced compare.

A FOC2*x* strobe will not generate any interrupt, nor will it clear the timer in CTC mode using OCR2A as TOP. The FOC2*x* bit is always read as zero.

- WGM22: Waveform Generation Mode. See the WGM21:0 description in register TCCR2A.

- CS22-0: Clock Select. The three clock select bits select the clock source to be used by the Timer/Counter as shown in Table 7.15.

Table 7.15: Timer2 Clock Select Bit Description

CS22-0	Description
000	No clock source (Timer/Counter stopped)
001	clk_{T2S}/1 (No pre-scaling)
010	clk_{T2S}/8 (From pre-scaler)
011	clk_{T2S}/32 (From pre-scaler)
100	clk_{T2S}/64 (From pre-scaler)
101	clk_{T2S}/128 (From pre-scaler)
110	clk_{T2S}/256 (From pre-scaler)
111	clk_{T2S}/1024 (From pre-scaler)

7.3.15 TCNT2 - TIMER/COUNTER2 REGISTER

Bit	7	6	5	4	3	2	1	0
0xB2	TCNT2[7:0]							
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

The Timer/Counter Register gives direct access, both for read and write operations, to the Timer/Counter unit 8-bit counter. Writing to the TCNT2 Register blocks (removes) the compare match on the following timer clock for all compare units. Modifying TCNT2 while the counter is running, introduces a risk of missing a compare match between TCNT2 and the OCR2*x* Registers.

7.3.16 OCR2A - OUTPUT COMPARE2 REGISTER A

Bit	7	6	5	4	3	2	1	0
0xB3	OCR2A[7:0]							
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

The Output Compare Register A contains an 8-bit value that is continuously compared with the counter value TCNT2. A match can be used to generate an Output Compare interrupt or to generate a waveform output on the OC2A = PORTB3 pin.

7.3.17 OCR2B - OUTPUT COMPARE2 REGISTER B

Bit	7	6	5	4	3	2	1	0
0xB4	OCR2B[7:0]							
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

The Output Compare Register B contains an 8-bit value that is continuously compared with the counter value TCNT2. A match can be used to generate an Output Compare interrupt or to generate a waveform output on the OC2B = PORTD3 pin.

7.3.18 ASSR - ASYNCHRONOUS STATUS REGISTER

Bit	7	6	5	4	3	2	1	0
0xB6	-	EXCLK	AS2	TCN2UB	OCR2AUB	OCR2BUB	TCR2AUB	TCR2BUB
Read/Write	R	R/W	R/W	R	R	R	R	R
Default	0	0	0	0	0	0	0	0

- EXCLK: Enable External Clock Input. When EXCLK is written to one, and asynchronous clock is selected, the external clock input buffer is enabled and an external clock can be input on Timer Oscillator 1 (TOSC1) pin instead of a 32 kHz crystal. Writing to EXCLK should be done before asynchronous operation is selected. Note that the crystal oscillator will only run when this bit is zero.
- AS2: Asynchronous Timer/Counter2. When AS2 is written to zero, Timer/Counter2 is clocked from the I/O clock, $clk_{I/O}$. When AS2 is written to one, Timer/Counter2 is clocked from a crystal oscillator connected to the Timer Oscillator 1 (TOSC1) pin. When the value of AS2 is changed, the contents of TCNT2, OCR2A, OCR2B, TCCR2A and TCCR2B might be corrupted.
- TCN2UB: Timer/Counter2 Update Busy. When Timer/Counter2 operates asynchronously and TCNT2 is written, this bit becomes set. When TCNT2 has been updated from the temporary

storage register, this bit is cleared by hardware. A logical zero in this bit indicates that TCNT2 is ready to be updated with a new value.

- OCR2AUB: Output Compare Register 2 Channel A Update Busy. When Timer/Counter2 operates asynchronously and OCR2A is written, this bit becomes set. When OCR2A has been updated from the temporary storage register, this bit is cleared by hardware. A logical zero in this bit indicates that OCR2A is ready to be updated with a new value.

- OCR2BUB: Output Compare Register 2 Channel B Update Busy. When Timer/Counter2 operates asynchronously and OCR2B is written, this bit becomes set. When OCR2B has been updated from the temporary storage register, this bit is cleared by hardware. A logical zero in this bit indicates that OCR2B is ready to be updated with a new value.

- TCR2AUB: Timer/Counter Control Register 2 Channel A Update Busy. When Timer/-Counter2 operates asynchronously and TCCR2A is written, this bit becomes set. When TCCR2A has been updated from the temporary storage register, this bit is cleared by hardware. A logical zero in this bit indicates that TCCR2A is ready to be updated with a new value.

- TCR2BUB: Timer/Counter Control Register 2 Channel B Update Busy. When Timer/-Counter2 operates asynchronously and TCCR2B is written, this bit becomes set. When TCCR2B has been updated from the temporary storage register, this bit is cleared by hardware. A logical zero in this bit indicates that TCCR2B is ready to be updated with a new value.

7.3.19 GTCCR - GENERAL TIMER/COUNTER CONTROL REGISTER

Bit	7	6	5	4	3	2	1	0
0x43	TSM	-	-	-	-	-	PSRASY	PSRSYNC
Read/Write	R/W	R	R	R	R	R	R/W	R/W
Default	0	0	0	0	0	0	0	0

- TSM: Timer/Counter Synchronization Mode. Writing the TSM bit to one activates the Timer/-Counter Synchronization mode. In this mode, the value that is written to the PSRASY and PSRSYNC bits is kept, hence keeping the corresponding pre-scaler reset signals asserted. This ensures that the corresponding Timer/Counters are halted and can be configured to the same value without the risk of one of them advancing during configuration. When the TSM bit is written to zero, the PSRASY and PSRSYNC bits are cleared by hardware, and the Timer/Counters start counting simultaneously.

- PSRASY: Pre-scaler Reset Timer/Counter2 When this bit is one, the Timer/Counter2 pre-scaler will be reset. This bit is normally cleared immediately by hardware. If the bit is written when Timer/Counter2 is operating in asynchronous mode, the bit will remain one until the pre-scaler has been reset. The bit will not be cleared by hardware if the TSM bit is set.

- PSRSYNC: Pre-scaler Reset When this bit is one, Timer/Counter1 and Timer/Counter0 pre-scaler will be reset. This bit is normally cleared immediately by hardware, except if the `TSM` bit is set. Note that Timer/Counter1 and Timer/Counter0 share the same pre-scaler and a reset of this pre-scaler will affect both timers.

PROBLEMS

7.1 Determine the appropriate bit settings for `TCCR2A` and `TCCR2B` to generate a PWM output using the following specific details

- use channel B for the output waveform,
- use phase-correct PWM mode,
- use clear on up-counting, set on down-counting,
- have TOP defined as `OCR2A`,
- use a pre-scaler division value of 1.

7.2 Create a function that initializes timer 2 based on the values determined in problem 7.1. Additionally, set the `OCR2A` value to something that allows for a small-enough period such that the resulting analog output (after LPF) has a small amount of ripple. Note this last step will involve trial-and-error; a small value will allow a higher PWM frequency which will result in less ripple, but there will be fewer possible output levels; a large value will result in more ripple in an output signal.

7.3 Create a function that takes as an input a floating point number between 0 and 1. Based on the floating point value, the function needs to set the appropriate percent value in the `OCR2B` register. For example, if the input value is 0.6, then the count should be 60% of TOP.

7.4 Write a program that outputs a 2 kHz, 0 to 5 V saw-tooth PWM signal. Have the port pin output to a simple LPF where the f_h is well above 2 kHz (e.g., 16 kHz). Use an oscilloscope to verify your signal. Note that `DDR`*x* still has to be set to an output for the PWM port pin.

Tip: in order to generate a 2 kHz signal, you will have to repeatedly call the function you created in problem 7.3 with the appropriate analog value between 0 and 1, and with the appropriate amount of delay between samples. The `delay()` function is too slow to allow for a 2 kHz signal, but you can use either `micros()` to see how much time has passed from a previous call (like polling an input switch), or `delayMicroseconds()` to force a specific delay.

7.5 Write a program that monitors and debounces an input switch. As the user presses the switch, generate a 2 kHz sinusoid, a 2 kHz triangle wave, and a 2 kHz saw-tooth wave. Another press should turn off the output all together.

CHAPTER 8

Analog Input Ports

8.1 ANALOG-TO-DIGITAL CONVERTERS

An important aspect to most embedded systems involves one or more input signals. Up until now, any input messages that you have seen in previous chapters have been strictly digital. However, most naturally occurring signals are analog, meaning the voltage level can be any value within some interval $V_{min} \leq V_A \leq V_{max}$. In many systems, different actions and decisions are made based on the value of V_A. For example, consider the following list of analog sensor devices.

- Potentiometer - a mechanical knob is used to adjust the resistance between terminals;
- Thermistor - a device that changes resistance based on temperature;
- Accelerometer - a device that measures the acceleration of gravity in three dimensions and outputs an associated analog value for each dimension (e.g., this is the basis behind the Nintendo Wii video-game controllers);
- Ambient light sensor - a device that measure the environmental ambient light intensity and outputs an associated analog value (e.g., many laptop computers will adjust the backlight level based on the surrounding environment light level);
- Microphone - a device that converts vibrations (i.e., acoustic-waves) into analog levels.

This is only a small list of many devices that are necessary for bridging the analog environment in which we live to the digital world in which we perform computations. However, these device only provide the analog voltages and currents. In order for a microcontroller to operate on the sensor outputs, an Analog-to-Digital Conversion (*ADC*) circuit must be used, such as depicted in Fig. 8.1.

The parallel ADC shown in Fig. 8.1 uses typical values for V_{min} and V_{max} where the minimum voltage is simply set to ground and V_{max} is controlled externally via the user-defined reference voltage V_{REF}. The resistor network on the left-hand side is designed such that

$$V_{REF} - V_2 = V_2 - V_1 = V_1 - V_0 = V_0$$

which implies there is uniform spacing between the inverting inputs on all of the comparators. The analog voltage V_A is first converted into a three-bit codeword defined by $(c_2c_1c_0)$, where each bit is the output of a comparator. This can be seen in Fig. 8.2, where V_A is swept from 0 to V_{REF}. The four

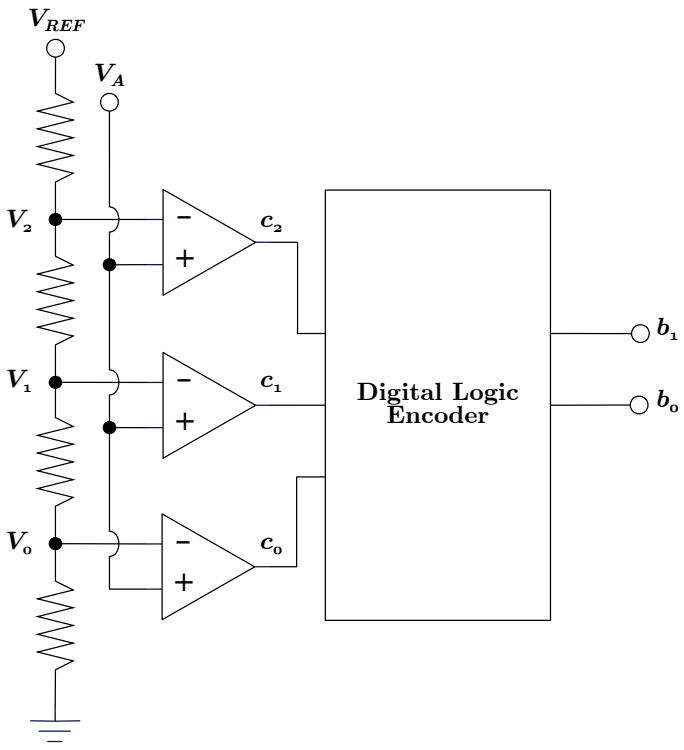


Figure 8.1: A two-bit parallel ADC.

possible codewords are then passed through an encoder that outputs the final two-bit value ($b_1 b_0$). Most microcontrollers provide at least one ADC, which will often output 10-bit resolutions for a very reasonable digital representation of the analog voltage. For example, with a reference voltage of 5 V, a 10-bit ADC provides an accuracy of $5/(2^{10}) = 4.9$ mV.

While the parallel ADC is very fast, it suffers from the problem of requiring $2^N - 1$ comparators and 2^N resistors to convert an analog signal into an N-bit digital representation. This circuit will require a lot of space and power in order to function, which are both drawbacks in an embedded processing environment. One possible solution is a counting ADC which is shown in Fig. 8.3.

The counting ADC functions by comparing the analog voltage V_A to a voltage level that is output from a Digital-to-Analog Converter (*DAC*) circuit. The level output from the DAC is controlled by the N-bit digital input vector ($b_{N-1}...b_0$). When the ADC process begins, the initial N-bit vector is cleared to (0...0) which will generate the lowest analog level from the DAC. As long as V_A is above the DAC output voltage, the comparator output c will be 5 V. This signal is used to

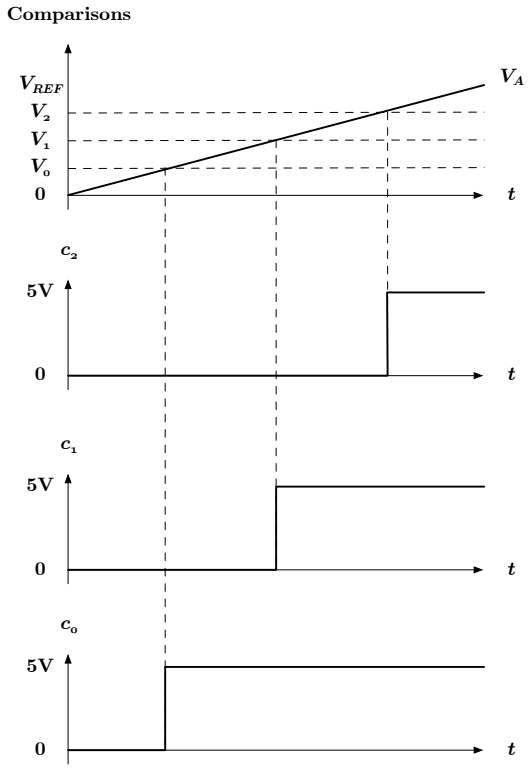


Figure 8.2: The initial ADC codewords defined by the output of individual comparators.

enable the counter to increase the N-bit vector by one when the next clock edge occurs. Eventually, the N-bit vector will increase to the point that the DAC output voltage is greater than V_A, at which time, c will go to 0, which will stop the counter. In this way, the N-bit value used to control the DAC output voltage is also the same value that represents V_A. The drawback of the counting ADC is the time necessary for the counter to lock onto the proper binary output vector. It is often the case that ADCs found in embedded processors do require some significant delay before the digital value can be trusted.

For the sake of completeness, and for anyone who might be curious, an N-bit DAC circuit may be realized as in Fig. 8.4. This particular topology is nice because only two different resistor values are used, R for all horizontally-drawn resistors and $2R$ for all vertically-drawn resistors. The R-$2R$ ladder is designed so that the amount of current entering each switch is a factor of two less than that of the switch immediately proceeding it. Then each b_i will control if that particular amount of current is passed to the feedback path or not. The result is 2^N different combinations of current

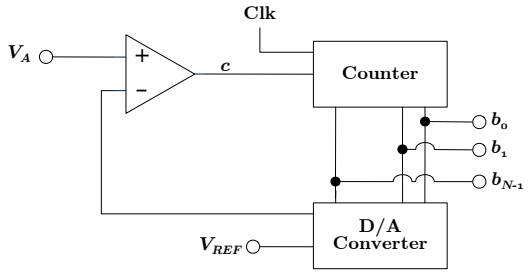


Figure 8.3: An N-bit counting ADC.

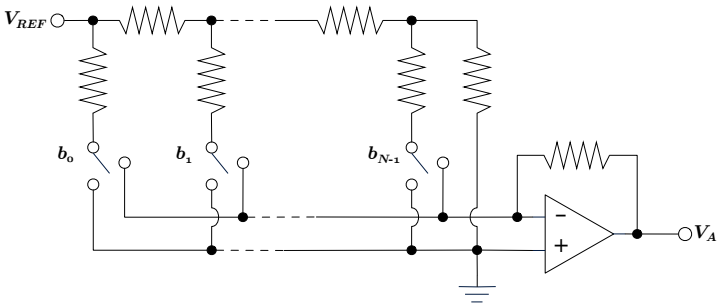


Figure 8.4: An N-bit R-2R ladder network DAC.

being sent to the feedback resistor, which provides 2^N different output voltages uniformly separated between 0 and $-V_{REF}$. So, there needs to be some additional circuitry (not shown) in order to invert the voltage so V_A is between 0 and V_{REF}.

8.1.1 ADC PERIPHERAL

In the case of the ATmega328P, there are six 10-bit ADCs (note: there are eight 10-bit ADCs on some packages, but not on the package present on the Arduino development board), all of which are found on Port C. The registers are listed in Sec. 8.3.

Note, in the `init()` function within the Wiring library, the ADC peripheral system is pre-initialized only so far as to set the pre-scalar and turn on the ADC functionality.

For generic usage of the ADC port functionality, the following steps should be addressed.

- Register `ADCSRA` needs to be enabled, started, auto trigger enabled, and an appropriate pre-scalar selected;

- register ADCSRB needs to have the auto trigger source set to free running, so the program can read the converted value of a low-frequency sensor (which is often our application);
- register ADMUX needs to have a V_{REF} source selected, right or left justification selected, and the desired ADC channel selected;
- register DIDR0 should have the input pin associated with the ADC channel selected in ADMUX disabled.

Once the ADC is initialized, the program can poll the ADC output by reading the 16-bit ADC register which contains the 10-bit result. As the math implies, eight bits sit in one 8-bit register and the remaining two bits are provided in the other register, with six bits being unused. It turns out that the ADC hardware will not update the ADC register until the ADCH portion is read out by itself, or the entire 16-bit register is read out as a single unit. So, if only eight bits of resolution are needed, the ADC value can be left-justified in ADMUX and the high-order byte may be read, as in the following code snip.

Example 8.1

```
unsigned char *portADCDataRegisterHigh;
unsigned char value;

portADCDataRegisterHigh = (unsigned char *) 0x79;

value = *portADCDataRegisterHigh;
```

However, if all 10 bits of resolution are necessary, then the two registers comprising the 16-bit space need to be accessed as in the following code snip (assuming left justification in ADMUX).

Example 8.2

```
unsigned short *portADCDataRegister;
unsigned short value;

portADCDataRegister = (unsigned short *) 0x78;

value = (*portADCDataRegister & 0xFFC0) >> 6;
```

Or, if right justification is used, the following would be more appropriate.

Example 8.3

```
unsigned short *portADCDataRegister;
unsigned short value;

portADCDataRegister = (unsigned short *) 0x78;

value = (*portADCDataRegister & 0x03FF);
```

8.2 ANALOG COMPARATOR

In addition to the ADC peripheral, the ATmega328P also provides a pure analog comparator peripheral. As shown in the simplified schematic in Fig. 8.5, two external input pins are connected to the two inputs of an op-amp configured as a comparator. As a result, software can use this peripheral in order to constantly monitor two analog signals. When the voltage on the positive terminal becomes greater than that of the negative terminal, the output signal is set. The comparator's output signal can be used to trigger the Timer1 input capture function, or else it can be used to generate an independent interrupt signal.

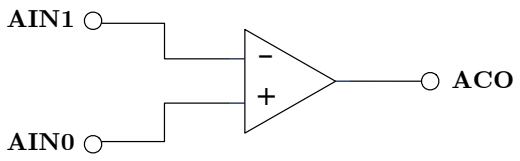


Figure 8.5: Simplified schematic of the analog comparator peripheral.

8.3 PERTINENT REGISTER DESCRIPTIONS

The information presented in this section was taken from ATMEL (2009).

8.3.1 ADMUX - ADC MULTIPLEXER SELECTION REGISTER

Bit	7	6	5	4	3	2	1	0
0x7C	REFS1	REFS0	ADLAR	-	MUX3	MUX2	MUX1	MUX0
Read/Write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

- REFS1-0: Reference Selection Bits. These bits select the voltage reference for the ADC, as shown in Table 8.1. If these bits are changed during a conversion, the change will not go in

effect until this conversion is complete (ADIF in ADCSRA is set). The internal voltage reference options may not be used if an external reference voltage is being applied to the AREF pin.

Table 8.1: Voltage Reference Selections for ADC

REFS1-0	Description
00	AREF, internal V_{ref} turned off
01	AV_{CC} with external capacitor at AREF pin Note: Arduino already has capacitor placed on line.
10	Reserved
11	Internal 1.1V reference with external capacitor at AREF pin Note: Arduino already has capacitor placed on line.

- ADLAR: ADC Left Adjust Result. The ADLAR bit affects the presentation of the ADC conversion result in the ADC Data Register. Write one to ADLAR to left adjust the result. Otherwise, the result is right adjusted. Changing the ADLAR bit will affect the ADC Data Register immediately, regardless of any ongoing conversions.

- MUX3-0: Analog Channel Selection Bits. The value of these bits selects which analog inputs are connected to the ADC. See Table 8.2 for details. If these bits are changed during a conversion, the change will not go in effect until this conversion is complete (ADIF in ADCSRA is set).

Table 8.2: Input Channel Selections

MUX3-0	Single Ended Input
0000	ADC0
0001	ADC1
0010	ADC2
0011	ADC3
0100	ADC4
0101	ADC5
0110	ADC6
0111	ADC7
1000	ADC8
1001-1101	Reserved
1110	1.1V
1111	GND

8.3.2 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
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

- 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.

- 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.

- ADATE: ADC Auto Trigger Enable. When this bit is written to one, Auto Triggering of the ADC is enabled. The ADC will start a conversion on a positive edge of the selected trigger signal. The trigger source is selected by setting the ADC trigger select bits, `ADTS` in `ADCSRB`.

- ADPS2-0: ADC Pre-scaler Select Bits. These bits determine the division factor between the system clock frequency and the input clock to the ADC.

Table 8.3: ADC Pre-scaler Selections

ADPS2-0	Division Factor
000	1
001	2
010	4
011	8
100	16
101	32
110	64
111	128

- See Ch. 9 for interrupt-related bit descriptions.

8.3.3 ADCH AND ADCL - ADC DATA REGISTER

8.3.3.1 ADLAR = 0

Bit	7	6	5	4	3	2	1	0
0x79	-	-	-	-	-	-	ADC9	ADC8
0x78	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0
Read/Write	R	R	R	R	R	R	R	R
Default	0	0	0	0	0	0	0	0

8.3.3.2 ADLAR = 1

Bit	7	6	5	4	3	2	1	0
0x79	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2
0x78	ADC1	ADC0	-	-	-	-	-	-
Read/Write	R	R	R	R	R	R	R	R
Default	0	0	0	0	0	0	0	0

When an ADC conversion is complete, the result is found in these two registers.

When ADCL is read, the ADC data register is not updated until ADCH is read. Consequently, if the result is left adjusted and no more than 8-bit precision is required, it is sufficient to read ADCH. Otherwise, ADCL must be read first, then ADCH.

The ADLAR bit in ADMUX, and the MUX*x* bits in ADMUX affect the way the result is read from the registers. If ADLAR is set, the result is left adjusted. If ADLAR is cleared (default), the result is right adjusted.

8.3.4 ADCSRB - ADC CONTROL AND STATUS REGISTER B

Bit	7	6	5	4	3	2	1	0
0x7B	-	ACME	-	-	-	ADTS2	ADTS1	ADTS0
Read/Write	R	R/W	R	R	R	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

- ACME: Analog Comparator Multiplexer Enable. When this bit is written logic one and the ADC is switched off (ADEN in ADCSRA is zero), the ADC multiplexer selects the negative input to the analog comparator. When this bit is written logic zero, AIN1 is applied to the negative input of the analog comparator.

- ADTS2-0: ADC Auto Trigger Source. If ADATE in ADCSRA is written to one, the value of these bits selects which source will trigger an ADC conversion. If ADATE is cleared, the ADTS2:0 settings will have no effect. A conversion will be triggered by the rising edge of the selected interrupt flag. Note that switching from a trigger source that is cleared to a trigger source that is set, will generate a positive edge on the trigger signal. If ADEN in ADCSRA is set, this will start

a conversion. Switching to free running mode, (ADTS2:0 = 0) will not cause a trigger event, even if the ADC interrupt flag is set.

Table 8.4: ADC Auto Trigger Source Selections

ADTS2-0	Trigger Source
000	Free Running mode
001	Analog Comparator
010	External Interrupt Request 0
011	Timer/Counter0 Compare Match A
100	Timer/Counter0 Overflow
101	Timer/Counter1 Compare Match B
110	Timer/Counter1 Overflow
111	Timer/Counter1 Capture Event

8.3.5 DIDR0 - DIGITAL INPUT DISABLE REGISTER 0

Bit	7	6	5	4	3	2	1	0
0x7E	-	-	ADC5D	ADC4D	ADC3D	ADC2D	ADC1D	ADC0D
Read/Write	R	R	R/W	R/W	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

- ADC5D-ADC0D: ADC5-0 Digital Input Disable. When this bit is written logic one, the digital input buffer on the corresponding ADC pin is disabled. The corresponding PIN Register bit will always read as zero when this bit is set. When an analog signal is applied to the ADC5–0 pin and the digital input from this pin is not needed, this bit should be written logic one to reduce power consumption in the digital input buffer.

  Note that ADC pins ADC7 and ADC6 do not have digital input buffers, and therefore do not require digital input disable bits.

8.3.6 ACSR - ANALOG COMPARATOR CONTROL AND STATUS REGISTER

Bit	7	6	5	4	3	2	1	0
0x50	ACD	ACBG	ACO	ACIF	ACIE	ACIC	ACIS1	ACIS0
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W
Default	0	0	-	0	0	0	0	0

- ACD: Analog Comparator Disable. When this bit is written logic one, the power to the Analog Comparator is switched off. This bit can be set at any time to turn off the Analog Comparator. This will reduce power consumption in Active and Idle mode. When changing

the ACD bit, the Analog Comparator Interrupt must be disabled by clearing the ACIE bit in ACSR. Otherwise, an interrupt can occur when the bit is changed.

