COMPUTING IN THE PHYSICAL WORLD

Roger D. Chamberlain and Ron K. Cytron

Draft Version 0.07

Contents

Preface

Microcontrollers are fascinating devices. Complete computers on a chip, and inexpensive enough to be affordable by hobbyists, they have been used for almost every purpose imaginable. They maintain proper water chemistry in swimming pools, they automatically feed the chickens on the farm, they count the steps we take as part of helping us monitor our health, they control the anti-lock brakes on our vehicles (along with the ignition timing and a host of other things), they manufacture an innumerable variety of widgets on production lines, they do all kinds of things.

The phrase *embedded computer* is often used to describe a computer that is "embedded" (i.e., part of) a larger system (like a vehicle). For example, when I buy a car, I don't separately buy a computer to run the anti-lock brakes, that computer "comes with" the car and is part of the car. From the consumers' point of view, they haven't purchased a computer, they've purchased a car that happens to have an embedded computer that is part of it. In practice, there are actually many computers in modern cars.

As a developer of applications that run on embedded computers, one must be aware of how these machines are similar to and are different than traditional desktop, laptop, or server computers. Embedded computers often have very limited memory (kilobytes or megabytes rather than gigabytes). While modern desktop systems are almost universally 64-bit machines, embedded processors are frequently 32-bits wide or even less (8-bit machines are actually quite common).

The examples throughout the book exploit the Arduino Uno platform. The Arduino Uno is one of a family of small microcontroller systems based upon the AVR series of chips from Microchip. The AVR is an 8-bit microcontroller with a small on-chip memory, which helps keep it quite inexpensive. Rather than spending $500 or more for a computer, an Arduino Uno costs closer to $20. The Arduino platform (in all its variations) has quite a large following in the hobbyist community.

There are many books about microcontrollers, how to interface microcon-

trollers to physical devices (both input devices and output devices), and how to write code that interacts with these devices. Many of these books target the Arduino platform (both the Arduino Uno board and its associated integrated development environment, or IDE). Most of these books, however, are written for the hobbyist. The focus is on some particular set of experiments or projects to build, rather than the general principles that underlay those experiments or projects