- ACBG: Analog Comparator Bandgap Select. When this bit is set, a fixed bandgap reference voltage replaces the positive input to the Analog Comparator. When this bit is cleared, AIN0 is applied to the positive input of the Analog Comparator. When the bandgap referance is used as input to the Analog Comparator, it will take a certain time for the voltage to stabilize. If not stabilized, the first conversion may give a wrong value.
- ACO: Analog Comparator Output. The output of the Analog Comparator is synchronized and then directly connected to ACO. The synchronization introduces a delay of 1 - 2 clock cycles.
- ACIC: Analog Comparator Input Capture Enable. When written logic one, this bit enables the input capture function in Timer/Counter1 to be triggered by the Analog Comparator. The comparator output is in this case directly connected to the input capture front-end logic, making the comparator utilize the noise canceler and edge select features of the Timer/Counter1 Input Capture interrupt. When written logic zero, no connection between the Analog Comparator and the input capture function exists. To make the comparator trigger the Timer/Counter1 Input Capture interrupt, the ICIE1 bit in the Timer Interrupt Mask Register (TIMSK1) must be set.
- See Ch. 9 for interrupt-related bit descriptions.

8.3.7 DIDR1 - DIGITAL INPUT DISABLE REGISTER 1

Bit	7	6	5	4	3	2	1	0
0x7F	-	-	-	-	-	-	AIN1D	AIN0D
Read/Write	R	R	R	R	R	R	R/W	R/W
Default	0	0	0	0	0	0	0	0

- AIN1D-AIN0D: AIN1-0 Digital Input Disable. When this bit is written logic one, the digital input buffer on the corresponding AIN1/0 pin is disabled. The corresponding PIN Register bit will always read as zero when this bit is set. When an analog signal is applied to the AIN1/0 pin and the digital input from this pin is not needed, this bit should be written logic one to reduce power consumption in the digital input buffer.

PROBLEMS

8.1 Determine the appropriate bit settings for ADCSRA, ADCSRB, ADMUX,and DIDR0 to read an analog input using the following specific details.

– use a pre-scale division of 16,

- use the internal 5 V V_{REF},
- use right justification,
- use ADC channel 5.

8.2 Create a function that initializes the ADC sub-system based on the values determined in problem 8.1.

8.3 Use a trim-pot (or any other reasonable analog sensing device) as the input to ADC channel 5. For a simple example, see Fig. 8.6.

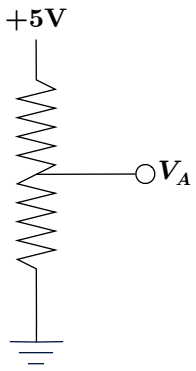


Figure 8.6: Schematic of a three-terminal trim-pot (i.e., potentiometer).

8.4 Write a program that polls ADC channel 5 for the analog value of the sensor set up in problem 8.3. Based on the value, create a series of LED patterns on a 7-segment display, and adjust the rate at which the pattern changes based on the analog input value.

8.5 Write a program that polls ADC channel 5 for the analog value of the sensor set up in problem 8.3. Based on the value, adjust the PWM duty-cycle of a PWM output pin; the PWM output could be connected to an LED or a DC motor.

8.6 Write a program that polls ADC channel 5 for the analog value of the sensor set up in problem 8.3. Based on the value, adjust the frequency of a PWM-generated sinusoid of a PWM output pin; the PWM output could be connected to a speaker.

8.7 Write a program that polls ADC channel 5 for the analog value of the sensor set up in problem 8.3. Based on the value, adjust the frequency of the various PWM signals from problem 7.5.

CHAPTER 9

Interrupt Processing

9.1 INTRODUCTION

All processing performed in previous chapters has been completely deterministic. For example, any time a program needed to retrieve an input signal, it did so by repeatedly checking the desired information source, a process called *polling*. This concept is shown in Fig. 9.1. Each block in the

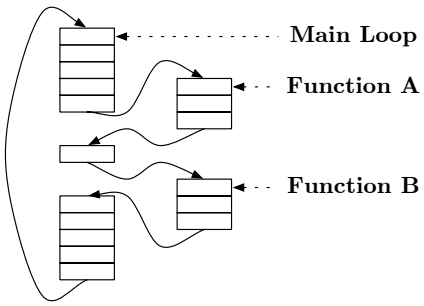


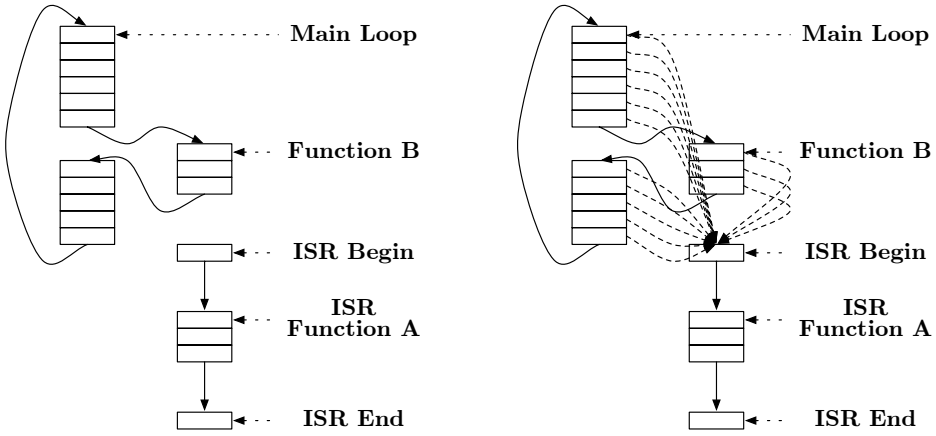
Figure 9.1: A conceptual view of software polling.

figure is meant to indicate a single instruction, and the solid directional lines show the path that the Program Counter (PC) follows. As can be seen in this example, the program executes **Main Loop** which makes two different function calls, one to **Function A** and one to **Function B**. We have already been able to create software that follows this topology in previous chapter problems.

While polling works fine, there are a couple of reasons to avoid it. First, performance can suffer in more complicated embedded systems which require input from many sources. For example, a high-tier cell-phone has to monitor an entire key-pad, menu buttons, ambient light sensor, QWERTY keyboard, etc. When software is written such that the processor is constantly polling all inputs for activity, it will never have a chance to perform other operations that generally control the behavior of the device. Second, an important feature for many embedded systems is to periodically power-down hardware, a process called sleeping. When a processor sleeps, it is able to conserve power, which leads to longer portable battery lives. Many environments that involve polling don't allow the microprocessor to sleep at all, resulting in batteries draining much faster.

An alternative version to the polling example is shown in Fig. 9.2(a). **Function A** has been moved from the deterministic pathway of the polling software to an isolated function that gets called

in response to an interrupt condition. Interrupts are signals used to notify the CPU that some new event has just occurred. Because most interrupt sources occur outside the CPU boundary, interrupts



(a) Program with ISR (b) ISR called from anywhere

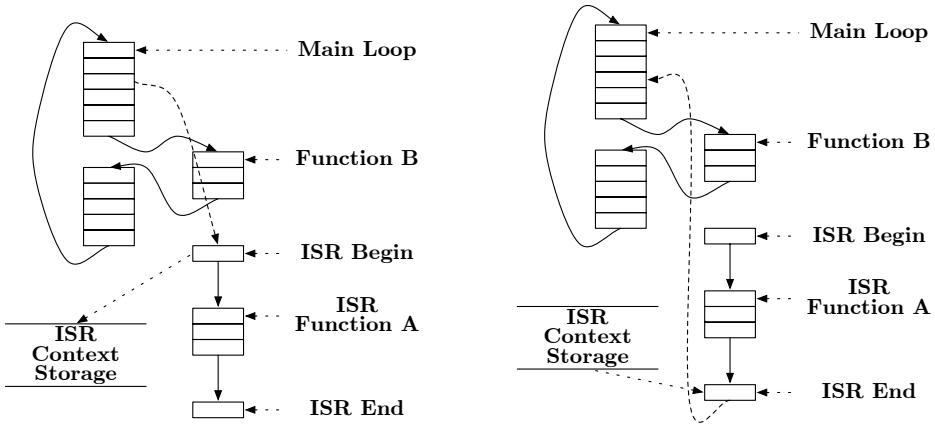
Figure 9.2: A conceptual view of software containing an ISR. The interrupt signal may occur at any time during the primary algorithm's processing path.

may be thought of as random signals that can occur at any point during the otherwise predictable flow of the primary algorithm. When an interrupt occurs, the CPU stops whatever it is doing and jumps to an associated Interrupt Service Routine (*ISR*). An ISR is a special function written to handle the fact that the interrupt signal occurred. This process is depicted in Fig. 9.2(b). In the polling example, **Function A** always occurred at the same time within the loop of processing. By contrast, the new example algorithm has moved **Function A** processing outside the main algorithm's path, reducing the work constantly performed by the main program. The drawback is the added complexity of the fact that now **Function A** can be called at any time, and so software needs to be written with this behavior in mind.

9.1.1 CONTEXT

The software *context* may be defined as the CPU environment as seen by each assembly instruction. Typically, the context may be specified by the set of CPU registers, including the PC which points to the current assembly instruction. For the example interrupt-based algorithm to work properly, it is certainly necessary that **Main Loop** function as it did in the polling case. This requires the current context to appear as though each assembly instruction is executed in the intended sequence. Interrupt processing is made possible by saving the context on ISR entry and restoring the context when the ISR completes. For example, suppose an interrupt occurs at the third (machine) instruction

of **Main Loop**, as shown in Fig. 9.3(a). As a result, the address of the fourth (machine) instruction is



(a) ISR is called (b) ISR finishes

Figure 9.3: A conceptual view of software containing an ISR. The interrupt signal occurs at the third instruction of the primary algorithm's processing path.

stored as part of the current context. Then, any CPU registers used within **Function A** are also saved in memory before they are overwritten by **Function A**'s instructions. After **Function A** finishes, the entire context is restored and the PC is reset to the fourth instruction of the main program, as shown in Fig. 9.3(b). The CPU begins executing the fourth instruction of **Main Loop** with CPU registers appearing as though the third instruction just finished because the context was saved prior to entering the ISR and then restored after the ISR completed.

9.1.2 ISR AND MAIN TASK COMMUNICATION

The final general topic that needs attention is the method for an ISR to communicate with the main program. That is, the ISR is not a direct function call, so there are no parameters that can be passed in, nor is there a return statement that is able to pass back any values. Therefore, the only way for an ISR to communicate with the rest of the program is to use shared memory, such as global variables. However, because an interrupt can occur at any time, care must be taken when reading or writing variables that are accessed within ISRs. As seen in Fig. 9.4, the reason is because individual C instructions are often composed of several machine instructions, especially when dealing with 16- and 32-bit variables. So, it is likely the main program will be interrupted part-way through variable access. The associated ISR could change the contents of the variable before it returns to the previous location. At that time, the main program would continue accessing the variable, which is now different. The result is that the main program operates on a variable value that is half current

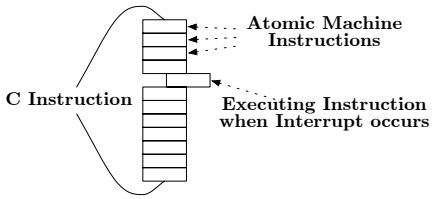


Figure 9.4: A single C instruction is composed of several machine instructions. This example shows an interrupt signal occurring at the fifth machine instruction.

and half incorrect. In order to mitigate these problems, any memory that is shared between ISRs and the main program need to be protected. The protection necessary is usually in the form of turning off global interrupts before accessing the shared variable. When finished, the global interrupts need to be restored to their previous state.

9.1.3 ATMEGA328P INTERRUPTS IN C

A great amount of care must be taken when writing ISR functions in assembly due to the context management. The same care must be taken when writing ISR functions in C. However, most compilers provide special macros or functions that cause the compiler to handle most of the context management. As a result, interrupt-based software does not look all that different from polling software.

In the case of the AVR-GCC compiler for the ATmega328P, there is a set of predefined function names that correspond to every interrupt signal that can occur on the processor. When an algorithm needs to handle an interrupt, all that is required is to write the desired function using the appropriate function name. The entire list of interrupts and their associated function names are presented in Table 9.1.

Aside from the specific ISR function name, there is a set of macros defined in the header file `avr/interrupt.h` that are used to tell the compiler how to manage certain functions. `ISR()` is the most basic macro used to register and mark a function as an interrupt handler. The macro is defined to take the vector function name as its first parameter followed by optional attributes including `ISR_BLOCK`, `ISR_NOBLOCK`, `ISR_NAKED` and `ISR_ALIASOF(vect)`. Each attribute provides the following:

- `ISR_BLOCK` - (default behavior) identical to an ISR with no attributes specified. Global interrupts are disabled by the ATmega328P hardware when entering the ISR, without the compiler modifying this state.

Table 9.1: ATmega328P Interrupts

Pri.	Address	Interrupt Source	ISR C Function Name	Description
1	0x0000	RESET		System reset (power-on)
2	0x0002	INT0	`INT0_vect`	External Interrupt Request 0
3	0x0004	INT1	`INT1_vect`	External Interrupt Request 1
4	0x0006	PCINT0	`PCINT0_vect`	Pin Change Interrupt Request 0
5	0x0008	PCINT1	`PCINT1_vect`	Pin Change Interrupt Request 1
6	0x000A	PCINT2	`PCINT2_vect`	Pin Change Interrupt Request 2
7	0x000C	WDT	`WDT_vect`	Watchdog Time-out Interrupt
8	0x000E	TIMER2 COMPA	`TIMER2_COMPA_vect`	Timer/Counter2 Compare Match A
9	0x0010	TIMER2 COMPB	`TIMER2_COMPB_vect`	Timer/Counter2 Compare Match B
10	0x0012	TIMER2 OVF	`TIMER2_OVF_vect`	Timer/Counter2 Overflow
11	0x0014	TIMER1 CAPT	`TIMER1_CAPT_vect`	Timer/Counter1 Capture Event
12	0x0016	TIMER1 COMPA	`TIMER1_COMPA_vect`	Timer/Counter1 Compare Match A
13	0x0018	TIMER1 COMPB	`TIMER1_COMPB_vect`	Timer/Counter1 Compare Match B
14	0x001A	TIMER1 OVF	`TIMER1_OVF_vect`	Timer/Counter1 Overflow
15	0x001C	TIMER0 COMPA	`TIMER0_COMPA_vect`	Timer/Counter0 Compare Match A
16	0x001E	TIMER0 COMPB	`TIMER0_COMPB_vect`	Timer/Counter0 Compare Match B
17	0x0020	TIMER0 OVF	`TIMER0_OVF_vect`	Timer/Counter0 Overflow
18	0x0022	SPI, STC	`SPI_STC_vect`	SPI Serial Transfer Complete
19	0x0024	USART, RX	`USART_RX_vect`	USART Receive Complete
20	0x0026	USART, UDRE	`USART_UDRE_vect`	USART Data Register Empty
21	0x0028	USART, TX	`USART_TX_vect`	USART Transmit Complete
22	0x002A	ADC	`ADC_vect`	ADC Conversion Complete
23	0x002C	EE READY	`EE_READY_vect`	EEPROM Ready
24	0x002E	ANALOG COMP	`ANALOG_COMP_vect`	Analog Comparator
25	0x0030	TWI	`TWI_vect`	2-wire Serial Interface
26	0x0032	SPM READY	`SPM_READY_vect`	Store Program Memory Ready

- `ISR_NOBLOCK` - Global interrupts are reenabled by the compiler-generated code when the ISR is initially entered. This may be used to allow nested ISRs, meaning an ISR may be interrupted by a higher-priority interrupt.
- `ISR_NAKED` - The compiler does not generate any context management code. The ISR function has to explicitly save and restore the context, including adding the `reti` instruction at the end of the ISR to return the PC to is prior location.
- `ISR_ALIASOF(vect)` - The ISR is linked to the ISR specified by the `vect` parameter. This allows a single ISR function to handle multiple interrupt signals.

As an example, suppose we would like to enable the interrupt for Timer1 overflow. Once the timer is set up (not shown), the interrupt is enabled with the following.

Example 9.1

```
#define TIMSK1_ADDR                (unsigned char *) 0x6F

#define TIMSK1_ICIE_MASK           0x20
#define TIMSK1_OCIEB_MASK          0x04
#define TIMSK1_OCIEA_MASK          0x02
#define TIMSK1_TOIE_MASK           0x01

unsigned long g_timerCount;

void InitializeTimer (void)
{
   unsigned char *portTimerCounterInterruptMaskRegister;
   unsigned char shadow;

   portTimerCounterInterruptMaskRegister = TIMSK1_ADDR;

   shadow = *portTimerCounterInterruptMaskRegister;
   shadow &= ~(TIMSK1_ICIE_MASK |
               TIMSK1_OCIEB_MASK |
               TIMSK1_OCIEA_MASK |
               TIMSK1_TOIE_MASK);
   shadow |= TIMSK1_TOIE_MASK;
   *portTimerCounterInterruptMaskRegister = shadow;

   g_timerCount = 0;
}
```

This code piece clears the four different interrupt bits before setting the single bit indicating the desire to receive interrupt signals whenever the overflow occurs in Timer1. The ISR may be something like the following.

Example 9.2

```
ISR(TIMER1_OVF_vect)
{
   g_timerCount++;
}
```

It is a rule-of-thumb not to perform extensive processing inside an ISR so as not to starve the main program. For example, function calls should not be made from within an ISR. Usually an ISR

contains a few instructions that might set a global variable to indicate to the main program that the signal has occurred. In this example, a global variable is incremented whenever the overflow occurs.

To complete the example, the main program needs to do something with the global variable set by the ISR. Because this is shared memory between the main program and the ISR, protection needs to be placed around the variable access to prevent the ISR from corrupting the value in the memory location. The protection is realized by turning off global interrupts before accessing the variable. When finished, the global interrupts need to be restored to their previous state. This is managed via the I-bit in the SREG register such as in the following example.

Example 9.3

```
#define SREG_ADDR                   (unsigned char *) 0x5F

#define SREG_GLOBAL_INT_ENABLE      0x80

void loop()
{
   unsigned char *statusRegister;
   unsigned char shadow;
   unsigned long timerCountCopy;

   /*
      Save the current global interrupt bit value in shadow.
   */
   statusRegister = SREG_ADDR;
   shadow = *statusRegister;

   /*
      Disable global interrupts.
   */
   *statusRegister &= ~SREG_GLOBAL_INT_ENABLE;

   /* access g_timerCount here */
   timerCountCopy = g_timerCount;

   /*
      Restore the global interrupt bit to previous value.
   */
   if ((shadow & SREG_GLOBAL_INT_ENABLE) == SREG_GLOBAL_INT_ENABLE)
   {
      *statusRegister |= SREG_GLOBAL_INT_ENABLE;
   }

   /* Now do stuff on the copy of the g_timerCount variable */
}
```

9.2 PERTINENT REGISTER DESCRIPTIONS

The information presented in this section was taken from ATMEL (2009).

9.2.1 EICRA - EXTERNAL INTERRUPT CONTROL REGISTER A

Bit	7	6	5	4	3	2	1	0
0x69	-	-	-	-	ISC11	ISC10	ISC01	ISC00
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

- ISC11-0: Interrupt Sense Control 1 Bit 1 and Bit 0.
- ISC01-0: Interrupt Sense Control 0 Bit 1 and Bit 0. The External Interrupt x is activated by the external pin INTx if the SREG I-flag and the corresponding interrupt mask are set. The level and edges on the external INTx pin that activate the interrupt are defined in Table 9.1. The value on the INTx pin is sampled before detecting edges. If edge or toggle interrupt is selected, pulses that last longer than one clock period will generate an interrupt. Shorter pulses are not guaranteed to generate an interrupt. If low level interrupt is selected, the low level must be held until the completion of the currently executing instruction to generate an interrupt.

Table 9.2: Interrupt x Sense Control

ISCx1-0	Description
00	The low level of INTx generates an interrupt request.
01	Any logical change on INTx generates an interrupt request.
10	The falling edge of INTx generates an interrupt request.
11	The rising edge of INTx generates an interrupt request.

9.2.2 EIMSK - EXTERNAL INTERRUPT MASK REGISTER

Bit	7	6	5	4	3	2	1	0
0x3D	-	-	-	-	-	-	INT1	INT0
Read/Write	R	R	R	R	R	R	R/W	R/W
Default	0	0	0	0	0	0	0	0

- INT1: External Interrupt Request 1 Enable.
- INT0: External Interrupt Request 0 Enable. When the INTx bit is set (one) and the I-bit in the Status Register (SREG) is set (one), the external pin interrupt is enabled. The Interrupt Sense Control bits 1/0 (ISCx1:0) in the External Interrupt Control Register A (EICRA) define whether the external interrupt is activated on the rising and/or falling edge of the INTx pin or level sensed. Activity on the pin will cause an interrupt request even if INTx is configured

as an output. The corresponding interrupt of External Interrupt Request *x* is executed from the INT*x* Interrupt Vector.

9.2.3 EIFR - EXTERNAL INTERRUPT FLAG REGISTER

Bit	7	6	5	4	3	2	1	0
0x3C	-	-	-	-	-	-	INTF1	INTF0
Read/Write	R	R	R	R	R	R	R/W	R/W
Default	0	0	0	0	0	0	0	0

- INTF1: External Interrupt Flag 1.
- INTF0: External Interrupt Flag 0. When an edge or logic change on the INT*x* pin triggers an interrupt request, INTF*x* becomes set (one). If the I-bit in SREG and the INTF*x* bit in EIMSK are set (one), the MCU will jump to the corresponding Interrupt Vector. The flag is cleared when the interrupt routine is executed. Alternatively, the flag can be cleared by writing a logical one to it. This flag is always cleared when INTF*x* is configured as a level interrupt.

9.2.4 PCICR - PIN CHANGE INTERRUPT CONTROL REGISTER

Bit	7	6	5	4	3	2	1	0
0x68	-	-	-	-	-	PCIE2	PCIE1	PCIE0
Read/Write	R	R	R	R	R	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

- PCIE2: Pin Change Interrupt Enable 2. This bit affects PCINT23:16 (in the following, $y \in \{16, ..., 23\}$).
- PCIE1: Pin Change Interrupt Enable 1. This bit affects PCINT14:8 (in the following, $y \in \{8, ..., 14\}$).
- PCIE0: Pin Change Interrupt Enable 0. This bit affects PCINT7:0 (in the following, $y \in \{0, ..., 7\}$). When the PCIE*x* bit is set (one) and the I-bit in the Status Register (SREG) is set (one), pin change interrupt *x* is enabled. Any change on any enabled PCINT*y* pin will cause an interrupt. The corresponding interrupt of Pin Change Interrupt Request is executed from the PCI*x* Interrupt Vector. PCINT*y* pins are enabled individually by the PCMSK*x* Register.

9.2.5 PCIFR - PIN CHANGE INTERRUPT FLAG REGISTER

Bit	7	6	5	4	3	2	1	0
0x3B	-	-	-	-	-	PCIF2	PCIF1	PCIF0
Read/Write	R	R	R	R	R	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

- PCIF2: Pin Change Interrupt Flag 2. This bit affects PCINT23:16 (in the following, $y \in \{16, ..., 23\}$).
- PCIF1: Pin Change Interrupt Flag 1. This bit affects PCINT14:8 (in the following, $y \in \{8, ..., 14\}$).
- PCIF0: Pin Change Interrupt Flag 0. This bit affects PCINT7:0 (in the following, $y \in \{0, ..., 7\}$). When a logic change on any PCINT*y* pin triggers an interrupt request, PCIF*x* becomes set (one). If the I-bit in SREG and the PCIE*x* bit in PCICR are set (one), the MCU will jump to the corresponding Interrupt Vector. The flag is cleared when the interrupt routine is executed. Alternatively, the flag can be cleared by writing a logical one to it.

9.2.6 PCMSK2 - PIN CHANGE MASK REGISTER 2

Bit	7	6	5	4	3	2	1	0
0x6D	PCINT23	PCINT22	PCINT21	PCINT20	PCINT19	PCINT18	PCINT17	PCINT16
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

- PCINT23-16: Pin Change Enable Mask 23-16. Each PCINT23:16-bit selects whether pin change interrupt is enabled on the corresponding I/O pin. If PCINT23:16 is set and the PCIE2 bit in PCICR is set, pin change interrupt is enabled on the corresponding I/O pin. If PCINT23:16 is cleared, pin change interrupt on the corresponding I/O pin is disabled.

9.2.7 PCMSK1 - PIN CHANGE MASK REGISTER 1

Bit	7	6	5	4	3	2	1	0
0x6C	-	PCINT14	PCINT13	PCINT12	PCINT11	PCINT10	PCINT9	PCINT8
Read/Write	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

- PCINT14-8: Pin Change Enable Mask 14-8. Each PCINT14:8-bit selects whether pin change interrupt is enabled on the corresponding I/O pin. If PCINT14:8 is set and the PCIE1 bit in PCICR is set, pin change interrupt is enabled on the corresponding I/O pin. If PCINT14:8 is cleared, pin change interrupt on the corresponding I/O pin is disabled.

9.2.8 PCMSK0 - PIN CHANGE MASK REGISTER 0

Bit	7	6	5	4	3	2	1	0
0x6B	PCINT7	PCINT6	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

- PCINT7-0: Pin Change Enable Mask 7-0. Each PCINT7:0-bit selects whether pin change interrupt is enabled on the corresponding I/O pin. If PCINT7:0 is set and the PCIE0 bit in PCICR is set, pin change interrupt is enabled on the corresponding I/O pin. If PCINT7:0 is cleared, pin change interrupt on the corresponding I/O pin is disabled.

9.2.9 TIMSK0 - TIMER/COUNTER0 INTERRUPT MASK REGISTER

Bit	7	6	5	4	3	2	1	0
0x6E	-	-	-	-	-	OCIE0B	OCIE0A	TOIE0
Read/Write	R	R	R	R	R	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

- OCIE0B: Timer/Counter0 Output Compare Match Channel B Interrupt Enable. When the OCIE0B bit is written to one, and the I-bit in the Status Register (SREG) is set, the Timer/-Counter0 Compare Match B interrupt is enabled. The corresponding interrupt is executed if a Compare Match in Timer/Counter0 on channel B occurs, i.e., when the OCF0B bit is set in the Timer/Counter0 Interrupt Flag Register (TIFR0).

- OCIE0A: Timer/Counter0 Output Compare Match Channel A Interrupt Enable. When the OCIE0A bit is written to one, and the I-bit in the Status Register (SREG) is set, the Timer/Counter0 Compare Match A interrupt is enabled. The corresponding interrupt is executed if a Compare Match in Timer/Counter0 on channel A occurs, i.e., when the OCF0A bit is set in the Timer/Counter0 Interrupt Flag Register (TIFR0).

- TOIE0: Timer/Counter0 Overflow Interrupt Enable. When the TOIE0 bit is written to one, and the I-bit in the Status Register (SREG) is set, the Timer/Counter0 Overflow interrupt is enabled. The corresponding interrupt is executed if an overflow in Timer/Counter0 occurs, i.e., when the TOV0 bit is set in the Timer/Counter0 Interrupt Flag Register (TIFR0).

9.2.10 TIFR0 - TIMER/COUNTER0 INTERRUPT FLAG REGISTER

Bit	7	6	5	4	3	2	1	0
0x35	-	-	-	-	-	OCF0B	OCF0A	TOV0
Read/Write	R	R	R	R	R	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

- OCF0B: Timer/Counter0 Output Compare Match Channel B Flag. The OCF0B bit is set when a Compare Match occurs between the Timer/Counter0 and the data in OCR0B. OCF0B is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, OCF0B is cleared by writing a logic one to the flag. When the I-bit in SREG, OCIE0B, and OCF0B are set, the Timer/Counter0 Compare Match B Interrupt is executed.

- OCF0A: Timer/Counter0 Output Compare Match Channel A Flag. The OCF0A bit is set when a Compare Match occurs between the Timer/Counter0 and the data in OCR0A. OCF0A is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, OCF0A is cleared by writing a logic one to the flag. When the I-bit in SREG, OCIE0A, and OCF0A are set, the Timer/Counter0 Compare Match A Interrupt is executed.

- TOV0: Timer/Counter0 Overflow Flag. The TOV0 bit is set when an overflow occurs in Timer/Counter0. TOV0 is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, TOV0 is cleared by writing a logic one to the flag. When the I-bit in SREG, TOIE0, and TOV0 are set, the Timer/Counter0 Overflow interrupt is executed.

9.2.11 TIMSK1 - TIMER/COUNTER1 INTERRUPT MASK REGISTER

Bit	7	6	5	4	3	2	1	0
0x6F	-	-	ICIE1	-	-	OCIE1B	OCIE1A	TOIE1
Read/Write	R	R	R/W	R	R	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

- ICIE1: Timer/Counter1 Input Capture Interrupt Enable. When the ICIE1 bit is written to one, and the I-bit in the Status Register (SREG) is set, the Timer/Counter1 Input Capture interrupt is enabled. The corresponding interrupt is executed if an Input Capture occurs, i.e., when the ICF1 bit is set in the Timer/Counter1 Interrupt Flag Register (TIFR1).

- OCIE1B: Timer/Counter1 Output Compare Match Channel B Interrupt Enable. When the OCIE1B bit is written to one, and the I-bit in the Status Register (SREG) is set, the Timer/-Counter1 Compare Match B interrupt is enabled. The corresponding interrupt is executed if a Compare Match in Timer/Counter1 on channel B occurs, i.e., when the OCF1B bit is set in the Timer/Counter1 Interrupt Flag Register (TIFR1).

- OCIE1A: Timer/Counter1 Output Compare Match Channel A Interrupt Enable. When the OCIE1A bit is written to one, and the I-bit in the Status Register (SREG) is set, the Timer/Counter1 Compare Match A interrupt is enabled. The corresponding interrupt is executed if a Compare Match in Timer/Counter1 on channel A occurs, i.e., when the OCF1A bit is set in the Timer/Counter1 Interrupt Flag Register (TIFR1).

- TOIE1: Timer/Counter1 Overflow Interrupt Enable. When the TOIE1 bit is written to one, and the I-bit in the Status Register (SREG) is set, the Timer/Counter1 Overflow interrupt is enabled. The corresponding interrupt is executed if an overflow in Timer/Counter1 occurs, i.e., when the TOV1 bit is set in the Timer/Counter1 Interrupt Flag Register (TIFR1).

9.2.12 TIFR1 - TIMER/COUNTER1 INTERRUPT FLAG REGISTER

Bit	7	6	5	4	3	2	1	0
0x36	-	-	ICF1	-	-	OCF1B	OCF1A	TOV1
Read/Write	R	R	R/W	R	R	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

- ICF1: Timer/Counter1 Input Capture Flag. The `ICF1` bit is set when a Capture event occurs on the `ICP1` pin. When the Input Capture Register (`ICR1`) is set by the `WGM13:0` to be used as the TOP value, the `ICF1` flag is set when the counter reaches the TOP value. `ICF1` is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, `ICF1` is cleared by writing a logic one to the flag. When the `I`-bit in `SREG`, `ICIE1`, and `ICF1` are set, the Timer/Counter1 Capture Event Interrupt is executed.
- OCF1B: Timer/Counter1 Output Compare Match Channel B Flag. The `OCF1B` bit is set when a Compare Match occurs between the Timer/Counter1 and the data in `OCR1B`. `OCF1B` is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, `OCF1B` is cleared by writing a logic one to the flag. When the `I`-bit in `SREG`, `OCIE1B`, and `OCF1B` are set, the Timer/Counter1 Compare Match B Interrupt is executed.
- OCF1A: Timer/Counter1 Output Compare Match Channel A Flag. The `OCF1A` bit is set when a Compare Match occurs between the Timer/Counter1 and the data in `OCR1A`. `OCF1A` is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, `OCF1A` is cleared by writing a logic one to the flag. When the `I`-bit in `SREG`, `OCIE1A`, and `OCF1A` are set, the Timer/Counter1 Compare Match A Interrupt is executed.
- TOV1: Timer/Counter1 Overflow Flag. The `TOV1` bit is set when an overflow occurs in Timer/Counter1. `TOV1` is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, `TOV1` is cleared by writing a logic one to the flag. When the `I`-bit in `SREG`, `TOIE1`, and `TOV1` are set, the Timer/Counter1 Overflow interrupt is executed.

9.2.13 TIMSK2 - TIMER/COUNTER2 INTERRUPT MASK REGISTER

Bit	7	6	5	4	3	2	1	0
0x70	-	-	-	-	-	OCIE2B	OCIE2A	TOIE2
Read/Write	R	R	R	R	R	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

- OCIE2B: Timer/Counter2 Output Compare Match Channel B Interrupt Enable. When the `OCIE2B` bit is written to one, and the `I`-bit in the Status Register (`SREG`) is set, the Timer/-Counter2 Compare Match B interrupt is enabled. The corresponding interrupt is executed if a Compare Match in Timer/Counter2 on channel B occurs, i.e., when the `OCF2B` bit is set in the Timer/Counter2 Interrupt Flag Register (`TIFR2`).

- OCIE2A: Timer/Counter2 Output Compare Match Channel A Interrupt Enable. When the OCIE2A bit is written to one, and the I-bit in the Status Register (SREG) is set, the Timer/Counter2 Compare Match A interrupt is enabled. The corresponding interrupt is executed if a Compare Match in Timer/Counter2 on channel A occurs, i.e., when the OCF2A bit is set in the Timer/Counter2 Interrupt Flag Register (TIFR2).

- TOIE2: Timer/Counter2 Overflow Interrupt Enable. When the TOIE2 bit is written to one, and the I-bit in the Status Register (SREG) is set, the Timer/Counter2 Overflow interrupt is enabled. The corresponding interrupt is executed if an overflow in Timer/Counter2 occurs, i.e., when the TOV2 bit is set in the Timer/Counter2 Interrupt Flag Register (TIFR2).

9.2.14 TIFR2 - TIMER/COUNTER2 INTERRUPT FLAG REGISTER

Bit	7	6	5	4	3	2	1	0
0x37	-	-	-	-	-	OCF2B	OCF2A	TOV2
Read/Write	R	R	R	R	R	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

- OCF2B: Timer/Counter2 Output Compare Match Channel B Flag. The OCF2B bit is set when a Compare Match occurs between the Timer/Counter2 and the data in OCR2B. OCF2B is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, OCF2B is cleared by writing a logic one to the flag. When the I-bit in SREG, OCIE2B, and OCF2B are set, the Timer/Counter2 Compare Match B Interrupt is executed.

- OCF2A: Timer/Counter2 Output Compare Match Channel A Flag. The OCF2A bit is set when a Compare Match occurs between the Timer/Counter2 and the data in OCR2A. OCF2A is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, OCF2A is cleared by writing a logic one to the flag. When the I-bit in SREG, OCIE2A, and OCF2A are set, the Timer/Counter2 Compare Match A Interrupt is executed.

- TOV2: Timer/Counter2 Overflow Flag. The TOV2 bit is set when an overflow occurs in Timer/Counter2. TOV2 is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, TOV2 is cleared by writing a logic one to the flag. When the I-bit in SREG, TOIE2, and TOV2 are set, the Timer/Counter2 Overflow interrupt is executed. In PWM mode, this bit is set when Timer/Counter2 changes counting direction at 0x00.

9.2.15 SPCR - SPI CONTROL REGISTER

Bit	7	6	5	4	3	2	1	0
0x4C	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

- SPIE: SPI Interrupt Enable. This bit causes the SPI interrupt to be executed if SPIF bit in the SPSR Register is set and the if the I-bit in the Status Register (SREG) is set.

- See Ch. 10 for all other bit descriptions.

9.2.16 SPSR - SPI STATUS REGISTER

Bit	7	6	5	4	3	2	1	0
0x4D	SPIF	WCOL	-	-	-	-	-	SPI2X
Read/Write	R	R	R	R	R	R	R	R/W
Default	0	0	0	0	0	0	0	0

- SPIF: SPI Interrupt Flag. When a serial transfer is complete, the SPIF Flag is set. An interrupt is generated if SPIE in SPCR is set and global interrupts are enabled. If SS is an input and is driven low when the SPI is in Master mode, this will also set the SPIF Flag. SPIF is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, the SPIF bit is cleared by first reading the SPI Status Register with SPIF set, then accessing the SPI Data Register (SPDR).

- See Ch. 10 for all other bit descriptions.

9.2.17 UCSR0A - USART0 CONTROL AND STATUS REGISTER A

Bit	7	6	5	4	3	2	1	0
0xC0	RXC0	TXC0	UDRE0	FE0	DOR0	UPE0	U2X0	MPCM0
Read/Write	R	R/W	R	R	R	R	R/W	R/W
Default	0	0	1	0	0	0	0	0

- RXC0: USART0 Receive Complete. This flag bit is set when there are unread data in the receive buffer and cleared when the receive buffer is empty (i.e., does not contain any unread data). If the Receiver is disabled, the receive buffer will be flushed and, consequently, the RXC0 bit will become zero. The RXC0 Flag can be used to generate a Receive Complete interrupt (see description of the RXCIE0 bit).

- TXC0: USART0 Transmit Complete. This flag bit is set when the entire frame in the Transmit Shift Register has been shifted out and there are no new data currently present in the transmit buffer (UDR0). The TXC0 Flag bit is automatically cleared when a transmit complete interrupt is executed, or it can be cleared by writing a one to its bit location. The TXC0 Flag can generate a Transmit Complete interrupt (see description of the TXCIE0 bit).

- UDRE0: USART0 Data Register Empty. The UDRE0 Flag indicates if the transmit buffer (UDR0) is ready to receive new data. If UDRE0 is one, the buffer is empty, and therefore ready

to be written. The UDRE0 Flag can generate a Data Register Empty interrupt (see description of the UDRIE0 bit). UDRE0 is set after a reset to indicate that the Transmitter is ready.

- See Ch. 10 for all other bit descriptions.

9.2.18 UCSR0B - USART0 CONTROL AND STATUS REGISTER B

Bit	7	6	5	4	3	2	1	0
0xC1	RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	UCSZ02	RXB80	TXB80
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W
Default	0	0	0	0	0	0	0	0

- RXCIE0: USART0 Receive Complete Interrupt Enable. Writing this bit to one enables interrupt on the RXC0 Flag. A USART Receive Complete interrupt will be generated only if the RXCIE0 bit is written to one, the I-bit in SREG is written to one and the RXC0 bit in UCSR0A is set.

- TXCIE0: USART0 Transmit Complete Interrupt Enable. Writing this bit to one enables interrupt on the TXC0 Flag. A USART Transmit Complete interrupt will be generated only if the TXCIE0 bit is written to one, the I-bit in SREG is written to one and the TXC0 bit in UCSR0A is set.

- UDRIE0: USART0 Data Register Empty Interrupt Enable. Writing this bit to one enables interrupt on the UDRE0 Flag. A USART Data Register Empty interrupt will be generated only if the UDRIE0 bit is written to one, the I-bit in SREG is written to one and the UDRE0 bit in UCSR0A is set.

- See Ch. 10 for all other bit descriptions.

9.2.19 TWCR - TWI CONTROL REGISTER

Bit	7	6	5	4	3	2	1	0
0xBC	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE
Read/Write	R/W	R/W	R/W	R/W	R	R/W	R	R/W
Default	0	0	0	0	0	0	0	0

- TWINT: TWI Interrupt Flag. This bit is set by hardware when the TWI has finished its current job and expects application software response. If the I-bit in SREG and TWIE in TWCR are set, the MCU will jump to the TWI Interrupt Vector. While the TWINT Flag is set, the SCL low period is stretched. The TWINT Flag must be cleared by software by writing a logic one to it. Note that this flag is not automatically cleared by hardware when executing the interrupt routine. Also note that clearing this flag starts the operation of the TWI, so all accesses to the

TWI Address Register (TWAR), TWI Status Register (TWSR), and TWI Data Register (TWDR) must be complete before clearing this flag.

- TWIE: TWI Interrupt Enable. When this bit is written to one, and the I-bit in SREG is set, the TWI interrupt request will be activated for as long as the TWINT Flag is high.
- See Ch. 10 for all other bit descriptions.

9.2.20 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
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

- ADIF: ADC Interrupt Flag. This bit is set when an ADC conversion completes and the data registers are updated. The ADC conversion complete interrupt is executed if the ADIE bit and the I-bit in SREG are set. ADIF is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, ADIF is cleared by writing a logical one to the flag. Beware that if doing a Read-Modify-Write on ADCSRA, a pending interrupt can be disabled. This also applies if the SBI and CBI instructions are used.
- ADIE: ADC Interrupt Enable. When this bit is written to one and the I-bit in SREG is set, the ADC conversion complete interrupt is activated.
- See Ch. 8 for all other bit descriptions.

9.2.21 ACSR - ANALOG COMPARATOR CONTROL AND STATUS REGISTER

Bit	7	6	5	4	3	2	1	0
0x50	ACD	ACBG	ACO	ACIF	ACIE	ACIC	ACIS1	ACIS0
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W
Default	0	0	-	0	0	0	0	0

- ACIF: Analog Comparator Interrupt Flag. This bit is set by hardware when a comparator output event triggers the interrupt mode defined by ACIS1:0. The Analog Comparator interrupt routine is executed if the ACIE bit is set and the I-bit in SREG is set. ACIF is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, ACIF is cleared by writing a logic one to the flag.
- ACIE: Analog Comparator Interrupt Enable. When this bit is written to one and the I-bit in SREG is set, the Analog Comparator interrupt is activated. When written logic zero, the interrupt is disabled.

- ACIS1-0: Analog Comparator Interrupt Mode Select. These bits determine which comparator events that trigger the Analog Comparator interrupt. The different settings are shown in Table 9.3.

  When changing the `ACIS1:0` bits, the Analog Comparator Interrupt must be disabled by clearing its Interrupt Enable bit in the `ACSR` Register. Otherwise, an interrupt can occur when the bits are changed.

Table 9.3: Analog Comparator Interrupt Settings

ACIS1-0	Description
00	Comparator Interrupt on Output Toggle.
01	Reserved
10	Comparator Interrupt on Falling Output Edge.
11	Comparator Interrupt on Rising Output Edge.

- See Ch. 8 for all other bit descriptions.

9.2.22 EECR - EEPROM CONTROL REGISTER

Bit	7	6	5	4	3	2	1	0
0x3F	-	-	EEPM1	EEPM0	EERIE	EEMPE	EEPE	EERE
Read/Write	R	R	R/W	R/W	R/W	R/W	R/W	R/W
Default	0	0	-	-	0	0	-	0

- EERIE: EEPROM Ready Interrupt Enable. Writing `EERIE` to one enables the EEPROM Ready Interrupt if the `I`-bit in `SREG` is set. Writing `EERIE` to zero disables the interrupt. The EEPROM Ready interrupt generates a constant interrupt when `EEPE` is cleared. The interrupt will not be generated during EEPROM write or SPM.

- See Ch. 12 for all other bit descriptions.

PROBLEMS

9.1 Determine the appropriate bit settings for `TCCR1A`, `TCCR1B`, `OCR1A`,and `TIMSK1` using the following specific details:

- use Clear Timer on Compare (CTC) Match mode,
- disconnect both OC1 pins,
- use a pre-scale division of 1024,
- use the Output Compare A Match interrupt,
- use an OCRA value such that you receive 1 interrupt per second.

9.2 Create a function that initializes timer 1 based on the values determined in problem 9.1.

9.3 Create an ISR for timer 1.

9.4 Determine the appropriate bit settings for `PCICR` and `PCMSK2` using the following specific details:

- use Port D4 as an input pin,
- disable the internal pull-up resistor,
- using Port D4 corresponds to Pin Change interrupt 20.

9.5 Create a function that initializes the Pin Change interrupt based on the values determined in problem 9.4.

9.6 Create an ISR for Pin Change interrupt 20.

9.7 Create a program that utilizes the timer interrupt and the pin change interrupt in order to measure the frequency of a periodic signal on Port Pin D4. You can do this by saving and then clearing a counter each time the timer interrupt occurs (at a rate of once per second). Additionally, you can increment the counter by the number of low-to-high and high-to-low transitions on pin D4 with the pin change interrupt. Your main loop should monitor when the timer interrupt goes off, and then write the frequency to the serial port for the user to see. Note: your setup function should call your two initialization functions, and then enable global interrupts in the `SREG` register.

9.8 Connect a controllable signal generator source to port pin D4. Warning: you should calibrate the signal before connecting it to the Arduino. Do so by connecting the signal generator to an oscilloscope and verify you have a 0-to-5V source. Be sure to note the frequency.

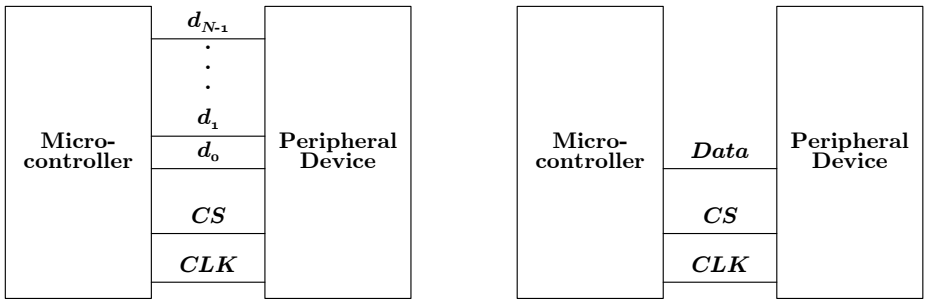
Use the program created in problem 9.7. Compare your output with that determined by the oscilloscope. Note: don't forget the pin change interrupt occurs on both rising and falling edges, so you will need to account for that in your output to the user.

CHAPTER 10

Serial Communications

10.1 INTRODUCTION

Many embedded systems include peripheral devices connected to the microprocessor in order to expand its capabilities. In previous chapters, we have seen a few examples of peripherals including the seven-segment display, H-bridge and DC motor, potentiometer and even a simple push-button switch. All of these peripherals have interfaced with the microcontroller via a custom protocol across direct parallel port connections. A *protocol* is the language that governs communications between systems or devices. Protocols may specify many aspects of inter-device communications including bit ordering, bit-pattern meanings, electrical connections, and even mechanical considerations in some cases. Communication methods can be divided into the two categories shown in Fig. 10.1, parallel and serial.



(a) Parallel (b) Serial

Figure 10.1: Communication methods between a microcontroller and a peripheral device.

Consider the familiar example of controlling a seven-segment display peripheral via a parallel interface. Such a program would use seven GPIO output lines (i.e., $\{d_0, \ldots, d_6\}$) to light up desired patterns based on various bit-strings. These bit-strings and their respective output patterns form the protocol between the microcontroller and the display. In this case, the display responds immediately to the change on the lines because the peripheral is completely passive; i.e., there is no controlling mechanism in place within the display. Many peripherals are complex enough to include their own embedded controller, in which case the protocol needs to be more sophisticated. In particular, the

microcontroller is considered the master while the embedded peripheral controller is considered the slave. The master device drives communications by controlling the chip select CS line(s) and the synchronous clock CLK line. Any slave peripheral connected to the communication bus will wait until the master issues a signal, at which time the slave will perform the indicated function. Common examples of sophisticated peripherals that requires a parallel data bus are liquid crystal displays (LCDs) which contain hundreds or thousands of pixels. For a real example, consider some of the beautiful consumer electronic devices that are able to display high quality real-time video. In order to do so, they use a high data-rate throughput which requires a parallel data bus.

Most peripherals don't have the requirement of large data-rate transfers, but they still need a method for sending and receiving bytes of information. An alternative to sending N bits at the same time across N data lines is to transfer N bits one-at-a-time across a single data line as implied in Fig. 10.1(b). Doing so reduces the number of external pins required by each peripheral's package thereby reducing chip sizes. This form of communication is called serial because each byte of information is used to create a series of bits, which is then transmitted and received along the data line. For example, suppose the microcontroller needs to send the eight-bit value `0x53` to the peripheral. Instead of using eight lines, the value is used to create the series {0, 1, 0, 1, 0, 0, 1, 1}. Then the CLK line is used by the microcontroller to tell the peripheral when the next bit in the series is ready to be sampled. The peripheral will shift in each bit when a CLK event occurs until all eight bits have been received. It will then convert the series of bits back into the eight-bit byte. Notice there are many questions that arise with this form of communication including:

- In what order are the series of bits shifted across the data line? Suppose the master transmits the most-significant bit (MSB) first, then the peripheral will receive the series in the order {0, 1, 0, 1, 0, 0, 1, 1}. Alternatively, the master could transmit the least-significant bit (LSB) first; in which case, the peripheral will receive the series {1, 1, 0, 0, 1, 0, 1, 0}. Either method is fine, but the peripheral and master must agree before hand; otherwise, incorrect bytes will be received.

- What constitutes a CLK event? The master could use either a falling-edge or a rising-edge clock to specify a sampling signal to the peripheral device.

- How does the peripheral know when it is supposed to receive bytes and when it is supposed to transmit bytes?

All of these questions and many more are answered by the protocol. The two protocols I^2C (pronounced "eye-squared see") and SPI (pronounced "spy") have become very common for a wide-variety of peripheral integrated circuits (ICs) including EEPROM non-volatile memory, audio amplifiers, temperature sensors, H-bridge chips, and accelerometers. Many times, when embedded systems are designed with proprietary CPLDs or FPGAs, they will include one of these two serial protocols in order to interact with the microcontroller. The following two sections present an overview of each protocol including specific information regarding the ATmega328P.

10.1.1 INTER-INTEGRATED CIRCUIT

The Inter-Integrated Circuit (I^2C or IIC) serial protocol was created by NXP Semiconductors, originally a Philips semiconductor division, to attach low-speed peripherals to an embedded microprocessor as shown in Figure 10.2. I^2C is a multi-point protocol in which more than two devices are

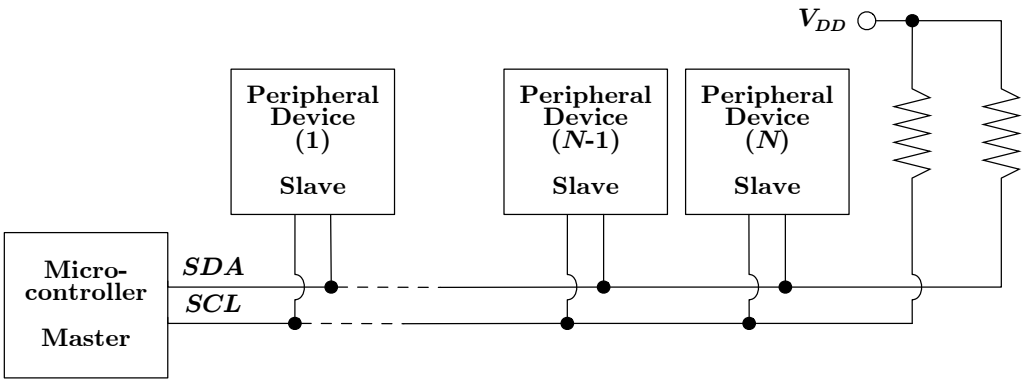


Figure 10.2: I^2C serial communication method between a microcontroller master and one or more slave peripheral devices.

able to communicate along the serial interface which is composed of a bidirectional serial data line (SDA) and a bidirectional serial clock (SCL). As a result, each line is configured as open-drain which means that each must be pulled-up to V_{DD} via an external resistor. Exactly one device on the bus is considered the master of the serial bus and all other devices are slaves. At the start of any serial transaction, each slave listens to the bus for its own specific address. The master begins communication by transmitting a single start bit followed by the unique 7-bit address of the slave device for which the master is attempting to access, followed by read/write bit. The corresponding slave device responds with an acknowledge bit if it is present on the serial bus. The master continues to issue clock events on the SCL line and either receives information from the slave or writes information to the slave depending on the read/write bit at the start of the session. The number of bits transferred during a single session is dependent upon the peripheral device and the agreed-upon higher-level protocol. Notice that any device writing to SDA only needs to pull the line low for each '0' written; otherwise, is leaves the line alone to write a '1', which occurs due to the lines being pulled high externally. Any manufacturer that includes the fully compatible protocol within their peripheral chip must pay a fee in order to have an I^2C slave address reserved by NXP, although the protocol itself is available free of charge. Some peripheral devices will provide a few external pins in order to "program" the I^2C slave address via pull-up and pull-down resistors. In this way, an embedded system can have multiple instances of the same component; for example, multiple temperature sensors from the same

manufacturer may co-exist on the same serial bus by programming each one with a unique address using pull-ups and pull-downs.

The ATmega328P provides an I^2C serial interface via the 2-wire Serial Interface (*TWI*). The bus allows for up to 128 different slave devices and up to 400 kHz data transfer speed. The TWI provides an interrupt-based system in which an interrupt is issued after all bus events such as reception of a byte or transmission of a start condition. You will see how to manage a serial port that utilizes interrupts later in the problems. The relevant TWI registers are listed in Sec. 10.2. Additionally, SDA and SCL use Port C pins 4 and 5 when configured to do so.

10.1.2 SERIAL PERIPHERAL INTERFACE

The Serial Peripheral Interface (SPI) serial protocol was created by Motorola to allow peripherals to communicate in full duplex mode as shown in Fig. 10.3. *Full duplex* means that two devices transmit

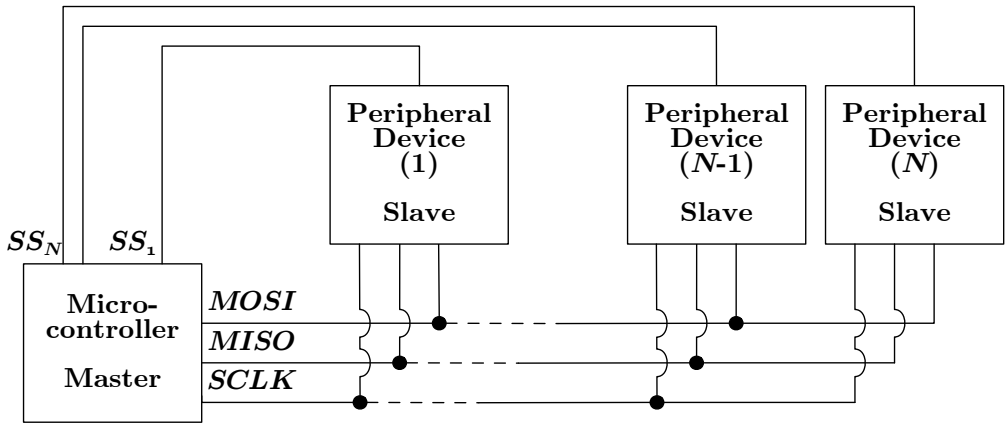


Figure 10.3: SPI serial communication method between a microcontroller master and one or more slave peripheral devices.

and receive data simultaneously (e.g., telephone communications), as opposed to *half duplex* which means that two devices transmit and receive data by taking turns (e.g., public-safety radio communications). SPI is another multi-point protocol in which devices communicate via the serial interface composed of serial clock ($SCLK$), master-out/slave-in ($MOSI$), master-in/slave-out ($MISO$) and slave-select (SS). Exactly one device on the bus is considered the master of the serial bus and all other devices are slaves. The master controls communication by asserting the SS_i line of the ith slave device and then issues clock events on the $SCLK$ line. The master simultaneously receives information from the slave on $MISO$ and writes information to the slave on $MOSI$. The meaning

and number of bits transferred in both directions is dependent upon the peripheral device and the agreed-upon higher-level protocol.

The ATmega328P provides a SPI serial interface that allows for multiple peripheral device connections. The SPI provides an interrupt-based system in which an interrupt is issued after each byte has been transmitted/received. When the ATmega328P processor is defined as the master, the program is responsible for managing each device's slave select line prior to starting the communication process. The relevant SPI registers are listed in Sec. 10.2. Additionally, $\overline{SS}$, $MOSI$, $MISO$ and SCK use Port B pins 2 through 5 when configured to do so. Note that $\overline{SS}$ is only used when the ATmega328P is configured to be a slave device, it is not automatically controlled during master operation.

10.1.3 UNIVERSAL ASYNCHRONOUS RECEIVER/TRANSMITTER

Another common serial communications sub-circuit present on most microprocessors is a Universal Asynchronous Receiver/Transmitter (UART), which is really just a fancy shift-register allowing for the transformation between parallel and serial forms. UARTs provide both a receive (RX) and a transmit (TX) line, such as shown in Fig. 10.4. UARTs are the physical layer upon which differ-

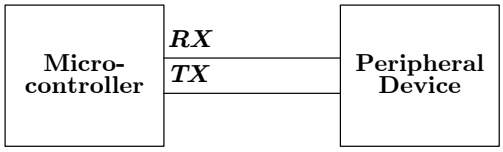


Figure 10.4: UART serial communication method between a microcontroller and a peripheral device.

ent serial protocols operate including the multi-point Recommended Standard 485 (RS-485) and the point-to-point RS-232 governed by the Electronic Industries Alliance (EIA). These common standards specify many aspects to the serial interface including voltage levels, connectors, pinouts, cable-lengths, bit orderings, and bit rates among others. One important difference in using a UART versus I^2C or SPI is the absence of a clock line to perform synchronization. Because there is no common clock signal, each device must search for a start bit by sampling its RX line with an internal clock. When the start bit is identified, the receiving device knows that a transmission has begun and can shift in the series of bits. Both transmitter and receiver must agree upon a common bit rate before communication begins; otherwise, the receiver will decode incorrect data.

10.1.4 USART ON ATMEGA328P

The ATmega328P provides a Universal Synchronous/Asynchronous Receiver/Transmitter (USART) that is configurable to allow for RS-232 communications. Further, the Arduino development

platform is designed such that the RX and TX lines connect to the FTDI USB-to-UART IC so that software can be downloaded to the platform. In other words, you have already been making use of the ATmega328P UART in terms of development as well as all of the `Serial`-class function calls for printing debugging information to the host computer's serial terminal. The bootloader present as part of the Arduino package configures the UART when power is applied to allow the host platform the ability to download new programs via USB. Because the USART is already connected to a peripheral device (the USB-to-UART converter chip), we will study how to manage a serial port in software by configuring the USART to perform RS-232 communications. The relevant USART0 registers are listed in Sec. 10.2. Aside from the Arduino serial peripheral connection, the RXD, TXD and XCK signals use Port D pins 0, 1 and 4 when configured to do so. Note that XCK is a clock signal used in the synchronous mode of the USART hardware.

It is interesting that the ATmega328P uses the same physical address for both the inbound and the outbound USART data registers (`UDR0`). The hardware is actually designed with two physical registers where CPU access controls which register is written-to or read-from via the single data-register address.

10.1.5 INTERRUPT-BASED SERIAL PORT MANAGEMENT IN C

It is very common for microprocessors to provide interrupt signals indicating when a single byte has been received or has completed transmission. In either case, the associated ISR must move quickly to make room for the next incoming or outgoing byte. Additionally, the software needs to provide enough allocated memory to hold the inbound and outbound messages. The amount of space necessary is defined by the application-level protocol. This means the engineer(s) responsible for defining the messages between the microcontroller and the peripheral should have a good idea what the largest single message will be and allocate the appropriate space accordingly. First, consider the steps necessary for managing the serial port receiver using three global variables.

Example 10.1

```
#define USART_RECEIVE_BUFFER_SIZE    80

unsigned char g_receiveBuffer[USART_RECEIVE_BUFFER_SIZE];
unsigned char g_receiveHead;
unsigned char g_receiveTail;
```

The `g_receiveBuffer` is the chunk of memory reserved to store the incoming data stream, and `g_receiveHead` and `g_receiveTail` are two indices used to write/read to/from the buffer. When used properly, these three variables form a circular buffer. As shown in Fig. 10.5, a circular buffer is an array treated as though the final element is succeeded by the first element. For example, pretend the elements of the array are laid out in a circle such that the final entry is sitting next to the first entry.

The following steps are used to manage the buffer.

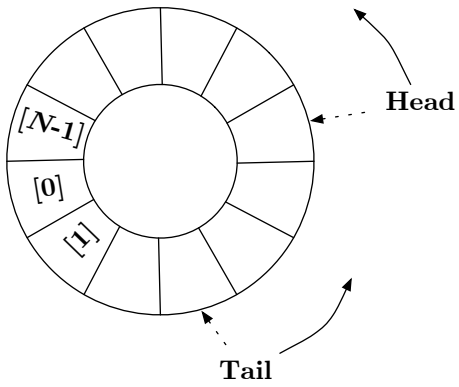


Figure 10.5: Schematic view of an array used as a circular buffer.

1. Check if `g_receiveHead == g_receiveTail`; if they are equal, do nothing; otherwise, go to step 2.

2. Because the head and tail are different, we know the ISR has moved the head index. As a result, we have some unprocessed bytes sitting in the receive buffer. Read the byte indicated by the `g_receiveTail` index, and then increment the tail to the next byte in the buffer.

3. After incrementing `g_receiveTail`, check to make sure the value is not greater-than or equal-to the maximum value defined by the buffer array size. If it is, reset it back to 0.

These steps are implemented as in the following code pieces. Step one is the simple check performed in the infinite loop function.

Example 10.2

```
void loop()
{
   unsigned char dataByte;

   if (ReceivedBytes() > 0)
   {
      dataByte = GetNextReceivedByte();

      /* Do something with the received data byte here. */
   }
}
```

The function for checking the head and tail should look similar to the following; note that global interrupts should be turned off prior to accessing the global head variable since the ISR will be writing to that location.

Example 10.3

```
unsigned char ReceivedBytes (void)
{
   unsigned char receiveHead;
   unsigned char receivedBytes = 0;

   /* Turn off global interrupts here. */

   receiveHead = g_receiveHead;

   /* Restore global interrupts here. */

   if (receiveHead > g_receiveTail)
   {
      receivedBytes = receiveHead - g_receiveTail;
   }
   else if (receiveHead < g_receiveTail)
   {
      /* Check the case that the head has circled around but the tail
         hasn't. */

      receivedBytes = receiveHead + (USART_RECEIVE_BUFFER_SIZE -
         g_receiveTail);
   }

   return receivedBytes;
}
```

Once it is known that the head and tail differ, reading the next byte in the buffer is managed similar to the following function.

Example 10.4

```
unsigned char GetNextReceivedByte (void)
{
   unsigned char receivedByte = 0;

   if (ReceivedBytes() > 0)
   {
      receivedByte = g_receiveBuffer[g_receiveTail++];

      if (g_receiveTail >= USART_RECEIVE_BUFFER_SIZE)
```

```
      {
         g_receiveTail = 0;
      }
   }

   return receivedByte;
}
```

The final component to managing the serial port receiver is the ISR. When the receive interrupt occurs, the ISR needs to store the received byte into the head location of the receive buffer. The AVR-GCC ISR could look something like the following.

Example 10.5

```
#define UCSR0A_ADDR                (unsigned char *) 0xC0
#define UDR0_ADDR                  (unsigned char *) 0xC6

#define UCSR0A_RXC_MASK            0x80
#define UCSR0A_RXC_EMPTY           0x00

ISR(USART_RX_vect)
{
   unsigned char *portUSARTDataRegister;
   unsigned char *portUSARTControlAndStatusRegisterA;
   unsigned char shadow;

   portUSARTDataRegister = UDR0_ADDR;
   portUSARTControlAndStatusRegisterA = UCSR0A_ADDR;

   while (((*portUSARTControlAndStatusRegisterA) & UCSR0A_RXC_MASK) !=
       UCSR0A_RXC_EMPTY)
   {
      g_receiveBuffer[g_receiveHead++] = *portUSARTDataRegister;

      if (g_receiveHead >= USART_RECEIVE_BUFFER_SIZE)
      {
         g_receiveHead = 0;
      }
   }
}
```

Next, consider the steps necessary for managing the serial port transmitter. Note the process is mostly the inverse of the receive process; however, the transmission interrupt is only turned on while there are bytes to be sent. Three more global variables are needed.

Example 10.6

```
#define USART_TRANSMIT_BUFFER_SIZE   80

unsigned char g_transmitBuffer[USART_TRANSMIT_BUFFER_SIZE];
unsigned char g_transmitHead;
unsigned char g_transmitTail;
```

As in the receiver case, the `g_transmitBuffer` is reserved memory used to store the outbound data stream, and `g_transmitHead` and `g_transmitTail` are indices used to write/read to/from the buffer. The following steps are used to transmit a stream of bytes.

1. Write each new byte at `g_transmitHead` index, and then increment the head to the next byte in the buffer.
2. After incrementing `g_transmitHead`, check to make sure the value is not greater-than or equal-to the maximum value defined by the buffer array size. If it is, reset it back to 0.
3. Once the entire message is written to the buffer, turn on the transmitter by enabling the proper interrupt. Once the interrupt is enabled, the *TX* ISR will be called which will start the process of copying the transmit buffer bytes to the serial port transmitter.

These steps are implemented as in the following function.

Example 10.7

```
#define UCSR0B_ADDR                (unsigned char *) 0xC1

#define UCSR0B_TXEN_MASK           0x08
#define UCSR0B_TXEN_ON             0x08
#define UCSR0B_TXEN_OFF            0x00

void TransmitString (const char *bytes, unsigned char numberOfBytes)
{
   unsigned char *portUSARTControlAndStatusRegisterB;
   unsigned char shadow;
   unsigned char i;

   for (i = 0; i < numberOfBytes; i++)
   {
      g_transmitBuffer[g_transmitHead++] = (unsigned char) bytes[i];

      if (g_transmitHead >= USART_TRANSMIT_BUFFER_SIZE)
      {
         g_transmitHead = 0;
      }
   }
```

```
   /* Turn on the TX interrupt. */
   portUSARTControlAndStatusRegisterB = UCSR0B_ADDR;

   shadow = *portUSARTControlAndStatusRegisterB;
   shadow &= ~(UCSR0B_TXEN_MASK);
   shadow |= (UCSR0B_TXEN_ON);
   *portUSARTControlAndStatusRegisterB = shadow;
}
```

The final component to manage the serial port transmitter is the ISR. When the transmit interrupt occurs, a check is made to see if any more bytes exist in the buffer. If no more bytes are to be sent, the *TX* interrupt is turned off. However, if more bytes need to be transmitted, the ISR copies the next buffer byte into the transmit data register and the *TX* interrupt is reenabled. The AVR-GCC ISR could look something like the following.

Example 10.8

```
ISR(USART_UDRE_vect)
{
   unsigned char *portUSARTDataRegister;
   unsigned char *portUSARTControlAndStatusRegisterB;
   unsigned char shadow;

   portUSARTDataRegister = UDR0_ADDR;
   portUSARTControlAndStatusRegisterB = UCSR0B_ADDR;

   shadow = *portUSARTControlAndStatusRegisterB;
   shadow &= ~(UCSR0B_TXEN_MASK);

   if (g_transmitHead != g_transmitTail)
   {
      *portUSARTDataRegister = g_transmitBuffer[g_transmitTail++];

      if (g_transmitTail >= USART_TRANSMIT_BUFFER_SIZE)
      {
         g_transmitTail = 0;
      }

      shadow |= (UCSR0B_TXEN_ON);
   }
   else
   {
      shadow |= (UCSR0B_TXEN_OFF);
   }

   *portUSARTControlAndStatusRegisterB = shadow;
```

```
}
```

One important modification can be made to the code presented in order to make the ISR functions more efficient. Consider making the length of each buffer array a power of 2. Then managing the indices can be made quicker by using a bit-mask as in the following example.

Example 10.9

```
#define USART_TRANSMIT_BUFFER_SIZE   32
#define USART_TRANSMIT_BUFFER_MASK   0x1F

ISR(USART_UDRE_vect)
{
   if (g_transmitHead != g_transmitTail)
   {
      *portUSARTDataRegister = g_transmitBuffer[g_transmitTail++];

      g_transmitTail &= USART_TRANSMIT_BUFFER_MASK;
   }
}
```

This concept can be taken one step further if the microcontroller has enough memory by letting each buffer be 256 bytes in length. Then, because an `unsigned char` is and 8-bit entity, the bit-mask is unnecessary as in the following example.

Example 10.10

```
#define USART_TRANSMIT_BUFFER_SIZE   256

ISR(USART_UDRE_vect)
{
   if (g_transmitHead != g_transmitTail)
   {
      *portUSARTDataRegister = g_transmitBuffer[g_transmitTail++];
   }
}
```

Now, when the head or tail is sitting at 255 and increased by one, it will naturally wrap around to 0. Personally, I normally don't like this kind of code as it represents poor maintainability; that is, given to a software developer, they would not automatically recognize that the wrap around will occur. However, because we are dealing with ISRs, sometimes the low-level functionality is more important than the maintainability of the software. Certainly this would be a valid location for a nice comment stating the fact that the wrap around is intentional.

10.2 PERTINENT REGISTER DESCRIPTIONS

The information presented in this section was taken from ATMEL (2009).

10.2.1 TWBR - TWI BIT RATE REGISTER

Bit	7	6	5	4	3	2	1	0
0xB8	TWBR7	TWBR6	TWBR5	TWBR4	TWBR3	TWBR2	TWBR1	TWBR0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

- TWBR: TWI Bit Rate. `TWBR` selects the division factor for the bit rate generator. The bit rate generator is a frequency divider which generates the SCL clock frequency in the Master modes.

$$f_{\text{SCL}} = \frac{16\ \text{MHz}}{16 + 2(\text{TWBR})(\text{PrescalerValue})}$$

10.2.2 TWCR - TWI CONTROL REGISTER

Bit	7	6	5	4	3	2	1	0
0xBC	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE
Read/Write	R/W	R/W	R/W	R/W	R	R/W	R	R/W
Default	0	0	0	0	0	0	0	0

- TWEA: TWI Enable Acknowledge Bit. The `TWEA` bit controls the generation of the acknowledge pulse. If the `TWEA` bit is written to one, the ACK pulse is generated on the TWI bus if the following conditions are met.

  1. The device's own slave address has been received.
  2. A general call has been received, while the `TWGCE` bit in the `TWAR` is set.
  3. A data byte has been received in Master Receiver or Slave Receiver mode.

  By writing the `TWEA` bit to zero, the device can be virtually disconnected from the 2-wire Serial Bus temporarily. Address recognition can then be resumed by writing the `TWEA` bit to one again.

- TWSTA: TWI START Condition Bit. The application writes the `TWSTA` bit to one when it desires to become a Master on the 2-wire Serial Bus. The TWI hardware checks if the bus is available and generates a START condition on the bus if it is free. However, if the bus is not free, the TWI waits until a STOP condition is detected and then generates a new START condition to claim the bus Master status. `TWSTA` must be cleared by software when the START condition has been transmitted.

- TWSTO: TWI STOP Condition Bit. Writing the TWSTO bit to one in Master mode will generate a STOP condition on the 2-wire Serial Bus. When the STOP condition is executed on the bus, the TWSTO bit is cleared automatically. In Slave mode, setting the TWSTO bit can be used to recover from an error condition. This will not generate a STOP condition, but the TWI returns to a well-defined unaddressed Slave mode and releases the SCL and SDA lines to a high impedance state.

- TWWC: TWI Write Collision Flag. The TWWC bit is set when attempting to write to the TWI Data Register (TWDR) when TWINT is low. This flag is cleared by writing the TWDR Register when TWINT is high.

- TWEN: TWI Enable Bit. The TWEN bit enables TWI operation and activates the TWI interface. When TWEN is written to one, the TWI takes control over the I/O pins connected to the SCL and SDA pins, enabling the slew-rate limiters and spike filters. If this bit is written to zero, the TWI is switched off and all TWI transmissions are terminated, regardless of any ongoing operation.

- See Ch. 9 for interrupt-related bit descriptions.

10.2.3 TWSR - TWI STATUS REGISTER

Bit	7	6	5	4	3	2	1	0
0xB9	TWS7	TWS6	TWS5	TWS4	TWS3	-	TWPS1	TWPS0
Read/Write	R	R	R	R	R	R	R/W	R/W
Default	1	1	1	1	1	0	0	0

- TWS7-3: TWI Status. These 5 bits reflect the status of the TWI logic and the 2-wire Serial Bus. The different status codes are described later in this section. Note that the value read from TWSR contains both the 5-bit status value and the 2-bit pre-scaler value. The application designer should mask the pre-scaler bits to zero when checking the Status bits. This makes status checking independent of pre-scaler setting.

- TWPS1-0: TWI Pre-scaler Bits. These bits can be read and written, and control the bit rate pre-scaler.

10.2.4 TWDR - TWI DATA REGISTER

Bit	7	6	5	4	3	2	1	0
0xBB	TWD7	TWD6	TWD5	TWD4	TWD3	TWD2	TWD1	TWD0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	1	1	1	1	1	1	1	1

Table 10.1: TWI Bit Rate Pre-scaler

TWPS1-0	Pre-scaler Value
00	1
01	4
10	16
11	64

- TWD7-0: TWI Data. In Transmit mode, `TWDR` contains the next byte to be transmitted. In Receive mode, the `TWDR` contains the last byte received. It is writable while the TWI is not in the process of shifting a byte. This occurs when the TWI Interrupt Flag (`TWINT`) is set by hardware. Note that the Data Register cannot be initialized by the user before the first interrupt occurs. The data in `TWDR` remains stable as long as `TWINT` is set. While data is shifted out, data on the bus is simultaneously shifted in. `TWDR` always contains the last byte present on the bus, except after a wake up from a sleep mode by the TWI interrupt. In this case, the contents of TWDR is undefined. In the case of a lost bus arbitration, no data is lost in the transition from Master to Slave. Handling of the ACK bit is controlled automatically by the TWI logic, the CPU cannot access the ACK bit directly.

10.2.5 TWAR - TWI SLAVE ADDRESS REGISTER

Bit	7	6	5	4	3	2	1	0
0xBA	TWA6	TWA5	TWA4	TWA3	TWA2	TWA1	TWA0	TWGCE
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	1	1	1	1	1	1	1	0

- TWA6-0: TWI Slave Address. The `TWAR` should be loaded with the 7-bit Slave address to which the TWI will respond when programmed as a Slave Transmitter or Receiver, and not needed in the Master modes. In multi master systems, `TWAR` must be set in masters which can be addressed as Slaves by other Masters.

- TWGCE: TWI General Call Recognition Enable Bit. This bit is used to enable recognition of the general call address (0x00). There is an associated address comparator that looks for the slave address (or general call address if enabled) in the received serial address. If a match is found, an interrupt request is generated.

10.2.6 TWAMR - TWI SLAVE ADDRESS MASK REGISTER

Bit	7	6	5	4	3	2	1	0
0xBD	TWAM6	TWAM5	TWAM4	TWAM3	TWAM2	TWAM1	TWAM0	-
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R
Default	0	0	0	0	0	0	0	0

- TWAM6-0: TWI Address Mask. The TWAMR can be loaded with a 7-bit Slave Address mask. Each of the bits in TWAMR can mask (disable) the corresponding address bits in the TWI Address Register (TWAR). If the mask bit is set to one, then the address match logic ignores the comparison between the incoming address bit and the corresponding bit in TWAR.

10.2.7 SPCR - SPI CONTROL REGISTER

Bit	7	6	5	4	3	2	1	0
0x4C	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

- SPE: SPI Enable. When the SPE bit is written to one, the SPI is enabled. This bit must be set to enable any SPI operations.
- DORD: Data Order. When the DORD bit is written to one, the LSB of the data word is transmitted first. When the DORD bit is written to zero, the MSB of the data word is transmitted first.
- MSTR: Master/Slave Select. This bit selects Master SPI mode when written to one, and Slave SPI mode when written logic zero. If $\overline{SS}$ is configured as an input and is driven low while MSTR is set, MSTR will be cleared, and SPIF in SPSR will become set. The user will then have to set MSTR to re-enable SPI Master mode.
- CPOL: Clock Polarity. When this bit is written to one, SCK is high when idle. When CPOL is written to zero, SCK is low when idle.
- CPHA: Clock Phase. The settings CPHA determine if data is sampled on the leading (first) or trailing (last) edge of SCK. When CPHA is written to zero, data is sampled on the leading edge. When CPHA is written to one, data is sampled on the trailing edge.
- SPR1-0: SPI Clock Rate Select Bits. These two bits control the SCK rate of the device configured as a Master. SPR1:0 have no effect on the Slave. The relationship between SCK and the Oscillator Clock frequency f_{osc} is shown in Table 10.2.
- See Ch. 9 for interrupt-related bit descriptions.

Table 10.2: Relationship Between SCK and f_{osc}.

SPI2X	SPR1-0	SCK Frequency
0	00	$f_{osc}/4$
0	01	$f_{osc}/16$
0	10	$f_{osc}/64$
0	11	$f_{osc}/128$
1	00	$f_{osc}/2$
1	01	$f_{osc}/8$
1	10	$f_{osc}/32$
1	11	$f_{osc}/64$

10.2.8 SPSR - SPI STATUS REGISTER

Bit	7	6	5	4	3	2	1	0
0x4D	SPIF	WCOL	-	-	-	-	-	SPI2X
Read/Write	R	R	R	R	R	R	R	R/W
Default	0	0	0	0	0	0	0	0

- WCOL: Write COLlision Flag. The WCOL bit is set if the SPI Data Register (SPDR) is written during a data transfer. The WCOL bit is cleared by first reading the SPI Status Register with WCOL set, and then accessing the SPI Data Register.

- SPI2X: Double SPI Speed Bit. When this bit is written logic one the SPI speed (SCK Frequency) will be doubled when the SPI is in Master mode. This means that the minimum SCK period will be two CPU clock periods. When the SPI is configured as Slave, the SPI is only guaranteed to work at $f_{osc}/4$ or lower.

- See Ch. 9 for interrupt-related bit descriptions.

10.2.9 SPDR - SPI DATA REGISTER

Bit	7	6	5	4	3	2	1	0
0x4E	SPD7	SPD6	SPD5	SPD4	SPD3	SPD2	SPD1	SPD0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	-	-	-	-	-	-	-	-

- SPD7-0: SPI Data. The SPI Data Register is a read/write register used for data transfer between the Register File and the SPI Shift Register. Writing to the register initiates data transmission. Reading the register causes the Shift Register Receive buffer to be read.

10.2.10 UDR0 - USART0 I/O DATA REGISTER

Bit	7	6	5	4	3	2	1	0
0xC6	UD7	UD6	UD5	UD4	UD3	UD2	UD1	UD0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

- UD7-0: USART0 Data. The USART0 Transmit Data Buffer Register and USART0 Receive Data Buffer Registers share the same I/O address referred to as USART0 Data Register or UDR0. The Transmit Data Buffer Register (TXB) will be the destination for data written to the UDR0 Register location. Reading the UDR0 Register location will return the contents of the Receive Data Buffer Register (RXB).

  For 5-, 6-, or 7-bit characters the upper unused bits will be ignored by the Transmitter and set to zero by the Receiver.

  The transmit buffer can only be written when the UDRE0 Flag in the UCSR0A Register is set. Data written to UDR0 when the UDRE0 Flag is not set, will be ignored by the USART0 Transmitter. When data is written to the transmit buffer and the Transmitter is enabled, the Transmitter will load the data into the Transmit Shift Register when the Shift Register is empty. Then the data will be serially transmitted on the TxD0 pin.

  The receive buffer consists of a two level FIFO. The FIFO will change its state whenever the receive buffer is accessed. Due to this behavior of the receive buffer, do not use Read-Modify-Write instructions (SBI and CBI) on this location. Be careful when using bit test instructions (SBIC and SBIS) since these also will change the state of the FIFO.

10.2.11 UCSR0A - USART0 CONTROL AND STATUS REGISTER A

Bit	7	6	5	4	3	2	1	0
0xC0	RXC0	TXC0	UDRE0	FE0	DOR0	UPE0	U2X0	MPCM0
Read/Write	R	R/W	R	R	R	R	R/W	R/W
Default	0	0	1	0	0	0	0	0

- FE0: Frame Error. This bit is set if the next character in the receive buffer had a Frame Error when received, i.e., when the first stop bit of the next character in the receive buffer is zero. This bit is valid until the receive buffer (UDR0) is read. The FE0 bit is zero when the stop bit of received data is one. Always set this bit to zero when writing to UCSR0A.

- DOR0: Data OverRun. This bit is set if a Data OverRun condition is detected. A Data OverRun occurs when the receive buffer is full (two characters), it is a new character waiting in the Receive Shift Register, and a new start bit is detected. This bit is valid until the receive buffer (UDR0) is read. Always set this bit to zero when writing to UCSR0A.

- UPE0: USART0 Parity Error. This bit is set if the next character in the receive buffer had a Parity Error when received and the Parity Checking was enabled at that point (`UPM01 = 1`). This bit is valid until the receive buffer (`UDR0`) is read. Always set this bit to zero when writing to `UCSR0A`.

- U2X0: Double the USART0 Transmission Speed. This bit only has effect for the asynchronous operation. Write this bit to zero when using synchronous operation.

  Writing this bit to one will reduce the divisor of the baud rate divider from 16 to 8, effectively doubling the transfer rate for asynchronous communication.

- MPCM0: Multi-processor Communication Mode. This bit enables the Multi-processor Communication mode. When the `MPCM0` bit is written to one, all the incoming frames received by the USART0 Receiver that do not contain address information will be ignored. The Transmitter is unaffected by the `MPCM0` setting.

- See Ch. 9 for interrupt-related bit descriptions.

10.2.12 UCSR0B - USART0 CONTROL AND STATUS REGISTER B

Bit	7	6	5	4	3	2	1	0
0xC1	RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	UCSZ02	RXB80	TXB80
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W
Default	0	0	0	0	0	0	0	0

- RXEN0: USART0 Receiver Enable. Writing this bit to one enables the USART0 Receiver. The Receiver will override normal port operation for the `RxD0` pin when enabled. Disabling the Receiver will flush the receive buffer invalidating the `FE0`, `DOR0`, and `UPE0` Flags.

- TXEN0: USART0 Transmitter Enable. Writing this bit to one enables the USART0 Transmitter. The Transmitter will override normal port operation for the `TxD0` pin when enabled. Disabling the Transmitter will not become effective until ongoing and pending transmissions are completed, i.e., when the Transmit Shift Register and Transmit Buffer Register do not contain data to be transmitted. When disabled, the Transmitter will no longer override the `TxD0` port.

- UCSZ02: USART0 Character Size. The `UCSZ02` bit combined with the `UCSZ01:0` bits in `UCSR0C` sets the number of data bits (Character SiZe) in a frame the Receiver and Transmitter use.

- RXB80: USART0 Receive Data Bit 8. `RXB80` is the ninth data bit of the received character when operating with serial frames with nine data bits. It must be read before reading the low bits from `UDR0`.

- TXB80: USART0 Transmit Data Bit 8. TXB80 is the ninth data bit in the character to be transmitted when operating with serial frames with nine data bits. It must be written before writing the low bits to UDR0.

- See Ch. 9 for interrupt-related bit descriptions.

10.2.13 UCSR0C - USART0 CONTROL AND STATUS REGISTER C

Bit	7	6	5	4	3	2	1	0
0xC2	UMSEL01	UMSEL00	UPM01	UPM00	USBS0	UCSZ01	UCSZ00	UCPOL0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	1	1	0

- UMSEL01-0: USART0 Mode Select Bits. These bits select the mode of operation of the USART0 as shown in Table 10.3.

Table 10.3: UMSEL0 Bit Settings.

UMSEL01-0	Mode
00	Asynchronous USART
01	Synchronous USART
10	Reserved
11	Master SPI (MSPIM)

- UPM01-0: USART0 Parity Mode. These bits enable and set type of parity generation and check. If enabled, the Transmitter will automatically generate and send the parity of the transmitted data bits within each frame. The Receiver will generate a parity value for the incoming data and compare it to the UPM0 setting. If a mismatch is detected, the UPE0 Flag in UCSR0A will be set.

Table 10.4: UPM0 Bit Settings.

UPM01-0	Parity Mode
00	Disabled
01	Reserved
10	Enabled, Even Parity
11	Enabled, Odd Parity

- USBS0: USART0 Stop Bit Select. This bit selects the number of stop bits to be inserted by the Transmitter. The Receiver ignores this setting. If USBS0 is cleared, there is 1 stop bit. If USBS0 is set, there are 2 stop bits.

- UCSZ01-0: USART0 Character Size. The UCSZ01:0 bits combined with the UCSZ02 bit in UCSR0B sets the number of data bits (Character SiZe) in a frame the Receiver and Transmitter use.

Table 10.5: UCSZ0 Bit Settings.

UCSZ02-0	Character Size
000	5-bit
001	6-bit
010	7-bit
011	8-bit
100	Reserved
101	Reserved
110	Reserved
111	9-bit

- UCPOL0: USART0 Clock Polarity. This bit is used for synchronous mode only. Write this bit to zero when asynchronous mode is used. The UCPOL0 bit sets the relationship between data output change and data input sample, and the synchronous clock (XCK0).

Table 10.6: UCPOL0 Bit Settings.

UCPOL0	Transmitted Data Changed	Received Data Sampled
0	Rising XCK0 Edge	Falling XCK0 Edge
1	Falling XCK0 Edge	Rising XCK0 Edge

10.2.14 UBRR0H AND UBRR0L - USART0 BAUD RATE REGISTERS

Bit	7	6	5	4	3	2	1	0
0xC5	-	-	-	-	UBRR0[11:8]			
0xC4	UBRR0[7:0]							
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

- UBRR011-0: USART0 Baud Rate Bits. This is a 12-bit register which contains the USART0 baud rate. The UBRR0H contains the four most significant bits, and the UBRR0L contains the eight least significant bits of the USART0 baud rate. Ongoing transmissions by the Transmitter and Receiver will be corrupted if the baud rate is changed. Writing UBRR0L will trigger an immediate update of the baud rate pre-scaler.

Table 10.7: UBRR0 Bit Settings for 16 MHz System Clock.

Baud Rate (bps)	U2X0 = 0, UBRR0	U2X0 = 1, UBRR0
2400	416	832
4800	207	416
9600	103	207
14400	68	138
19200	51	103
28800	34	68
38400	25	51
57600	16	34
76800	12	25
115200	8	16
230400	3	8
250000	3	7
500000	1	3
1000000	0	1

PROBLEMS

10.1 Determine the appropriate bit settings for `UCSR0A`, `UCSR0B`, `UCSR0C`,and `UBRR0` to manage a serial interface using the following specific details:

- use normal transmission speed (i.e., disable the x2 speed),
- disable the multi-processor communication mode,
- turn on the *RX* complete interrupt, turn off the *TX* complete interrupt, turn on the data register empty interrupt,
- turn on the receiver, turn off the transmitter,
- set the character size to 8 bits,
- use the asynchronous USART mode,
- use no parity,
- use 1 stop bit,
- set the baud rate to 115200 bits per second.

10.2 Create a function that initializes the USART based on the values determined in problem 10.1.

10.3 Create an ISR for both the *RX* and *TX* interrupts.

10.4 Create a program that utilizes the USART to send and receive bytes via the USB converter chip to and from the host computer terminal program. Use a switch statement to send a

different message in response to different bytes received. For example, consider the following code piece.

Example 10.11

```
switch (GetNextReceivedByte())
{
   case '1':
      TransmitString("One\n", 4);
      break;

   case '2':
      TransmitString("Two\n", 4);
      break;

   default:
      TransmitString("Default\n", 8);
      break;
}
```

Download your program and open a terminal program operating at 115200 baud rate. Verify that you receive appropriate messages in response to bytes that you send to the embedded device via the terminal program.

CHAPTER 11

Assembly Language

11.1 INTRODUCTION

Up to this point, every program created has been via writing a high-level set of instructions describing the behavior we want the microcontroller to follow. Sometimes, the instructions have been very specific, such as setting the address of a pointer to the location of some register, and then setting the desired bit patterns via the pointer. However, many other blocks of code have been much more general, such as creating loops or making sub-routine calls. By now, you probably have a good understanding of how these behavioral descriptions function, but really such high-level instructions are taken for granted. The reality is that a simple loop requires careful management of the loop variable within a CPU register, and sub-routine calls need to keep track of parameters being passed in and out, return location, register preservation, etc. The biggest benefit of high-level languages is that general behavior can be created in a source-code listing and processed with a set of tools in order to generate a set of machine-level instructions that perform the intended operations. The most common high-level language used in embedded systems is C and C++. One of the anomalies in technology, C has been around since the 1970's, and it doesn't appear to be leaving any time soon. Perhaps, the reason is that C is considered a low-level high-level language. Meaning, that C is high-level enough to create behavioral descriptions in source code, yet C is low-level enough that bit-wise operations can be performed on register contents in order to manipulate hardware directly.

The entire process of translating high-level source code into a machine-level program is shown in Fig. 11.1. The process begins by passing a source-code file `file.c` (or `file.cpp`) to the Preprocessor, whose job it is to manage all of the preprocessor directives. The two most common directives are the `#include` and `#define` statements. The preprocessor scans the entire source file looking for these statements. Each time it parses a `#include<filei.h>`, it replaces the preprocessor statement with the contents of the file listed in the angle brackets, `filei.h`. Whenever the preprocessor reads a `#define` *name value*, it adds an entry to a preprocessor table such that whenever the phrase *name* is found throughout the rest of the preprocessing stage, it is replaced by *value*. The preprocessor tool continues this way until the entire file does not contain any more preprocessing commands; that is, all have been replaced by their respective values.

The output of the preprocessor `file.i` (or `file.ii`) is provided as input to the Compiler, whose job it is to translate the general behavior and the specific statements into blocks of assembly instructions specific to the target processor. While the primary goal of the compiler is to create an assembly-code listing that will cause the desired behavior, a secondary objective is to be as efficient as possible. It is an important aspect of embedded systems to have programs with a very small foot-

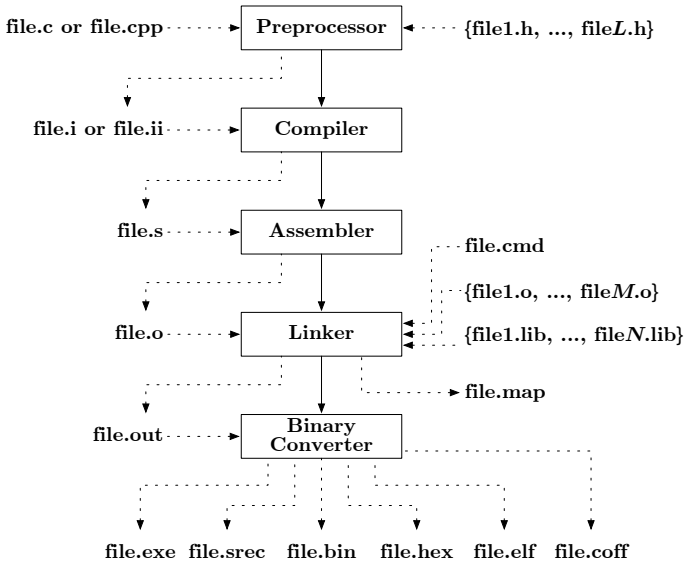


Figure 11.1: The entire build process beginning with a high-level source file and ending with a binary program.

print. This is especially true of microcontrollers that often have a very small amount of data and code space.

When the compiler completes, it will have generated an assembly file `file.s` that is passed to the Assembler. The assembler tool performs yet-another translation from assembly instructions into machine instructions. This process is much simpler compared to the high-level translation between C and assembly. The reason being there is a one-to-one correspondence between each assembly instruction and its target machine instruction. By contrast, the compiler must analyze a single C instruction and generate a set of several assembly instructions that will perform the desired function.

The set of machine instructions are stored in an object file `file.o` which must be processed by the Linker tool. This final step performs many important functions mostly relating to physical address assignment. Up to this point in the process, the tools have been only concerned with creating instructions that the CPU will be able to understand in order to function in the desired way. But none of the tools have had any knowledge of the physical target hardware. By that, we mean the target processor exists in a system connected to various forms of memory used to hold the program such as non-volatile flash, often called "code-space," and to provide space for variables in volatile RAM, often called "data-space." But because most programs involve more than one object file, there is no way for the compiler or assembler to associate instructions or variables with physical addresses.

Instead, the assembler will use relative addressing which will use a base address at the top of the file, and then offset deltas in-between each instruction or memory location. The linker will then take all of the object files and create a big table of function and memory labels, and assign physical addresses to all of the table entries. In order to do this, a special linker-command file `file.cmd` is passed in that contains all of the physical attributes of the target system, among other things. The other thing the linker takes in is the set of any pre-built object libraries, for example the ANSI C function `printf()` can always be called by a program, but you never actually write the function yourself as the functionality has already been built and is stored in a library file `filei.lib`.

The linker is often considered the final step in building the executable program `file.out`. However, many embedded systems require a special machine-code file format in order to download the program. There are many programs that will convert from the linker output (`.out`) file format into a tool-required file format; for example, Motorola s-record (`.srec`), Common Object File Format (`.coff`), Executable and Linkable Format (`.elf`), Intel HEX (`.hex`), etc.

11.2 ARDUINO TOOL-CHAIN

The Arduino development environment, like any other integrated tool-chain, hides the details of the build process from the user. In fact, the concept of the Arduino tool is to hide as much as possible from the software developer. This is evident by the fact that the tool only allows the user to create `.pde` files which are used to generate `.cpp` files before being processed and downloaded. Sometimes, this concept is useful, especially for a novice trying to learn embedded programming. However, eventually engineers need to be able to access the true set of steps in order to understand what is taking place within the system. Of particular interest is having the option to create hand-coded assembly programs. In general, it is best to develop software from a maintainable stand-point, which means "no assembly, ever". The reason will become evident when you perform the chapter problems – assembly programs are difficult for a human to easily comprehend and follow. Thus, they are generally avoided because they present a challenge to maintain. However, there is still a need for some assembly programming in the industry.

Typically, assembly will only find a place within Board Support Packages (BSPs), which are sets of low-level device driver functions necessary for Real-Time Operating Systems (RTOSes) to interface properly with specific target microprocessors. Also, hand-coded assembly can still fill a need in which there is a time-critical or space-limited situation. To understand why, imagine the process of translating prose from chinese to english, and then from english to spanish. Compare that with only one translation between english and spanish. The difference has to do with the fact that english and spanish are generally similar, whereas chinese is generally very different—some dialects have thousands of elements in their alphabet. While C compilers are generally very efficient, hand-coded assembly can always be made smaller and faster. As a personal example, several years ago I was working on a project for which there was a requirement that a Digital Signal Processor (DSP) process real-time encryption using 1-bit cipher-feedback on a 48kHz data stream. This basically means 48,000 encryption function calls needed to complete within 1 second. The encryption algorithm

was developed in C first, but there were not enough cycles available in the DSP to complete the processing in the required amount of time. So, I had to optimize the entire encryption process using 100% assembly code. The result was satisfying at the time, but I doubt I would be able to maintain or even understand that source code today.

The Arduino Integrated Development Environment (IDE) does not allow the user to create an entire program out of assembly. However, the assembler and linker tools do exist on the host computer, so it is just a matter of finding them and figuring out how to call them directly. The first step is locating the programs on your computer. Assuming you installed the Arduino IDE in the `Applications` folder on MacOSX, or placed it at `C:\` on Windows (XP), the path to the tools is given by:

- MacOSX:`/Applications/Arduino.app/Contents/Resources /Java/hardware/tools/avr/bin`
- Windows:`C:\arduino-0018\hardware\tools\avr\bin`

In order to access any of the programs, you need to:

- MacOSX: open `/Applications/Utilities/Terminal`
- Windows: open a `cmd` prompt by selecting `Start`→`Run...`, then typing `cmd` in the `Open:` field

For the rest of this section, let's define the following "environmental variable," so we don't need to re-state the entire path for each file being accessed.

- MacOSX:{*PATH*}=`/Applications/Arduino.app/Contents/Resources /Java/hardware/tools/avr`
- Windows:{*PATH*}=`C:\arduino-0018\hardware\tools\avr`

Now, for an example, consider the following C source-code file that blinks one of the LEDs on the Arduino circuit-board. Beginning with the necessary `#include` and `#defines`.

Example 11.1

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

#define TCCR1A_ADDR                (unsigned char *) 0x80
#define TCCR1A_COMA_MASK           0xC0
#define TCCR1A_COMA_NORMAL         0x00
#define TCCR1A_COMB_MASK           0x30
#define TCCR1A_COMB_NORMAL         0x00
#define TCCR1A_WGM_MASK            0x03
#define TCCR1A_WGM_CTC             0x00
```

```
#define TCCR1B_ADDR                (unsigned char *) 0x81
#define TCCR1B_WGM_MASK            0x18
#define TCCR1B_WGM_CTC             0x08
#define TCCR1B_CS_MASK             0x07
#define TCCR1B_CS_OFF              0x00
#define TCCR1B_CS_DIV_1            0x01
#define TCCR1B_CS_DIV_8            0x02
#define TCCR1B_CS_DIV_64           0x03
#define TCCR1B_CS_DIV_256          0x04
#define TCCR1B_CS_DIV_1024         0x05

#define OCR1AL_ADDR                (unsigned char *) 0x88
#define OCR1AH_ADDR                (unsigned char *) 0x89
#define OCR1A_ADDR                 (unsigned short *) 0x88
#define OCR1A_1_INT_PER_1_SEC      (16000000 / 1024)

#define TIMSK1_ADDR                (unsigned char *) 0x6F
#define TIMSK1_ICIE_MASK           0x20
#define TIMSK1_OCIEB_MASK          0x04
#define TIMSK1_OCIEA_MASK          0x02
#define TIMSK1_TOIE_MASK           0x01

#define SREG_ADDR                  (unsigned char *) 0x5F
#define SREG_GLOBAL_INT_ENABLE     0x80

#define PORTB_DDR_ADDR             (unsigned char *) 0x24
#define PORTB_DATA_ADDR            (unsigned char *) 0x25

#define LED_PIN_MASK               0x20

#define DELAY_IN_SECONDS           1
```

Next, we have the standard `setup()` and `loop()` functions used by the `main()` function. The Arduino IDE always generates and hides the `main()` function, so we need to explicitly add it as shown below.

Example 11.2

```
static volatile unsigned char m_timerCount;

static void setup(void);
static void loop(void);

int main(void)
{
   setup();
```

```
   for (;;)
   {
      loop();
   }

   return 0;
}

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

void setup(void)
{
   unsigned char *statusRegister;
   statusRegister = SREG_ADDR;

   *statusRegister &= ~SREG_GLOBAL_INT_ENABLE;

   InitializeOutput();
   InitializeTimer();

   *statusRegister |= SREG_GLOBAL_INT_ENABLE;
}

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

void loop(void)
{
   unsigned char *portData;

   portData = PORTB_DATA_ADDR;

   while (m_timerCount < DELAY_IN_SECONDS);
   m_timerCount = 0;

   *portData &= ~LED_PIN_MASK;

   while (m_timerCount < DELAY_IN_SECONDS);
   m_timerCount = 0;

   *portData |= LED_PIN_MASK;
}
```

And the necessary functions to initialize the port pin output and the timer interrupt, and, finally, the timer1 ISR itself.

Example 11.3

```
void InitializeOutput (void)
{
   unsigned char *portDDR;
   unsigned char *portData;

   portData = PORTB_DATA_ADDR;
   portDDR = PORTB_DDR_ADDR;

   *portData |= LED_PIN_MASK;
   *portDDR |= LED_PIN_MASK;
}

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

void InitializeTimer (void)
{
   unsigned char *portTimerCounterControlRegisterA;
   unsigned char *portTimerCounterControlRegisterB;
   unsigned short *portOutputCompareRegisterA;
   unsigned char *portTimerCounterInterruptMaskRegister;
   unsigned char shadow;

   portTimerCounterControlRegisterA = TCCR1A_ADDR;
   portTimerCounterControlRegisterB = TCCR1B_ADDR;
   portOutputCompareRegisterA = OCR1A_ADDR;
   portTimerCounterInterruptMaskRegister = TIMSK1_ADDR;

   shadow = *portTimerCounterControlRegisterA;
   shadow &= ~(TCCR1A_COMA_MASK | TCCR1A_COMB_MASK | TCCR1A_WGM_MASK);
   shadow |= (TCCR1A_COMA_NORMAL | TCCR1A_COMB_NORMAL | TCCR1A_WGM_CTC);
   *portTimerCounterControlRegisterA = shadow;

   shadow = *portTimerCounterControlRegisterB;
   shadow &= ~(TCCR1B_CS_MASK | TCCR1B_WGM_MASK);
   shadow |= (TCCR1B_CS_DIV_1024 | TCCR1B_WGM_CTC);
   *portTimerCounterControlRegisterB = shadow;

   *portOutputCompareRegisterA = OCR1A_1_INT_PER_1_SEC;

   shadow = *portTimerCounterInterruptMaskRegister;
   shadow &= ~(TIMSK1_ICIE_MASK | TIMSK1_OCIEB_MASK |
               TIMSK1_OCIEA_MASK | TIMSK1_TOIE_MASK);
   shadow |= TIMSK1_OCIEA_MASK;
   *portTimerCounterInterruptMaskRegister = shadow;

   m_timerCount = 0;
}
```

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

ISR(TIMER1_COMPA_vect)
{
   m_timerCount++;
}
```

Once we create this `BlinkInCTimer1.cpp` file, we can create the machine object code by issuing the following commands at the appropriate command prompt (don't forget to use the full path for the tool).

{*PATH*}/bin/avr-gcc -g -Wall -O2 -mmcu=atmega328p -c BlinkInCTimer1.c

The result of this command will be the object file `BlinkInCTimer1.o`. That is, the preprocessor, compiler, assembler and linker were all executed with this single step. The next command is used to generate the `BlinkInCTimer1.elf` binary file format.

{*PATH*}/bin/avr-gcc -g -Wall -O2 -mmcu=atmega328p -o BlinkInCTimer1.elf BlinkInCTimer1.o

An optional `BlinkInCTimer1.map` file containing the physical memory information for your program can be generated by adding the command-line argument `-Wl,-Map,BlinkInCTimer1.map` to the previous command (between `-mmcu=atmega328p` and `-o` was where I put it). Note that the command is `-Wl` ('el' not 'one'). Another program can be run in order to generate a disassembly listing of your program:

{*PATH*}/bin/avr-objdump -h -S BlinkInCTimer1.elf > BlinkInCTimer1.lst

This may be helpful when first learning how to create your own assembly-language programs. The final step necessary to create your program is to convert the `.elf` file into the `.hex` file format with the following command:

{*PATH*}/bin/avr-objcopy -j .text -j .data -O ihex BlinkInCTimer1.elf BlinkInCTimer1.hex

Once your `.hex` file has been created, we need a way of downloading it to the target platform. Often, this requires an external hardware debugger; a separate device that connects your host computer to the target platform. Alternatively, many embedded environments have a boot-loader program installed on the target microprocessor. This little program is "permanent" as long as

some catastrophic failure does not occur (e.g., erasing the entire flash). Its purpose is to provide the developer, a method for updating the main application as we have been doing all along with the Arduino. The boot-loader has the ability to communicate with the host computer to receive commands indicating that a new program is to be loaded. In the case that the program is already downloaded, all the boot-loader does is jump to the entry point of the application. What this means for us is that we can download our new program via serial communication with the boot-loader program, which is what the Arduino IDE does when you tell it to upload your program. The command is given by:

{*PATH*}/bin/avrdude -C [*conffile*] -p m328p -c stk500v1 -P [*serialport*] -b 57600 -D -U flash:w:BlinkInCTimer1.hex:i

where the configuration file is given by the path {*PATH*}/etc/avrdude.conf and the serial port can be found using the Arduino IDE under the Tools→Serial Port menu. Typically, this will be something like:

- MacOSX: /dev/tty.usbserial-A900abQ1
- Windows: com3

If you issue the command, you should see a verbose listing of what the program is doing while it is connected to the device. In particular, it should have written your program object code to the flash, and you should see the LED blink at a 1-second rate after you reset the board.

11.3 ARDUINO ASSEMBLY

Now consider using only assembly instructions to perform the same blink functionality. The Arduino tool-chain allows for C-like preprocessor commands, so we can create constants just like in C.

Example 11.4

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

#define TCCR1A_ADDR          0x80
#define TCCR1A_VALUE         0x00

#define TCCR1B_ADDR          0x81
#define TCCR1B_VALUE         0x0D

#define OCR1AL_ADDR          0x88
#define OCR1AL_VALUE         0x09

#define OCR1AH_ADDR          0x89
#define OCR1AH_VALUE         0x3D
```

```
#define TIMSK1_ADDR          0x6F
#define TIMSK1_VALUE         0x02

#define PORTB_DDR_ADDR       0x24
#define PORTB_DATA_ADDR      0x25

#define LED_PIN_MASK         0x20

#define DELAY_IN_SECONDS     1
```

The rest of the assembly description looks like the following (at least, this is my version of it).

Example 11.5

```
/*
   r16 = tempByte
   r17 = timerFlag
   r18 = currentOutput
*/
        .section .text

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

        .global main
main:

setup:
        cli

        rcall   InitializeOutput
        rcall   InitializeTimer

        sei

loop:

WaitToClear:
        cpi     r17, DELAY_IN_SECONDS
        brne    WaitToClear

        clr     r17

        cbr     r18, LED_PIN_MASK
        sts     PORTB_DATA_ADDR, r18

WaitToSet:
        cpi     r17, DELAY_IN_SECONDS
```

```
        brne    WaitToSet

        clr     r17

        sbr     r18, LED_PIN_MASK
        sts     PORTB_DATA_ADDR, r18

        rjmp    loop

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

InitializeOutput:
        ldi     r18, LED_PIN_MASK
        sts     PORTB_DDR_ADDR, r18
        sts     PORTB_DATA_ADDR, r18

        ret

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

InitializeTimer:
        ldi     r16, TCCR1A_VALUE
        sts     TCCR1A_ADDR, r16

        ldi     r16, TCCR1B_VALUE
        sts     TCCR1B_ADDR, r16

        ldi     r17, OCR1AH_VALUE
        ldi     r16, OCR1AL_VALUE
        sts     OCR1AH_ADDR, r17
        sts     OCR1AL_ADDR, r16

        ldi     r16, TIMSK1_VALUE
        sts     TIMSK1_ADDR, r16

        clr     r17

        ret

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

        .global TIMER1_COMPA_vect
TIMER1_COMPA_vect:
        inc     r17
        reti

        .end
```

One point about this program is the usage of specific registers to perform certain functions. Near the top of the source-code listing is a comment block detailing the fact that registers `r16-r18` are used for specific purposes. Because there is no compiler translation, then there is no mysterious under-cover register usage going on. Whatever we do with the registers is totally up to us. So, I have arbitrarily selected these three registers to always be used for the purpose of keeping track of some global memory. By doing this, I can reduce the number of instructions since I don't need to reload memory every time I access a piece of information. The comment block is very helpful in reminding the programmer which registers hold what values.

It is beyond the scope of this chapter to explain assembly-code programming further. Rather, the goal is to state how to set up the tools to build an object file given an assembly program. The list of all ATmega328P assembly instructions may be found in Apx. C. Additionally, various aspects of assembly programming on the ATmega328P are detailed within the Arduino tool-chain documentation found at:

- MacOSX: {*PATH*}`/doc/avr-libc/assembler.html`
- Windows: {*PATH*}`\doc\avr-libc\avr-libc-user-manual\assembler.html`

Once you create the file `BlinkInAsmTimer1.S` (note: the 'S' must be capital for the preprocessor to run on it), you can issue the same exact commands as in the case of the `BlinkInCTimer1.c` program.

11.4 ARDUINO INLINE ASSEMBLY

Now that we have seen all the work involved in creating a program from direct assembly language, let's move back to the Arduino IDE. We can use the Arduino editor to create pseudo-assembly programs by using inline assembly. Basically, we have the ability to insert specific assembly instructions into the middle of C functions. There are a few new challenges including how to know what registers are already being used by the compiler, and how to use variables declared in C in an assembly instruction, but we gain the benefit of using the Arduino IDE. Consider the following inline assembly program to perform the blink functionality, as before.

Example 11.6

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

#define TCCR1A_ADDR          0x80
#define TCCR1A_VALUE         0x00

#define TCCR1B_ADDR          0x81
#define TCCR1B_VALUE         0x0D

#define OCR1AL_ADDR          0x88
#define OCR1AL_VALUE         0x09
```

```
#define OCR1AH_ADDR         0x89
#define OCR1AH_VALUE        0x3D

#define TIMSK1_ADDR         0x6F
#define TIMSK1_VALUE        0x02

#define PORTB_DDR_ADDR      0x24
#define PORTB_DATA_ADDR     0x25

#define LED_PIN_MASK        0x20

#define DELAY_IN_SECONDS    1

/*
   r16 = tempByte
   r17 = timerFlag
   r18 = currentOutput
*/

void setup()
{
   asm volatile("cli\n\t"
                "\n\t"
                "ldi r18, %[theLedPinMask]\n\t"
                "sts %[thePortBDdrAddress], r18\n\t"
                "sts %[thePortBDataAddress], r18\n\t"
                "\n\t"
                "ldi r16, %[theTccr1aValue]\n\t"
                "sts %[theTccr1aAddress], r16\n\t"
                "ldi r16, %[theTccr1bValue]\n\t"
                "sts %[theTccr1bAddress], r16\n\t"
                "ldi r17, %[theOcr1ahValue]\n\t"
                "ldi r16, %[theOcr1alValue]\n\t"
                "sts %[theOcr1ahAddress], r17\n\t"
                "sts %[theOcr1alAddress], r16\n\t"
                "ldi r16, %[theTimsk1Value]\n\t"
                "sts %[theTimsk1Address], r16\n\t"
                "clr r17\n\t"
                "\n\t"
                "sei\n\t"
                "\n\t"
                "1:" "cpi r17, %[theDelay]\n\t"
                "brne 1b\n\t"
                "\n\t"
                "clr r17\n\t"
                "\n\t"
                "cbr r18, %[theLedPinMask]\n\t"
```

```
                "sts %[thePortBDataAddress], r18\n\t"
                "\n\t"
                "2:" "cpi r17, %[theDelay]\n\t"
                "brne 2b\n\t"
                "\n\t"
                "clr r17\n\t"
                "\n\t"
                "sbr r18, %[theLedPinMask]\n\t"
                "sts %[thePortBDataAddress], r18\n\t"
                "rjmp 1b\n\t"
                :
                : [theLedPinMask] "M" (LED_PIN_MASK),
                  [thePortBDdrAddress] "M" (PORTB_DDR_ADDR),
                  [thePortBDataAddress] "M" (PORTB_DATA_ADDR),
                  [theTccr1aValue] "M" (TCCR1A_VALUE),
                  [theTccr1aAddress] "M" (TCCR1A_ADDR),
                  [theTccr1bValue] "M" (TCCR1B_VALUE),
                  [theTccr1bAddress] "M" (TCCR1B_ADDR),
                  [theOcr1ahValue] "M" (OCR1AH_VALUE),
                  [theOcr1ahAddress] "M" (OCR1AH_ADDR),
                  [theOcr1alValue] "M" (OCR1AL_VALUE),
                  [theOcr1alAddress] "M" (OCR1AL_ADDR),
                  [theTimsk1Value] "M" (TIMSK1_VALUE),
                  [theTimsk1Address] "M" (TIMSK1_ADDR),
                  [theDelay] "M" (DELAY_IN_SECONDS));
}

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

void loop()
{
}

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

ISR(TIMER1_COMPA_vect, ISR_NAKED)
{
   asm volatile("inc r17\n\t"
                "reti\n\t"
                :: );
}
```

This program was formed using the structure of the C program, with the exact instructions from the assembly program. In general, inline assembly provides a middle-ground between pure assembly and C, plus it allows us to use the Arduino IDE instead of issuing all the mysterious commands by hand. More details on inline assembly programming on the ATmega328P are provided in the Arduino tool-chain documentation found at:

- MacOSX: {*PATH*}/doc/avr-libc/inline_asm.html
- Windows: {*PATH*}\doc\avr-libc\avr-libc-user-manual\inline_asm.html

Once you create the file `BlinkInlineAsmTimer1.pde` you can build and upload the program using the Arduino IDE just like you have done in all the previous chapter problems.

All three programs exhibited the desired 1-second blink control of the Arduino LED. The difference in program size is presented in Table 11.1. In order to compare the use of inline assembly versus C, both programs were compiled using the command line and the Arduino IDE. As expected, the hand-coded assembly program is about half the size as any of the C programs. The next-smallest program is the C code using all inline assembly instructions being compiled using the command-line instructions. The command-line results are likely smaller due to an extra optimization parameter being turned on that may not be enabled within the IDE.

Table 11.1: Code size comparisons.

Program	Build Method	Code-space Bytes
BlinkInAsmTimer1.S	Command line	216
BlinkInCTimer1.cpp	Arduino IDE	582
BlinkInlineAsmTimer1.cpp	Arduino IDE	512
BlinkInCTimer1.cpp	Command line	406
BlinkInlineAsmTimer1.cpp	Command line	324

11.5 C-INSTRUCTION EFFICIENCY

It is always fortunate to have a high-level language compiler available for an embedded system. To understand why this is true, consider the seemingly simple problem of adding two unsigned integers together. Let $A = \{a_7a_6a_5a_4a_3a_2a_1a_0\}$ and $B = \{b_7b_6b_5b_4b_3b_2b_1b_0\}$ be eight-bit vectors. Then, a "full adder" is a digital circuit that performs the addition of the ith stage

$$[c_i, s_i] = a_i + b_i + c_{i-1}$$

where s_i is called the ith sum bit and c_i is the ith carry-out bit which is fed to the $i + 1$ stage as the carry-in. Note that in this equation, $+$ is one-bit addition, not logical OR. It turns out that it is possible to create fast adders using full adder building blocks along with additional pieces, depending on how fast the adder needs to be. All processors have an Algorithmic/Logic Unit (ALU) that contains many arithmetic blocks capable of performing basic operations such as addition, subtraction and multiplication.

Because every specific arithmetic operation requires underlying hardware to perform the desired function, one of two things happen:

1. either large amounts of digital circuitry are added to support many functions, or

2. only a few simple functions are provided in hardware and the software engineer must explicitly call combinations of simple functions to perform a larger, more-complex function.

As an example, consider the following operations that you might assume the processor has the natural ability to perform:

- 32-bit integer addition/subtraction;
- 32-bit integer multiplication;
- integer division;
- floating-point addition/subtraction, multiplication, division.

It turns out none of these operations exist in hardware on the ATmega328P. On the other hand, it is well-known through the previous chapters that the ability to perform all of these operations is available via the high-level language C, among other languages. The observant reader should be asking themselves how this is possible.

The answer lies in the C compiler, whose job it is to parse and convert the high-level instructions into sets of assembly instructions that will make the most efficient use of the microcontroller's available hardware resources. Referring back to the adder discussion, it turns out that most arithmetic operations are ultimately just repeated use of adders and subtractors. As an example, consider that integer division is just $A = BQ + R$, and multiplication is just repeated addition.

PROBLEMS

11.1 Create a program in C that repeatedly outputs the digits '0' through '9' on a seven-segment display with a one-second interval between each digit.

11.2 Create a program that performs the same function as problem 11.1, but use inline assembly.

11.3 Create a program that performs the same function as problem 11.1, but use assembly. Use the commands presented in this chapter to assemble the program and download it.

11.4 Given the inline assembly source code

```
in r1, 5
sbr r1, 32
mov r2, r1
cbr r2 32
out 5, r2
```

use Apx. C to state what is taking place.

11.5 Compile and load the following source code.

```
unsigned char g_theOrRegister;
unsigned char g_theAndRegister;

void setup()
{
   unsigned char *portDDRB;

   portDDRB = (unsigned char *) 0x24;
   *portDDRB |= 0x20;

   asm volatile("in %[theOrRegister], %[thePort]\n\t"
                "sbr %[theOrRegister], %[theBitMask]\n\t"
                "mov %[theAndRegister], %[theOrRegister]\n\t"
                "cbr %[theAndRegister], %[theBitMask]\n\t"
                "out %[thePort], %[theAndRegister]\n\t"
                : [theOrRegister] "+r" (g_theOrRegister),
                  [theAndRegister] "+r" (g_theAndRegister)
                : [theBitMask] "M" (0x20),
                  [thePort] "M" (0x05));
}

void loop()
{
   /*
      Keep this line to turn on the LED.
   */
   asm volatile("out %[thePort], %[theOrRegister]\n\t"
                : [theOrRegister] "+r" (g_theOrRegister)
                : [thePort] "M" (0x05));

   /*
      Measured operation goes in here.  After you measure
      the following inline assembly instruction, remove
      it and put in your own C instruction to measure
      the number of cycles.
   */

   asm volatile("nop\n\t"
                "nop\n\t"
                "nop\n\t"
                "nop\n\t"
                "nop\n\t"
                ::);

   /*
      Keep this line to turn off the LED.
   */
   asm volatile("out %[thePort], %[theAndRegister]\n\t"
```

```
                : [theAndRegister] "+r" (g_theAndRegister)
                : [thePort] "M" (0x05));
   delay(1000);
}
```

Attach an oscilloscope to the LED output and measure the amount of time that the LED is high. Then remove the `nop` assembly instructions and re-measure the amount of time that the LED is high. Use the difference between the measurements as the time required to execute the five assembly `nop`s. Take this measurement and the number of cycles required for the five `nop`s to determine the amount of time per cycle. You should be able to determine the number of cycles by looking at the inline assembly instructions and comparing to the number of cycles required in Apx. C.

11.6 From your measurements to problem 11.5, how fast do you calculate your processor clock to be? How much time per cycle?

11.7 Modify the source code in problem 11.5 to measure the following (just remove the middle `asm volatile` instruction – you need to leave the first and third in order to get the LED to flash):

1. Cycles for an 8-bit addition/subtraction of two variables;
2. Cycles for an 8-bit multiplication of two variables (that is, two 8-bit numbers multiplied together to result in a 16-bit number);
3. Cycles for an 8-bit division of two variables;
4. Cycles for an 8-bit division of one variable by a constant power of 2;
5. Cycles for an 8-bit division of one variable by a constant value other than a power of 2;
6. Cycles for a 32-bit addition/subtraction of two variables;
7. Cycles for a 32-bit multiplication of two variables (that is, two 16-bit numbers multiplied together to result in a 32-bit number);
8. Cycles for a 32-bit division of two variables (that is, a 32-bit integer divided by some other integer);
9. Cycles for a 32-bit division of one variable by a constant power of 2;
10. Cycles for a 32-bit division of one variable by a constant value other than a power of 2;
11. Cycles for a floating point addition/subtraction of two variables;
12. Cycles for a floating point multiplication of two variables;
13. Cycles for a floating point division of two variables.

11.8 From your measurements to problem 11.7, state your estimations for number of cycles per each high-level operation.

11.9 From your measurements to problem 11.7, state the number of bytes for each program.

11.10 From your measurements to problem 11.7, what was the most expensive high-level instruction?

11.11 From your measurements to problem 11.7, was there a difference between the different integer divisions? Explain.

11.12 From your measurements to problem 11.7, did you see a correlation between the number of cycles and the number of bytes for each program? Explain.

CHAPTER 12

Non-volatile Memory

12.1 INTRODUCTION

Many useful embedded systems require some kind of non-volatile memory for holding information other than that of the code-space. Some of the most recent, common examples include smart phones and portable music players. Smart phones such as Apple's iPhone or Motorola's Droid are digital cell phones with the ability to run "apps" (short for applications). These apps are novel because they are purchased and downloaded electronically, and then stored on the phone for subsequent usage. These programs have to be saved in non-volatile memory; otherwise, you would lose the program every time you turned off the phone or ran out of battery life. Similarly, the portable music players such as Apple's iPod connect to a host computer in order to download music files that are stored on the portable device for subsequent listening. Again, the files need to be stored in non-volatile memory. Both of these example systems require a very large amount of non-volatile memory since apps and music are megabytes in size. It turns out that flash memory serves this purpose well. In fact, there are portable electronic devices being created with gigabytes of flash storage used as the physical medium of the file systems for real-time operating systems such as Embedded Linux. The RTOS is able to treat the flash memory as though it were a hard-disk such that it can store files, just like on your host computer.

This is all well and good for large-scale embedded systems with microprocessors, but you might be wondering what good a lot of flash would do a microcontroller, which is generally resource limited in many ways. It turns out that many embedded systems have the need to store small pieces of data, on the order of kilobytes, in non-volatile memory, such that flash sizes are overkill. The other primary non-volatile memory is the Electrically Erasable/Programmable Read Only Memory (*EEPROM* or E^2 – pronounced "e-squared"), which is often manufactured in much smaller sizes (e.g., 8 KB to 32 KB). Then, embedded systems will connect an EEPROM to a microcontroller, usually via a serial bus, in order to save important operating parameters. As an example of the need, consider a cell phone in which you can adjust the backlight of the display, or the volume of the ring-tone, or vibration mode. After you make such adjustments, the settings you choose are stored in an EEPROM. Consider if they weren't, and you remove your battery. If they were not saved, all of your settings would have been lost.

12.1.1 EEPROM VIA C ON ATMEGA328P

A very nice feature of the ATmega328P is the inclusion of an on-chip 1 KB EEPROM and an associated peripheral interface used to read and write information from and to it. It turns out that

you can't just read/write bytes of information from/to the EEPROM like you would SRAM because of the difference in access times; that is, non-volatile memories generally take more time to access, especially when writing information to them. So, there are flags necessary for indicating when information has been transferred properly. Software then needs to monitor the associated flags in order to prevent any kind of corruption. The following function for writing to the ATmega328P on-chip EEPROM was modified from ATMEL (2009).

Example 12.1

```
#define EEARH_ADDR                (unsigned char *) 0x42
#define EEARL_ADDR                (unsigned char *) 0x41
#define EEDR_ADDR                 (unsigned char *) 0x40
#define EECR_ADDR                 (unsigned char *) 0x3F

#define EECR_EEPE_MASK            0x02
#define EECR_EEMPE_MASK           0x04

void EepromWrite(unsigned short address, unsigned char data)
{
   unsigned char *portEEPROMControlRegister;
   unsigned char *portEEPROMAddressHighRegister;
   unsigned char *portEEPROMAddressLowRegister;
   unsigned char *portEEPROMDataRegister;

   portEEPROMControlRegister = EECR_ADDR;
   portEEPROMAddressHighRegister = EEARH_ADDR;
   portEEPROMAddressLowRegister = EEARL_ADDR;
   portEEPROMDataRegister = EEDR_ADDR;

   /* Wait for completion of previous write */
   while(*portEEPROMControlRegister & EECR_EEPE_MASK);

   /* Set up address and Data Registers */
   *portEEPROMAddressHighRegister = ((address >> 8) & 0x03);
   *portEEPROMAddressLowRegister = (address & 0xFF);
   *portEEPROMDataRegister = data;

   /* Write logical one to EEMPE */
   *portEEPROMControlRegister |= EECR_EEMPE_MASK;

   /* Start eeprom write by setting EEPE */
   *portEEPROMControlRegister |= EECR_EEPE_MASK;
}
```

Notice that the 10-bit address is split across two 8-bit registers. You might be tempted to create a pointer to an `unsigned short` in order to load the entire address in one C instruction.

However, this is not possible because the memory-mapped address of the high-byte address register is 0x42 while the low-byte address register is 0x41. In order to have a 16-bit access, the two addresses need to fall on a 16-bit boundary; i.e., the address bits need to be identical except for the LSB. This is not the case since the two LSBs differ in 0x42 and 0x41. The result is that each address register must be loaded individually.

The following function for reading from the ATmega328P on-chip EEPROM was modified from ATMEL (2009).

Example 12.2

```
#define EEARH_ADDR                (unsigned char *) 0x42
#define EEARL_ADDR                (unsigned char *) 0x41
#define EEDR_ADDR                 (unsigned char *) 0x40
#define EECR_ADDR                 (unsigned char *) 0x3F

#define EECR_EEPE_MASK            0x02
#define EECR_EERE_MASK            0x01

unsigned char EepromRead(unsigned short address)
{
   unsigned char *portEEPROMControlRegister;
   unsigned char *portEEPROMAddressHighRegister;
   unsigned char *portEEPROMAddressLowRegister;
   unsigned char *portEEPROMDataRegister;
   unsigned char data;

   portEEPROMControlRegister = EECR_ADDR;
   portEEPROMAddressHighRegister = EEARH_ADDR;
   portEEPROMAddressLowRegister = EEARL_ADDR;
   portEEPROMDataRegister = EEDR_ADDR;

   /* Wait for completion of previous write */
   while(*portEEPROMControlRegister & EECR_EEPE_MASK);

   /* Set up address register */
   *portEEPROMAddressHighRegister = ((address >> 8) & 0x03);
   *portEEPROMAddressLowRegister = (address & 0xFF);

   /* Start eeprom read by writing EERE */
   *portEEPROMControlRegister |= EECR_EERE_MASK;

   /* Return data from Data Register */
   data = *portEEPROMDataRegister;

   return data;
}
```

12.2 PERTINENT REGISTER DESCRIPTIONS

The information presented in this section was taken from ATMEL (2009).

12.2.1 EEARH - EEPROM HIGH ADDRESS REGISTER

Bit	7	6	5	4	3	2	1	0
0x42	-	-	-	-	-	-	EEAR9	EEAR8
Read/Write	R	R	R	R	R	R	R/W	R/W
Default	0	0	0	0	0	0	-	-

- EEAR9-8: EEPROM Address Bit. The EEPROM Address Registers `EEARH` and `EEARL` specify the EEPROM address in the 1KB EEPROM space. The EEPROM data bytes are addressed linearly between 0 and 1023. The initial value of `EEAR9:8` is undefined. A proper value must be written before the EEPROM may be accessed.

12.2.2 EEARL - EEPROM LOW ADDRESS REGISTER

Bit	7	6	5	4	3	2	1	0
0x41	EEAR7	EEAR6	EEAR5	EEAR4	EEAR3	EEAR2	EEAR1	EEAR0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	-	-	-	-	-	-	-	-

- EEAR7-0: EEPROM Address Bits. The EEPROM Address Registers `EEARH` and `EEARL` specify the EEPROM address in the 1KB EEPROM space. The EEPROM data bytes are addressed linearly between 0 and 1023. The initial value of `EEAR7:0` is undefined. A proper value must be written before the EEPROM may be accessed.

12.2.3 EEDR - EEPROM DATA REGISTER

Bit	7	6	5	4	3	2	1	0
0x40	EEDR7	EEDR6	EEDR5	EEDR4	EEDR3	EEDR2	EEDR1	EEDR0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

- EEDR7-0: EEPROM Data Bits. For the EEPROM write operation, the `EEDR` Register contains the data to be written to the EEPROM in the address given by the `EEAR` Register. For the EEPROM read operation, the `EEDR` contains the data read out from the EEPROM at the address given by `EEAR`.

12.2.4 EECR - EEPROM CONTROL REGISTER

Bit	7	6	5	4	3	2	1	0
0x3F	-	-	EEPM1	EEPM0	EERIE	EEMPE	EEPE	EERE
Read/Write	R	R	R/W	R/W	R/W	R/W	R/W	R/W
Default	0	0	-	-	0	0	-	0

- EEPM1-0: EEPROM Programming Mode Bits. The EEPROM Programming mode bit setting defines which programming action that will be triggered when writing EEPE. It is possible to program data in one atomic operation (erase the old value and program the new value) or to split the Erase and Write operations in two different operations. The programming times for the different modes are shown in Table 12.1. While EEPE is set, any write to EEPM*x* will be ignored. During reset, the EEPM*x* bits will be reset to 00 unless the EEPROM is busy programming.

Table 12.1: EEPROM Mode Bits

EEPM1-0	Programming Time	Operation
00	3.4 msec	Erase and Write in one atomic operation
01	1.8 msec	Erase only
10	1.8 msec	Write only
11	-	Reserved

- EEMPE: EEPROM Master Write Enable. The EEMPE bit determines whether setting EEPE to one causes the EEPROM to be written. When EEMPE is set, setting EEPE within four clock cycles will write data to the EEPROM at the selected address. If EEMPE is zero, setting EEPE will have no effect. When EEMPE has been written to one by software, hardware clears the bit to zero after four clock cycles.

- EEPE: EEPROM Write Enable. The EEPROM Write Enable Signal EEPE is the write strobe to the EEPROM. When address and data are correctly set up, the EEPE bit must be written to one to write the value into the EEPROM. The EEMPE bit must be written to one before a logical one is written to EEPE; otherwise, no EEPROM write takes place. The procedure listed in Ex. 12.1 should be followed when writing the EEPROM.

- EERE: EEPROM Read Enable. The EEPROM Read Enable Signal EERE is the read strobe to the EEPROM. When the correct address is set up in the EEAR Register, the EERE bit must be written to a logic one to trigger the EEPROM read. The EEPROM read access takes one instruction, and the requested data is available immediately. When the EEPROM is read, the CPU is halted for four cycles before the next instruction is executed.

  The user should poll the EEPE bit before starting the read operation. If a write operation is in progress, it is neither possible to read the EEPROM nor to change the EEAR Register.

The procedure listed in Ex. 12.2 should be followed when reading from the EEPROM.

- See Ch. 9 for interrupt-related bit descriptions.

PROBLEMS

12.1 Pick an address in the EEPROM (i.e., $addr \in \{0, \ldots, 1023\}$) to hold a special byte of data used to indicate the presence of stored data. Create a program that writes a special byte (e.g., `0xAA`) to the EEPROM.

12.2 Create a program that clears the byte stored at the address in problem 12.1. That is, write `0x00` to the location.

12.3 Create a program that reads the byte stored at the address in problem 12.1. If the byte is equal to the special byte used in problem 12.1, then read out the stored value for use as the initial value in the rest of the program. If the byte is not equal to the special value, use a pre-defined constant value as the initial value in the rest of the program.

For the rest of the program, given the initial value between 0 and 9, repeatedly output the digits '0' through '9' on a seven-segment display with a one-second interval between each digit. After each digit is displayed, save the current digit to a location in the EEPROM, and be sure to write the special byte to the address determined in problem 12.1.

Verify that when you reset the board, the value is read out of the EEPROM and continues counting from where it left off.

12.4 Use the program from problem 12.2 to clear the EEPROM. Verify that the program in problem 12.3 starts at the default value after the EEPROM is cleared.

APPENDIX A

Arduino 2009 Schematic

Table A.1: ATmega328P Pin Configurations

Pin #	Arduino Connection	GPIO	Alternate Functions	Description
1	RESET	PC6	RESET	Processor reset signal
			PCINT14	Pin change interrupt 14
2	J1-1 "0"	PD0	RXD	USART receive
			PCINT16	Pin change interrupt 16
3	J1-2 "1"	PD1	TXD	USART transmit
			PCINT17	Pin change interrupt 17
4	J1-3 "2"	PD2	INT0	External interrupt request 0
			PCINT18	Pin change interrupt 18
5	J1-4 "3"	PD3	INT1	External interrupt request 1
			PCINT19	Pin change interrupt 19
			OC2B	Output compare pin, Timer2 Unit B
6	J1-5 "4"	PD4	T0	Timer0 external counter input
			PCINT20	Pin change interrupt 20
			XCK	USART external clock I/O
7	+5V	-	VCC	Power
8	GND	-	GND	Ground
9	16MHz Crystal	PB6	XTAL1	Chip clock oscillator pin 1
			PCINT6	Pin change interrupt 6
			TOSC1	Timer oscillator pin 1
10	16MHz Crystal	PB7	XTAL2	Chip clock oscillator pin 2
			PCINT7	Pin change interrupt 7
			TOSC2	Timer oscillator pin 2
11	J1-6 "5"	PD5	T1	Timer1 external counter input
			PCINT21	Pin change interrupt 21
			OC0B	Output compare pin, Timer0 Unit B
12	J1-7 "6"	PD6	AIN0	Analog comparator positive input
			PCINT22	Pin change interrupt 22
			OC0A	Output compare pin, Timer0 Unit A

Table A.1: ATmega328P Pin Configurations

Pin #	Arduino Connection	GPIO	Alternate Functions	Description
13	J1-8 "7"	PD7	AIN1	Analog comparator negative input
			PCINT23	Pin change interrupt 23
14	J3-1 "8"	PB0	ICP1	Timer1 input capture input
			PCINT0	Pin change interrupt 0
			CLKO	Divided system clock output
15	J3-2 "9"	PB1	OC1A	Output compare pin, Timer1 Unit A
			PCINT1	Pin change interrupt 1
16	J3-3 "10"	PB2	OC1B	Output compare pin, Timer1 Unit B
			PCINT2	Pin change interrupt 2
			SS	SPI slave select
17	J3-4 "11"	PB3	OC2A	Output compare pin, Timer2 Unit A
			PCINT3	Pin change interrupt 3
			MOSI	SPI master-out/slave-in
18	J3-5 "12"	PB4	PCINT4	Pin change interrupt 4
			MISO	SPI master-in/slave-out
19	J3-6 "13"	PB5	PCINT5	Pin change interrupt 5
			SCK	SPI clock input
20	+5V	-	AVCC	Analog supply voltage for ADC
21	J3-8 "AREF"	-	AREF	External ADC reference voltage
22	GND	-	GND	Ground
23	J2-1 "0"	PC0	ADC0	ADC input channel 0
			PCINT8	Pin change interrupt 8
24	J2-2 "1"	PC1	ADC1	ADC input channel 1
			PCINT9	Pin change interrupt 9
25	J2-3 "2"	PC2	ADC2	ADC input channel 2
			PCINT10	Pin change interrupt 10
26	J2-4 "3"	PC3	ADC3	ADC input channel 3
			PCINT11	Pin change interrupt 11
27	J2-5 "4"	PC4	ADC4	ADC input channel 4
			PCINT12	Pin change interrupt 12
			SDA	TWI serial data
28	J2-6 "5"	PC5	ADC5	ADC input channel 5
			PCINT13	Pin change interrupt 13
			SCL	TWI serial clock

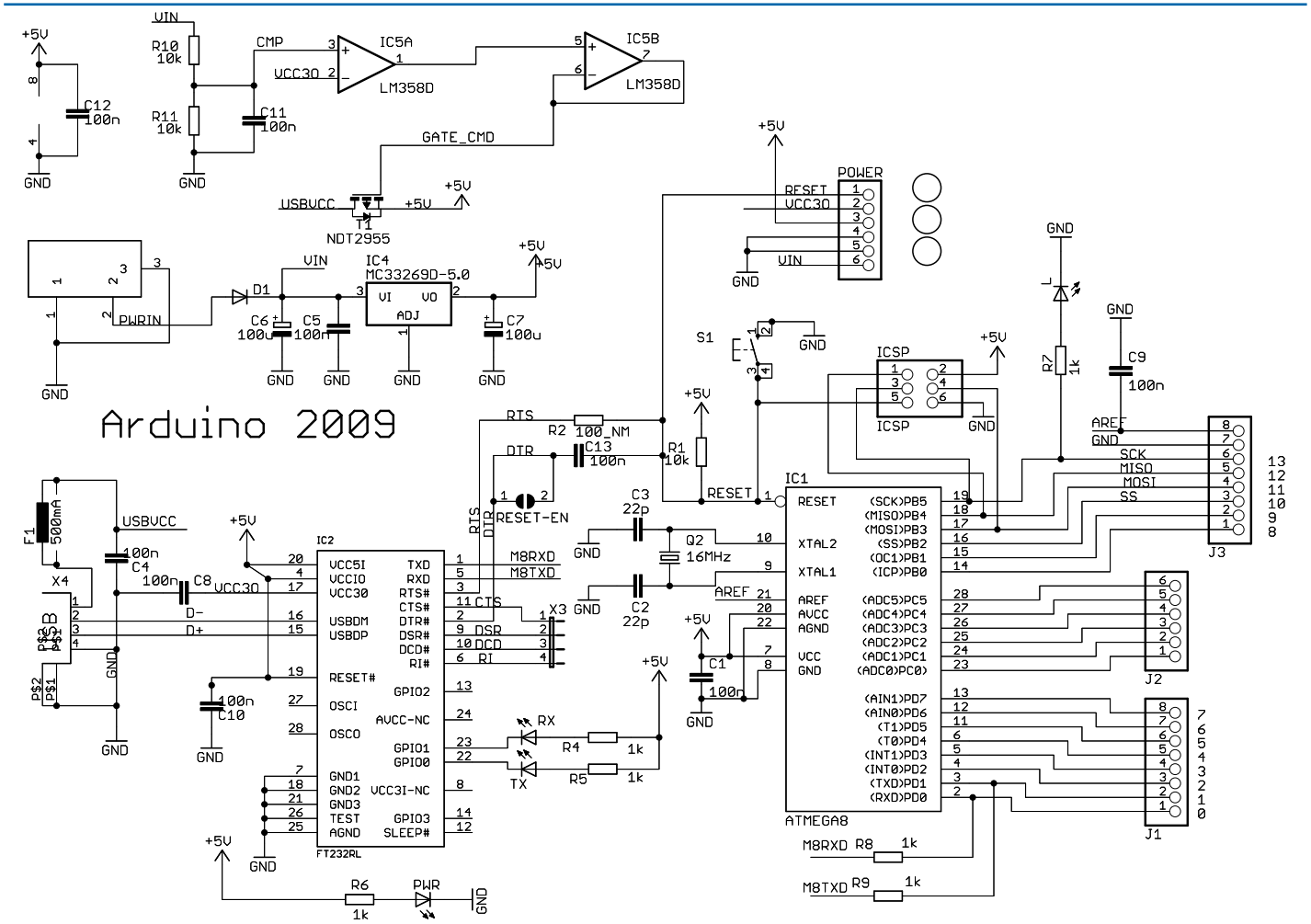


Figure A.1: Arduino Duemilanove ("2009") schematic.

Source: Arduino, http://www.arduino.cc ©2009 Arduino. Distributable under a Creative Commons Attribution-ShareAlike 3.0 License, http://creativecommons.org/licenses/by-sa/3.0/.

APPENDIX B

ATmega328P Registers

B.1 REGISTER SUMMARY

Table B.1: Memory Mapped Register Summary

		Bit							
Addr	Name	7	6	5	4	3	2	1	0
0x00-1F	R0-R31	CPU General Purpose Working Registers mapped to data space							
0x20-22	Reserved	-	-	-	-	-	-	-	-
0x23	PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0
0x24	DDRB	DDRB7	DDRB6	DDRB5	DDRB4	DDRB3	DDRB2	DDRB1	DDRB0
0x25	PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0
0x26	PINC	-	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0
0x27	DDRC	-	DDRC6	DDRC5	DDRC4	DDRC3	DDRC2	DDRC1	DDRC0
0x28	PORTC	-	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0
0x29	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0
0x2A	DDRD	DDRD7	DDRD6	DDRD5	DDRD4	DDRD3	DDRD2	DDRD1	DDRD0
0x2B	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0
0x2C-34	Reserved	-	-	-	-	-	-	-	-
0x35	TIFR0	-	-	-	-	-	OCF0B	OCF0A	TOV0
0x36	TIFR1	-	-	ICF1	-	-	OCF1B	OCF1A	TOV1
0x37	TIFR2	-	-	-	-	-	OCF2B	OCF2A	TOV2
0x38-3A	Reserved	-	-	-	-	-	-	-	-
0x3B	PCIFR	-	-	-	-	-	PCIF2	PCIF1	PCIF0
0x3C	EIFR	-	-	-	-	-	-	INTF1	INTF0
0x3D	EIMSK	-	-	-	-	-	-	INT1	INT0
0x3E	GPIOR0	General Purpose I/O Register 0							
0x3F	EECR	-	-	EEPM1	EEPM0	EERIE	EEMPE	EEPE	EERE
0x40	EEDR	EEPROM Data Register							
0x41	EEARL	EEPROM Address Register Low Byte							
0x42	EEARH	EEPROM Address Register High Byte							
0x43	GTCCR	TSM	-	-	-	-	-	PSRASY	PSRSYN
0x44	TCCR0A	COM0A1	COM0A0	COM0B1	COM0B0	-	-	WGM01	WGM00
0x45	TCCR0B	FOC0A	FOC0B	-	-	WGM02	CS02	CS01	CS00

Table B.1: Memory Mapped Register Summary

		Bit							
Addr	Name	7	6	5	4	3	2	1	0
0x46	TCNT0	Timer/Counter0							
0x47	OCR0A	Timer/Counter0 Output Compare Register A							
0x48	OCR0B	Timer/Counter0 Output Compare Register B							
0x49	Reserved	-	-	-	-	-	-	-	-
0x4A	GPIOR1	General Purpose I/O Register 1							
0x4B	GPIOR2	General Purpose I/O Register 2							
0x4C	SPCR	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0
0x4D	SPSR	SPIF	WCOL	-	-	-	-	-	SPI2X
0x4E	SPDR	SPI Data Register							
0x4F	Reserved	-	-	-	-	-	-	-	-
0x50	ACSR	ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0
0x51-52	Reserved	-	-	-	-	-	-	-	-
0x53	SMCR	-	-	-	-	SM2	SM1	SM0	SE
0x54	MCUSR	-	-	-	-	WDRF	BORF	EXTRF	PORF
0x55	MCUCR	-	BODS	BODSE	PUD	-	-	IVSEL	IVCE
0x56	Reserved	-	-	-	-	-	-	-	-
0x57	SPMCSR	SPMIE	RWWSB	-	RWWSRE	BLBSET	PGWRT	PGERS	SLFPRG
0x58-5C	Reserved	-	-	-	-	-	-	-	-
0x5D	SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0
0x5E	SPH	-	-	-	-	-	SP10	SP9	SP8
0x5F	SREG	I	T	H	S	V	N	Z	C
0x60	WDTCSR	WDIF	WDIE	WDP3	WDCE	WDE	WDP2	WDP1	WDP0
0x61	CLKPR	CLKPCE	-	-	-	CLKPS3	CLKPS2	CLKPS1	CLKPS0
0x62-63	Reserved	-	-	-	-	-	-	-	-
0x64	PRR	PRTWI	PRTIM2	PRTIM0	-	PRTIM1	PRSPI	PRUSART0	PRADC
0x65	Reserved	-	-	-	-	-	-	-	-
0x66	OSCCAL	Oscillator Calibration Register							
0x67	Reserved	-	-	-	-	-	-	-	-
0x68	PCICR	-	-	-	-	-	PCIE2	PCIE1	PCIE0
0x69	EICRA	-	-	-	-	ISC11	ISC10	ISC01	ISC00
0x6A	Reserved	-	-	-	-	-	-	-	-
0x6B	PCMSK0	PCINT7	PCINT6	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0
0x6C	PCMSK1	-	PCINT14	PCINT13	PCINT12	PCINT11	PCINT10	PCINT9	PCINT8
0x6D	PCMSK2	PCINT23	PCINT22	PCINT21	PCINT20	PCINT19	PCINT18	PCINT17	PCINT16
0x6E	TIMSK0	-	-	-	-	-	OCIE0B	OCIE0A	TOIE0

Table B.1: Memory Mapped Register Summary

		Bit							
Addr	Name	7	6	5	4	3	2	1	0
0x6F	TIMSK1	-	-	ICIE1	-	-	OCIE1B	OCIE1A	TOIE1
0x70	TIMSK2	-	-	-	-	-	OCIE2B	OCIE2A	TOIE2
0x71-77	Reserved	-	-	-	-	-	-	-	-
0x78	ADCL	ADC Data Register Low Byte							
0x79	ADCH	ADC Data Register High Byte							
0x7A	ADCSRA	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
0x7B	ADCSRB	-	ACME	-	-	-	ADTS2	ADTS1	ADTS0
0x7C	ADMUX	REFS1	REFS0	ADLAR	-	MUX3	MUX2	MUX1	MUX0
0x7D	Reserved	-	-	-	-	-	-	-	-
0x7E	DIDR0	-	-	ADC5D	ADC4D	ADC3D	ADC2D	ADC1D	ADC0D
0x7F	DIDR1	-	-	-	-	-	-	AIN1D	AIN0D
0x80	TCCR1A	COM1A1	COM1A0	COM1B1	COM1B0	-	-	WGM11	WGM10
0x81	TCCR1B	ICNC1	ICES1	-	WGM13	WGM12	CS12	CS11	CS10
0x82	TCCR1C	FOC1A	FOC1B	-	-	-	-	-	-
0x83	Reserved	-	-	-	-	-	-	-	-
0x84	TCNT1L	Timer/Counter1 - Counter Register Low Byte							
0x85	TCNT1H	Timer/Counter1 - Counter Register High Byte							
0x86	ICR1L	Timer/Counter1 - Input Capture Register Low Byte							
0x87	ICR1H	Timer/Counter1 - Input Capture Register High Byte							
0x88	OCR1AL	Timer/Counter1 - Output Compare Register A Low Byte							
0x89	OCR1AH	Timer/Counter1 - Output Compare Register A High Byte							
0x8A	OCR1BL	Timer/Counter1 - Output Compare Register B Low Byte							
0x8B	OCR1BH	Timer/Counter1 - Output Compare Register B High Byte							
0x8C-AF	Reserved	-	-	-	-	-	-	-	-
0xB0	TCCR2A	COM2A1	COM2A0	COM2B1	COM2B0	-	-	WGM21	WGM20
0xB1	TCCR2B	FOC2A	FOC2B	-	-	WGM22	CS22	CS21	CS20
0xB2	TCNT2	Timer/Counter2							
0xB3	OCR2A	Timer/Counter2 Output Compare Register A							
0xB4	OCR2B	Timer/Counter2 Output Compare Register B							
0xB5	Reserved	-	-	-	-	-	-	-	-
0xB6	ASSR	-	EXCLK	AS2	TCN2UB	OCR2AUB	OCR2BUB	TCR2AUB	TCR2BUB
0xB7	Reserved	-	-	-	-	-	-	-	-
0xB8	TWBR	2-Wire Serial Interface Bit Rate Register							
0xB9	TWSR	TWS7	TWS6	TWS5	TWS4	TWS3	-	TWS1	TWS0
0xBA	TWAR	TWA6	TWA5	TWA4	TWA3	TWA2	TWA1	TWA0	TWGCE

Table B.1: Memory Mapped Register Summary

		Bit							
Addr	Name	7	6	5	4	3	2	1	0
0xBB	TWDR	2-Wire Serial Interface Data Register							
0xBC	TWCR	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE
0xBD	TWAMR	TWAM6	TWAM5	TWAM4	TWAM3	TWAM2	TWAM1	TWAM0	-
0xBE-BF	Reserved	-	-	-	-	-	-	-	-
0xC0	UCSR0A	RXC0	TXC0	UDRE0	FE0	DOR0	UPE0	U2X0	MPCM0
0xC1	UCSR0B	RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	UCSZ02	RXB80	TXB80
0xC2	UCSR0C	UMSEL01	UMSEL00	UPM01	UPM00	USBS0	UCSZ01	UCSZ00	UCPOL0
0xC3	Reserved	-	-	-	-	-	-	-	-
0xC4	UBRR0L	USART Baud Rate Register Low							
0xC5	UBRR0H	-	-	-	-	USART Baud Rate Register High			
0xC6	UDR0	USART I/O Data Register							
0xC7-FF	Reserved	-	-	-	-	-	-	-	-

Source: ATMEL, "8-bit AVR Microcontroller with 4/8/16/32K Bytes In-System Programmable Flash," Rev. 8161D - 10/09 ©2009 Atmel Corporation.

APPENDIX C

ATmega328P Assembly Instructions

C.1 INSTRUCTION SET SUMMARY

Table C.1: Arithmetic Instructions

Mnemonic	Description	Operation	Affected SREG Bits	Clk Cyc
ADD Rd,Rr	Add two registers	Rd ← Rd + Rr	-,-,H,-,V,N,Z,C	1
ADC Rd,Rr	Add with carry two registers	Rd ← Rd + Rr +SREG[C]	-,-,H,-,V,N,Z,C	1
ADIW Rdl,K	Add immediate to word	Rdh:Rdl ←Rdh:Rdl + K	-,-,-,S,V,N,Z,C	2
SUB Rd,Rr	Subtract two registers	Rd ← Rd - Rr	-,-,H,-,V,N,Z,C	1
SUBI Rd,K	Subtract constant from register	Rd ← Rd - K	-,-,H,-,V,N,Z,C	1
SBC Rd,Rr	Subtract with carry two registers	Rd ← Rd - Rr -SREG[C]	-,-,H,-,V,N,Z,C	1
SBCI Rd,K	Subtract with carry immediate from reg.	Rd ← Rd - K -SREG[C]	-,-,H,-,V,N,Z,C	1
SBIW Rdl,K	Subtract immediate from word	Rdh:Rdl ←Rdh:Rdl - K	-,-,-,S,V,N,Z,C	2
NEG Rd	Two's complement negation register	Rd ← 0x00 - Rd	-,-,H,-,V,N,Z,C	1
INC Rd	Increment register	Rd ← Rd + 1	-,-,-,-,V,N,Z,-	1
DEC Rd	Decrement register	Rd ← Rd - 1	-,-,-,-,V,N,Z,-	1
TST Rd	Test register for zero of minus	Rd ← Rd ∩ Rd	-,-,-,-,V,N,Z,-	1
MUL Rd,Rr	Multiply unsigned two registers	R1:R0 ←Rd × Rr	-,-,-,-,-,-,Z,C	2
MULS Rd,Rr	Multiply signed two registers	R1:R0 ←Rd × Rr	-,-,-,-,-,-,Z,C	2
MULSU Rd,Rr	Multiply signed with unsigned registers	R1:R0 ←Rd × Rr	-,-,-,-,-,-,Z,C	2
FMUL Rd,Rr	Fractional multiply unsigned two registers	R1:R0 ←(Rd × Rr) ≪ 1	-,-,-,-,-,-,Z,C	2
FMULS Rd,Rr	Frac. mult. signed two registers	R1:R0 ←(Rd × Rr) ≪ 1	-,-,-,-,-,-,Z,C	2
FMULSU Rd,Rr	Frac. mult. signed with unsigned registers	R1:R0 ←(Rd × Rr) ≪ 1	-,-,-,-,-,-,Z,C	2

Source: ATMEL, "8-bit AVR Microcontroller with 4/8/16/32K Bytes In-System Programmable Flash," Rev. 8161D - 10/09 ©2009 Atmel Corporation.

Table C.2: Logic Instructions

Mnemonic	Description	Operation	Affected SREG Bits	Clk Cyc
AND Rd,Rr	Logical AND two registers	Rd ← Rd ∩ Rr	-,-,-,-,V,N,Z,-	1
ANDI Rd,K	Logical AND immediate and register	Rd ← Rd ∩ K	-,-,-,-,V,N,Z,-	1
OR Rd,Rr	Logical OR two registers	Rd ← Rd ∪ Rr	-,-,-,-,V,N,Z,-	1
ORI Rd,K	Logical OR constant and register	Rd ← Rd ∪ K	-,-,-,-,V,N,Z,-	1
EOR Rd,Rr	Exclusive OR two registers	Rd ← Rd ⊕ Rr	-,-,-,-,V,N,Z,-	1
COM Rd	One's complement negation register	Rd ← 0xFF - Rd	-,-,-,-,V,N,Z,C	1
SBR Rd,K	Set bit(s) in register	Rd ← Rd ∪ K	-,-,-,-,V,N,Z,-	1
CBR Rd,K	Clear bit(s) in register	Rd ← Rd ∩ (0xFF - K)	-,-,-,-,V,N,Z,-	1
CLR Rd	Clear register	Rd ← Rd ⊕ Rd	-,-,-,-,V,N,Z,-	1
SER Rd	Set register	Rd ← 0xFF	-,-,-,-,-,-,-,-	1

Source: ATMEL, "8-bit AVR Microcontroller with 4/8/16/32K Bytes In-System Programmable Flash," Rev. 8161D - 10/09 ©2009 Atmel Corporation.

Table C.3: Branch Instructions

Mnemonic	Description	Operation	Affected SREG Bits	Clk Cyc
RJMP k	Relative jump	PC ← PC + k + 1	-,-,-,-,-,-,-,-	2
IJMP	Indirect jump to (Z)	PC ← Z	-,-,-,-,-,-,-,-	2
JMP k	Direct jump	PC ← k	-,-,-,-,-,-,-,-	3
RCALL k	Relative subroutine call	(SP) ← PC + S SP ← SP - 2 PC ← PC + k + 1	-,-,-,-,-,-,-,-	3
ICALL	Indirect call to (Z)	(SP) ← PC + S SP ← SP - 2 PC ← Z	-,-,-,-,-,-,-,-	3
CALL k	Direct subroutine call	(SP) ← PC + S SP ← SP - 2 PC ← k	-,-,-,-,-,-,-,-	4
RET	Subroutine return	SP ← SP + 2 PC ← (SP)	-,-,-,-,-,-,-,-	4
RETI	Interrupt subroutine return	SP ← SP + 2 PC ← (SP)	I,-,-,-,-,-,-,-	4
CPSE Rd,Rr	Compare, skip if equal	PC ← PC + 1 if (Rd=Rr)then PC ← PC + N	-,-,-,-,-,-,-,-	1,2 or 3

Table C.3: Branch Instructions

Mnemonic	Description	Operation	Affected SREG Bits	Clk Cyc
CP Rd,Rr	Compare two registers	Rd - Rr	-,-,H,-,V,N,Z,C	1
CPC Rd,Rr	Compare with carry two registers	Rd - Rr - SREG[C]	-,-,H,-,V,N,Z,C	1
CPI Rd,K	Compare immediate and register	Rd - K	-,-,H,-,V,N,Z,C	1
SBRC Rr,b	Skip if bit in register cleared	PC ← PC + 1 if (Rr[b]=0)then PC ← PC + N	-,-,-,-,-,-,-,-	1,2 or 3
SBRS Rr,b	Skip if bit in register set	PC ← PC + 1 if (Rr[b]=1)then PC ← PC + N	-,-,-,-,-,-,-,-	1,2 or 3
SBIC P,b	Skip if bit in I/O register cleared	PC ← PC + 1 if (P[b]=0)then PC ← PC + N	-,-,-,-,-,-,-,-	1,2 or 3
SBIS P,b	Skip if bit in I/O register set	PC ← PC + 1 if (P[b]=1)then PC ← PC + N	-,-,-,-,-,-,-,-	1,2 or 3
BRBC b,k	Branch if status bit cleared	PC ← PC + 1 if (SREG[b]=0)then PC ← PC + k	-,-,-,-,-,-,-,-	1,2
BRBS b,k	Branch if status bit set	PC ← PC + 1 if (SREG[b]=1)then PC ← PC + k	-,-,-,-,-,-,-,-	1,2
BRNE k	Branch if not equal	PC ← PC + 1 if (SREG[Z]=0)then PC ← PC + k	-,-,-,-,-,-,-,-	1,2
BREQ k	Branch if equal	PC ← PC + 1 if (SREG[Z]=1)then PC ← PC + k	-,-,-,-,-,-,-,-	1,2
BRCC k	Branch if carry cleared	PC ← PC + 1 if (SREG[C]=0)then PC ← PC + k	-,-,-,-,-,-,-,-	1,2
BRCS k	Branch if carry set	PC ← PC + 1 if (SREG[C]=1)then PC ← PC + k	-,-,-,-,-,-,-,-	1,2
BRSH k	Branch if same or higher	PC ← PC + 1 if (SREG[C]=0)then PC ← PC + k	-,-,-,-,-,-,-,-	1,2

Table C.3: Branch Instructions

Mnemonic	Description	Operation	Affected SREG Bits	Clk Cyc
BRLO k	Branch if lower	PC ← PC + 1 if (SREG[C]=1)then PC ← PC + k	-,-,-,-,-,-,-,-	1,2
BRPL k	Branch if positive	PC ← PC + 1 if (SREG[N]=0)then PC ← PC + k	-,-,-,-,-,-,-,-	1,2
BRMI k	Branch if negative	PC ← PC + 1 if (SREG[N]=1)then PC ← PC + k	-,-,-,-,-,-,-,-	1,2
BRGE k	Branch if signed greater or equal	PC ← PC + 1 if (SREG[S]=0)then PC ← PC + k	-,-,-,-,-,-,-,-	1,2
BRLT k	Branch if signed less than zero	PC ← PC + 1 if (SREG[S]=1)then PC ← PC + k	-,-,-,-,-,-,-,-	1,2
BRHC k	Branch if half-carry cleared	PC ← PC + 1 if (SREG[H]=0)then PC ← PC + k	-,-,-,-,-,-,-,-	1,2
BRHS k	Branch if half-carry set	PC ← PC + 1 if (SREG[H]=1)then PC ← PC + k	-,-,-,-,-,-,-,-	1,2
BRTC k	Branch if bit copy storage cleared	PC ← PC + 1 if (SREG[T]=0)then PC ← PC + k	-,-,-,-,-,-,-,-	1,2
BRTS k	Branch if bit copy storage set	PC ← PC + 1 if (SREG[T]=1)then PC ← PC + k	-,-,-,-,-,-,-,-	1,2
BRVC k	Branch if overflow cleared	PC ← PC + 1 if (SREG[V]=0)then PC ← PC + k	-,-,-,-,-,-,-,-	1,2
BRVS k	Branch if overflow set	PC ← PC + 1 if (SREG[V]=1)then PC ← PC + k	-,-,-,-,-,-,-,-	1,2
BRID k	Branch if interrupts disabled	PC ← PC + 1 if (SREG[I]=0)then PC ← PC + k	-,-,-,-,-,-,-,-	1,2

Table C.3: Branch Instructions

Mnemonic	Description	Operation	Affected SREG Bits	Clk Cyc
BRIE k	Branch if interrupts enabled	PC ← PC + 1 if (SREG[I]=1)then PC ← PC + k	-,-,-,-,-,-,-,-	1,2

Source: ATMEL, "8-bit AVR Microcontroller with 4/8/16/32K Bytes In-System Programmable Flash," Rev. 8161D - 10/09 ©2009 Atmel Corporation.

Table C.4: Bit and Bit-Test Instructions

Mnemonic	Description	Operation	Affected SREG Bits	Clk Cyc
SBI P,b	Set bit in I/O register	P[b] ←1	-,-,-,-,-,-,-,-	2
CBI P,b	Clear bit in I/O register	P[b] ←0	-,-,-,-,-,-,-,-	2
LSL Rd	Logical shift left register (i.e., multiply-by-2)	Rd[n] ←Rd[n-1] Rd[0] ←0	-,-,-,-,V,N,Z,C	1
LSR Rd	Logical shift right register (i.e., unsigned divide-by-2)	Rd[n-1] ←Rd[n] Rd[7] ←0	-,-,-,-,V,N,Z,C	1
ROL Rd	Rotate left via carry register	Rd[0] ←SREG[C] Rd[n] ←Rd[n-1] SREG[C] ←Rd[7]	-,-,-,-,V,N,Z,C	1
ROR Rd	Rotate right via carry register	Rd[7] ←SREG[C] Rd[n-1] ←Rd[n] SREG[C] ←Rd[0]	-,-,-,-,V,N,Z,C	1
ASR Rd	Arithmetic shift right register (i.e., signed divide-by-2)	Rd[n-1] ←Rd[n] Rd[7] ←Rd[7]	-,-,-,-,V,N,Z,C	1
SWAP Rd	Swap nibbles register	Rd[3..0] ←Rd[7..4] Rd[7..4] ←Rd[3..0]	-,-,-,-,-,-,-,-	1
BSET b	Set status bit	SREG[b] ←1	SREG[b]	1
BCLR b	Clear status bit	SREG[b] ←0	SREG[b]	1
BST Rr,b	Bit store from register to bit copy storage	SREG[T] ←Rr[b]	-,T,-,-,-,-,-,-	1
BLD Rd,b	Bit load from bit copy storage to register	Rd[b] ←SREG[T]	-,-,-,-,-,-,-,-	1
SEC	Set carry	SREG[C] ←1	-,-,-,-,-,-,-,C	1
CLC	Clear carry	SREG[C] ←0	-,-,-,-,-,-,-,C	1
SEN	Set negative	SREG[N] ←1	-,-,-,-,-,N,-,-	1
CLN	Clear negative	SREG[N] ←0	-,-,-,-,-,N,-,-	1
SEZ	Set zero	SREG[Z] ←1	-,-,-,-,-,-,Z,-	1
CLZ	Clear zero	SREG[Z] ←0	-,-,-,-,-,-,Z,-	1

Table C.4: Bit and Bit-Test Instructions

Mnemonic	Description	Operation	Affected SREG Bits	Clk Cyc
SEI	Global interrupts enabled	SREG[I] ←1	I,-,-,-,-,-,-,-	1
CLI	Global interrupts disabled	SREG[I] ←0	I,-,-,-,-,-,-,-	1
SES	Set signed	SREG[S] ←1	-,-,-,S,-,-,-,-	1
CLS	Clear signed	SREG[S] ←0	-,-,-,S,-,-,-,-	1
SEV	Set overflow	SREG[V] ←1	-,-,-,-,V,-,-,-	1
CLV	Clear overflow	SREG[V] ←0	-,-,-,-,V,-,-,-	1
SET	Set bit copy storage	SREG[T] ←1	-,T,-,-,-,-,-,-	1
CLT	Clear bit copy storage	SREG[T] ←0	-,T,-,-,-,-,-,-	1
SEH	Set half-carry	SREG[H] ←1	-,-,H,-,-,-,-,-	1
CLH	Clear half-carry	SREG[H] ←0	-,-,H,-,-,-,-,-	1

Source: ATMEL, "8-bit AVR Microcontroller with 4/8/16/32K Bytes In-System Programmable Flash," Rev. 8161D - 10/09 ©2009 Atmel Corporation.

Table C.5: Data Transfer Instructions

Mnemonic	Description	Operation	Affected SREG Bits	Clk Cyc
MOV Rd,Rr	Move between registers	Rd ← Rr	-,-,-,-,-,-,-,-	1
MOVW Rd,Rr	Copy register word	Rd+1:Rd ←Rr+1:Rr	-,-,-,-,-,-,-,-	1
LDI Rd,K	Load constant	Rd ← K	-,-,-,-,-,-,-,-	1
LD Rd,X	Load indirect	Rd ← (X)	-,-,-,-,-,-,-,-	2
LD Rd,X+	Load indirect and post-increment	Rd ← (X) X ← X + 1	-,-,-,-,-,-,-,-	2
LD Rd,-X	Pre-decrement and load indirect	X ← X - 1 Rd ← (X)	-,-,-,-,-,-,-,-	2
LD Rd,Y	Load indirect	Rd ← (Y)	-,-,-,-,-,-,-,-	2
LD Rd,Y+	Load indirect and post-increment	Rd ← (Y) Y ← Y + 1	-,-,-,-,-,-,-,-	2
LD Rd,-Y	Pre-decrement and load indirect	Y ← Y - 1 Rd ← (Y)	-,-,-,-,-,-,-,-	2
LDD Rd,Y+q	Load indirect with displacement	Rd ← (Y + q)	-,-,-,-,-,-,-,-	2
LD Rd,Z	Load indirect	Rd ← (Z)	-,-,-,-,-,-,-,-	2
LD Rd,Z+	Load indirect and post-increment	Rd ← (Z) Z ← Z + 1	-,-,-,-,-,-,-,-	2
LD Rd,-Z	Pre-decrement and load indirect	Z ← Z - 1 Rd ← (Z)	-,-,-,-,-,-,-,-	2

Table C.5: Data Transfer Instructions

Mnemonic	Description	Operation	Affected SREG Bits	Clk Cyc
LDD Rd,Z+q	Load indirect with displacement	Rd ← (Z + q)	-,-,-,-,-,-,-,-	2
LDS Rd,k	Load direct from SRAM	Rd ← (k)	-,-,-,-,-,-,-,-	2
ST X,Rr	Store indirect	(X) ← Rr	-,-,-,-,-,-,-,-	2
ST X+,Rr	Store indirect and post-increment	(X) ← Rr X ← X + 1	-,-,-,-,-,-,-,-	2
ST -X,Rr	Pre-decrement and store indirect	X ← X - 1 (X) ← Rr	-,-,-,-,-,-,-,-	2
ST Y,Rr	Store indirect	(Y) ← Rr	-,-,-,-,-,-,-,-	2
ST Y+,Rr	Store indirect and post-increment	(Y) ← Rr Y ← Y + 1	-,-,-,-,-,-,-,-	2
ST -Y,Rr	Pre-decrement and store indirect	Y ← Y - 1 (Y) ← Rr	-,-,-,-,-,-,-,-	2
STD Y+q,Rr	Store indirect with displacement	(Y + q) ← Rr	-,-,-,-,-,-,-,-	2
ST Z,Rr	Store indirect	(Z) ← Rr	-,-,-,-,-,-,-,-	2
ST Z+,Rr	Store indirect and post-increment	(Z) ← Rr Z ← Z + 1	-,-,-,-,-,-,-,-	2
ST -Z,Rr	Pre-decrement and store indirect	Z ← Z - 1 (Z) ← Rr	-,-,-,-,-,-,-,-	2
STD Z+q,Rr	Store indirect with displacement	(Z + q) ← Rr	-,-,-,-,-,-,-,-	2
STS k,Rr	Store direct to SRAM	(k) ← Rr	-,-,-,-,-,-,-,-	2
LPM	Load program memory	R0 ← (Z)	-,-,-,-,-,-,-,-	3
LPM Rd,Z	Load program memory	Rd ← (Z)	-,-,-,-,-,-,-,-	3
LPM Rd,Z+	Load program memory and post-increment	Rd ← (Z) Z ← Z + 1	-,-,-,-,-,-,-,-	3
SPM	Store program memory	(Z) ← R1:R0	-,-,-,-,-,-,-,-	-
IN Rd,P	In port	Rd ← P	-,-,-,-,-,-,-,-	1
OUT P,Rr	Out port	P ← Rr	-,-,-,-,-,-,-,-	1
PUSH Rr	Push register on stack	(SP) ← Rr SP ← SP - 1	-,-,-,-,-,-,-,-	2
POP Rd	Pop register from stack	SP ← SP + 1 Rd ← (SP)	-,-,-,-,-,-,-,-	2

Source: ATMEL, "8-bit AVR Microcontroller with 4/8/16/32K Bytes In-System Programmable Flash," Rev. 8161D - 10/09 ©2009 Atmel Corporation.

Table C.6: MCU Control Instructions

Mnemonic	Description	Operation	Affected SREG Bits	Clk Cyc
NOP	No operation		-,-,-,-,-,-,-,-	1
SLEEP	Sleep (1 of 6 modes available)		-,-,-,-,-,-,-,-	1
WDR	Watchdog reset		-,-,-,-,-,-,-,-	1
BREAK	Break (for on-chip debug only)		-,-,-,-,-,-,-,-	-

Source: ATMEL, "8-bit AVR Microcontroller with 4/8/16/32K Bytes In-System Programmable Flash," Rev. 8161D - 10/09 ©2009 Atmel Corporation.

C.2 INSTRUCTION SET NOTATION

C.2.1 SREG - AVR STATUS REGISTER

Bit	7	6	5	4	3	2	1	0
0x5F	I	T	H	S	V	N	Z	C
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	0	0	0	0	0	0	0	0

- I: Global Interrupt Enable. The Global Interrupt Enable bit must be set for the interrupts to be enabled. The individual interrupt enable control is then performed in separate control registers. If the Global Interrupt Enable bit is cleared, none of the interrupts are enabled independent of the individual interrupt enable settings. The I-bit is cleared by hardware after an interrupt has occurred, and it is set by the `RETI` instruction to enable subsequent interrupts. The I-bit can also be set and cleared by the application with the `SEI` and `CLI` instructions.
- T: Bit Copy Storage. The Bit Copy instructions `BLD` and `BST` use the T-bit as source or destination for the operated bit. A bit from a CPU register can be copied into T by the `BST` instruction, and a bit in T can be copied into a bit in a CPU register by the `BLD` instruction.
- H: Half-Carry. The H-bit indicates a half-carry in some arithmetic operations. Half-carry is useful in Binary-Coded Decimal (BCD) arithmetic.
- S: Sign Bit. The S-bit is always an exclusive-OR between N and V.
- V: Two's Complement Overflow. The V-bit supports two's complement arithmetics.
- N: Negative. The N-bit indicates a negative result in an arithmetic or logic operation.
- Z: Zero. The Z-bit indicates a zero result in an arithmetic or logic operation.
- C: Carry. The C-bit indicates a carry in an arithmetic or logic operation.

C.2.2 GENERAL PURPOSE REGISTER FILE

All assembly instructions containing operands `Rd` (register destination) or `Rr` (register read) make use of the 32 general purpose working 8-bit registers in the CPU. Each CPU register is also assigned a data memory address, mapping them directly into the first 32 locations of the user data space. Thus, `R0` is mapped to `0x00` and `R31` is mapped to `0x1F`.

The registers `R26` through `R31` have some added functions to their general purpose usage. These registers are 16-bit address pointers for indirect addressing of the data space. The three indirect address registers `X`, `Y`, and `Z` are defined as `(XH:HL) = (R27:R26)`, `(YH:YL) = (R29:R28)`, and `(ZH:ZL) = (R31:R30)`. In the different addressing modes, these address registers have functions as fixed displacement, automatic increment, and automatic decrement.

C.2.3 MISCELLANEOUS

The remaining notations are used in the assembly instruction tables.

- `PC` is the 16-bit Program Count register.
- `SP` is the 16-bit Stack Pointer register.
- `Rdl` in the 16-bit addition and subtraction operations must be one of `R24`, `R26`, `R28` or `R30`.
- `K` is an immediate 8-bit constant value (i.e., `0x00` through `0xFF`).
- `k` is a 16-bit constant value used in various addressing instructions. Different instructions apply their own limitations on the value of `k`.
- `S` represents the current machine instruction size, and it is used while manipulating the `PC`.
- `N` represents the next machine instruction size, and is used while manipulating the `PC`.
- `b` is the position of a bit in a register (i.e., `0` through `7`).
- `P` is the data space address of the registers between `0x20` through `0x3F` for the `SBI` and `CBI` instructions and `0x20` through `0x5F` for the `IN` and `OUT` instructions. Note that these addresses are adjusted by `0x20` to exclude the CPU registers, which are mapped to `0x00` through `0x1F`. So any `P` address corresponds to an adjusted register address. For example, the instruction `IN R0,0x03` would load register `R0` with the contents of `PINB`, which is normally mapped to the address `0x23` but has been adjusted to ignore the CPU register locations.
- `(X)`, `(Y)` and `(Z)` is the indirect addressing mode where the 16-bit value in `X`, `Y` or `Z` is used as an address. The value stored at the address contained in the register is used in the load or store instructions.
- `q` is a 6-bit constant value used in the indirect with displacement addressing modes (i.e., `0x00` through `0x3F`).

C.2.4 STACK POINTER

The Stack is mainly used for storing temporary data, for storing local variables and for storing return addresses after interrupts and subroutine calls. Note that the Stack is implemented as growing from higher to lower memory locations. The Stack Pointer register always points to the top of the Stack. The Stack Pointer points to the data SRAM Stack area where the subroutine and interrupt Stacks are located. A Stack `PUSH` command will decrease the Stack Pointer.

The Stack in the data SRAM must be defined by the program before any subroutine calls are executed or interrupts are enabled. The initial Stack Pointer value equals the last address of the internal SRAM, and the Stack Pointer must be set to point above start of the SRAM.

The AVR Stack Pointer is implemented as two 8-bit registers in the I/O space. The number of bits used in the ATmega328 is 11 (i.e., 2 Kbytes of SRAM provides `0x000` through `0x7FF`).

Bit	7	6	5	4	3	2	1	0
0x5E	-	-	-	-	-	SP10	SP9	SP8
Read/Write	R	R	R	R	R	R/W	R/W	R/W
Default	0	0	0	0	0	1	1	1

Bit	7	6	5	4	3	2	1	0
0x5D	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Default	1	1	1	1	1	1	1	1

APPENDIX D

Example C/C++ Software Coding Guidelines

D.1 INTRODUCTION

It is well accepted in industry that creating high-quality software requires source code to be consistent, modular and self-documenting. Software is unique in that, often, large teams of engineers will work on the same project over a long period of time. For example, my first engineering job was with Motorola in which I joined a team of 30 software engineers that had been working on a project for over five years. Imagine a fresh-out engineer coming onto a project containing millions of lines-of-code. If it were not for strict coding guidelines, I would never have been able to overcome the learning curve in a timely manner. Ever since, I have been an advocate of good software engineering practice.

Motorola recognized that the quality of software was directly related to the maintainability of the source code. In fact, engineers working on my project were not allowed to check in their work until it underwent a strict peer-review process. One of the corner-stones of the review involved consistency checks against a set of software coding rules for the group. One of the most important aspects to developing high-quality software is consistency among source files on a large project. Consistent source-code listings allow a higher degree of maintainability, as engineers are able to analyze software as a whole without the difficulty of switching from style to style. Unfortunately, the skill of writing consistent software is grossly neglected within most academic environments as many don't recognize its importance.

The remainder of this appendix contains an actual example C/C++ Software Coding Guidelines document compiled by Jim Preston of RELM Wireless Corp. This document represents the very minimum set of rules that many companies would use as a standard. Note that some rules have been removed, as they represent very specific features of the project.

D.1.1 PURPOSE

This document contains recommendations for the development of C/C++ software within (a company). The purpose of these recommendations is to promote the readability and maintainability of source code among multiple developers. This is a living document and subject to revision.

The recommendations are based on established standards collected from a number of sources, including individual experience, local requirements, and suggestions from the sources cited at the end of this document.

D.1.2 PHILOSOPHY

D.1.2.1 Code

- Code should be kept as simple and straightforward as possible.
- In the context of having proper architecture and design documentation, code should be written in a self-documenting manner.
- Optimization should only be done when proven to be necessary.

D.1.2.2 Comments

- Comments are not meant to take the place of separate architecture and design documentation.
- Comments should be meaningful, and they should not repeat what is stated in the code itself.
- Anything that is not obvious should be commented.
- Comments should be updated as the code is updated.

D.1.2.3 Editors

While a specific editor can enhance the readability of code by color coding, automatic formatting, etc., the code should not rely on such features to be readable. Source code should be written in a way that maximizes its readability independent of any editor.

D.1.3 FORMAT

D.1.3.1 Layout of the Recommendations

The recommendations are grouped by topic, and each recommendation is numbered to make references easier during reviews.

Recommendations have the following format:

n. Recommendation short description.
Example (if applicable)
Motivation, background and additional information.

D.1.3.2 Recommendation Importance

The terms *must*, *should* and *can* have special meaning. A *must* requirement must be followed, *should* is a strong recommendation, and *can* is a general guideline.

D.2 GENERAL RECOMMENDATIONS

1. Any violation to the guide is allowed if it enhances readability.

It is impossible to cover all the specific cases in a general guide and the developer should be flexible.

2. The rules can be violated if there are strong personal objections against them.

The attempt is to make a guideline, not to force a particular coding style onto individuals.

D.3 NAMING CONVENTIONS

3. Names representing types should be in mixed case starting with upper case.

```
Line, SavingsAccount
```

Common practice in the C++ development community.

5. Variable names should be in mixed case starting with lower case.

```
line, savingsAccount
```

Common practice in the C++ development community. Makes it easy to distinguish variables from types, and effectively resolves potential naming collisions as in the following declaration:

```
Line line;
```

6. C++ function names should be in mixed case starting with upper case.

```
SetName(), CurrentIndex()
```

Common practice in the C++ development community.

7. For C function names, underscores should be used to separate module abbreviations from the rest of the name. The use of two consecutive underscores should be avoided.

```
MM_Initialize()
```

Helps to locate function definitions within files.

8. Underscores must not be used at the beginning or end of an identifier.

```
_TaskCounter   // NO
messageIndex_  // NO
```

This convention is often reserved for system purposes.

9. Global variables must be prefixed with g_.

```
g_globalCounter
```

Distinguishes global variables from other variables. In general, the use of global variables should be avoided.

10. Private class variables must be prefixed with m_.

```
class CSomeClass
{
    private:
        int m_length;
}
```

Indicating class scope by using an "m_" prefix makes it easy to distinguish class variables from local variables.

11. Named constants (including enumeration values) must be all uppercase using underscore to separate words.

```
MAX_ITERATIONS, COLOR_RED, PI
```

This is common practice in the C++ development community.

12. Enumeration constants can be prefixed by a common type name.

```
enum Color
{
    COLOR_RED,
    COLOR_GREEN,
    COLOR_BLUE
};
```

This gives additional information of where the declaration can be found, which constants belong together, and what concept the constants represent.
An alternative approach is to always refer to the constants through their common type: `Color::RED`, `Airline::SOUTHWEST`, etc.

D.4 LAYOUT

13. Basic indentation should be 4 spaces.

```
for (i = 0; i < numElements; i++)
{
    m_elements[i] = 0;
}
```

This is a common indentation amount. Editors should be set to use spaces in place of tabs.

14. Block layout should be as illustrated in example 1 below, and it must not be as shown in examples 3 or 4.

```
// Example 1
while (!done)
{
    DoSomething();
    done = MoreToDo();
}

// Example 2
while (!done) {
    DoSomething();
    done = MoreToDo();
}

// Example 3
while (!done)
    {
    DoSomething();
    done = MoreToDo();
    }

// Example 4
while (!done)
    {
        DoSomething();
        done = MoreToDo();
    }
```

The styles in examples 3 and 4 don't emphasize the logical structure of the code as clearly as examples 1 and 2. Example 4 introduces an extra indentation level.

The choice between examples 1 and 2 is largely a matter of personal preference. However, standardizing on a particular style enhances readability. One argument in favor of example 1 is that brackets are generally easier to find when they are put on their own line.

15. Class declarations should have the following form:

```
class CSomeClass : public CBaseClass
{
    public:
        ...

    protected:
        ...

    private:
        ...
}
```

This partially follows from the general block rule above. The ordering is "most public first," so people who only wish to use the class can stop reading when they reach the protected/private sections. Sections that are not applicable should be left out.

16. A `switch` statement should have the following form:

```
switch (condition)
{
    case ABC:
        statements;
        break;

    case DEF:
    case XYZ:
        statements;
        break;

    default:
        statements;
        break;
}
```

Individual **`case`** blocks should be separated by a blank line after the **`break`** statement. **`switch`** statements should always include a **`default`** case to catch errors.

17. Single-statement `if-else`, `for` and `while` statements should have brackets.

```
if (condition)
{
    statement;
}

// NOT
if (condition)
    statement;
```

The primary motivation for this recommendation is so that single-statement and multi-statement forms of these constructs have a consistent layout. This enhances readability.

18. Whitespace:

- **Conventional operators should be surrounded by a space.**
- **C++ reserved words should be followed by a space.**
- **Commas should be followed by a space.**
- **Semicolons in `for` statements should be followed by a space.**

```
a = (b + c) * d; // NOT: a=(b+c)*d

while (true) // NOT: while(true)
{
    ...
}

DoSomething(a, b, c, d); // NOT: DoSomething(a,b,c,d);

for (i = 0; i < 10; i++) // NOT: for(i=0;i<10;i++)
{
    ...
}

return value;
```

Makes the individual components of the statements stand out. Enhances readability.

19. Expressions should be fully parenthesized.

```
total = CURRENCY * (dollars + change);
```

Enhances readability and prevents some calculation mistakes.

20. For a function with a large number of arguments, the function should be declared with one variable per line.

```
void GetTotal (const void* itemOne,
    const void* itemTwo,
    const void* itemThree,
    const void* itemFour,
    const void* itemFive,
    const void* itemSix);
```

Enhances readability. Also allows commenting each of the parameters individually if necessary.

D.5 STATEMENTS

D.5.1 TYPES

21. Types that are local to one file only can be declared inside that file.

Enforces information hiding.

D.5.2 VARIABLES

22. Variables should be declared one per line.

Enhances readability for both variable declaration and initialization.

23. Variables must not have dual meaning.

Enhances readability. Reduces chance of error by variable side effects.

24. Variables should be declared in the smallest scope possible.

Keeping the operations on a variable within a small scope, makes it is easier to control the effects and side effects of the variable.

25. Class variables should not be declared public.

The concept of C++ information hiding and encapsulation is violated by public variables. Use private variables and access functions instead.
One exception to this rule is when the class is essentially a data structure, with no behavior (equivalent to a C **struct**). In this case, it is appropriate to make the class' instance variables public.

26. C++ pointers and references should have their reference symbol next to the type rather than next to the name.

```
float* x; // NOT: float *x;
int& y;   // NOT: int &y;
```

The "pointer-ness" or "reference-ness" of a variable is a property of the type rather than the name.

D.5.3 LOOPS

27. Only loop control statements should be included in `for()` constructs.

```
sum = 0;                        // NOT:
for (i = 0; i < 100; i++)       // for (i = 0, sum = 0; i < 100; i++)
{                               // {
    sum += value[i];            //     sum += value[i];
}                               // }
```

This increases readability and maintainability. It makes a clear distinction between loop controls and loop content.

28. The use of `break` and `continue` should be avoided, except for the standard use of `break` in `switch` statements.

These statements should only be used if they produce higher readability than their structured counterparts.

D.5.4 CONDITIONALS

29. Executable statements in conditionals should be avoided.

```
File* fileHandle = open(fileName, "w");
if (!fileHandle)
{
    ...
}

// NOT:
if (!(fileHandle = open(fileName, "w")))
{
    ...
}
```

Conditionals with executable statements are difficult to read.

D.5.5 FUNCTIONS

30. Function arguments that are not altered within the function, and are passed by reference, should be passed with the `const` type qualifier.

```
void* GetTotal(const void* itemOne, const void* itemTwo)
{
    ...
}
```

This protects the variables from alteration and indicates that intent to the reader.

D.5.6 MISCELLANEOUS

31. Split lines must be made obvious.

```
totalSum = a + b + c +
           d + e;

if (conditionOne &&
    conditionTwo)

someSetting = (settings & SOME_MASK) >>
              SOME_SHIFT;
```

It is difficult to give rigid rules for how lines should be split, but in general:

- Break after a comma.
- Break after an operator.
- Align the new line with the beginning of the expression on the previous line.

32. If a complex expression is split over several lines, the logic of the expression should be made clear through the placement of the line breaks.

```
if ((conditionOne && conditionTwo) ||
    (conditionThree && conditionFour))

// NOT
if ((conditionOne && conditionTwo) || (conditionThree &&
    conditionFour))
```

Enhances readability.

33. The use of hard-coded magic numbers should be avoided. Numbers other than 0 and 1 should be declared as named constants instead.

If the number does not have an obvious meaning by itself, readability is enhanced by introducing a named constant instead.

34. If a named constant definition includes logic or arithmetic operations, the entire definition should be surrounded with parentheses.

```
#define PI      3.141590210
#define FOUR_PI (4 * PI)
```

Named constants are replaced by their defined value wherever they appear in code. Without parentheses, this can introduce problems with the order of operations.

35. `goto` should not be used.

`goto` statements violate the idea of structured code. Only in a few cases (for instance, breaking out of deeply nested structures) should **`goto`** be considered, and only if the alternative structured counterpart is less readable.

36. The use of macros should be kept to a minimum.

While they are useful, macros make debugging more difficult. Code within a macro should be simple and well understood, and should be kept to a few lines.

37. In general, there should be only one exit point from a method.

Multiple exit points (i.e., **`return`** statements in the middle of a method) should not be used unless they improve readability.

D.6 COMMENTS

39. `#endif` preprocessor statements should always be followed by a comment.

```
#ifdef USE_BLOCK

. . .

#endif // USE_BLOCK
```

The reader will either be able to see the beginning statement on the screen or a comment. The comment can be either an explanation or simply a copy of the code that began the block.

40. Comments should in general be above the statements being commented, not to the side.

```
// Some comment
statement

//NOT
statement     // Some comment
```

It may be more readable, in some cases, to put the comments for variable declarations and parameter lists to the right of the statement.

D.7 FILES

41. Classes and modules should be declared in a header file and defined in a source file where the names of the files match the name of the class/module.

```
MyClass.h, MyClass.cpp
```

This makes it easy to find the associated files of a given class.

42. File content should be kept within 110 columns.

Excessively long lines hinder readability.

43. Header files must contain an include guard.

```
#ifndef _FILENAME_H_
#define _FILENAME_H_
:
 [header file body].
:
#endif //_FILENAME_H_
```

Include guards protect against multiple inclusions.

44. Include statements should be sorted and grouped. They should be sorted by their hierarchical position in the system with low level files included first.

```
#include <fstream>
#include <iomanip>
#include "PropertiesDialog.h"
#include "MainWindow.h"
```

It gives an immediate clue about which modules are used.
Include file paths must never be absolute. Compiler directives should be used instead to indicate root directories for includes.

Bibliography

R. H. Katz, *Contemporary Logic Design*, second edition, Prentice Hall. 6

S. Brown and Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design*, third edition, McGraw-Hill. 6

B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, second edition, Prentice Hall. 23

Arduino, 2010, `http://www.arduino.cc/`. 81

ATMEL, *8-bit AVR Microcontroller with 4/8/16/32K Bytes In-System Programmable Flash*, Rev. 8161D - 10/2009, Atmel Corporation. 85, 95, 96, 97, 100, 101, 103, 105, 107, 119, 140, 154, 179, 212, 213, 214

ATMEL, *8-bit AVR Microcontrollers JTAGICE mkII Quick Start Guide*, Rev. 2562C - 07/2006, Atmel Corporation. 89

ATMEL, *AVR ONE! Quick-start Guide*, Rev. 32104B - 02/2010, Atmel Corporation. 89

Author's Biography

DAVID J. RUSSELL

David J. Russell received the B.S. degree in computer engineering and the M.S. degree in electrical engineering from the University of Nebraska, Lincoln, Nebraska, in 1996 and 2001, respectively. Currently, he is a Ph.D. candidate in the Department of Electrical Engineering of the University of Nebraska-Lincoln. His research interests include digital signal processing, wireless communications, and biological sequence analysis. From 1996 to 1997, he was with Motorola Space and Systems Technology Group, Scottsdale, Arizona, where he was a software engineer. From 1997 to 2004, he was with EFJohnson, Lincoln, Nebraska, where he was a principal development engineer. Since 2004, he has been with the University of Nebraska, Lincoln, where he currently serves as Lecturer. Additionally, he has worked as a contract engineer for Telex, a part of Bosch Security Systems. Most recently, he helped create and presently works for the contract engineering company Red-Cocoa.

Index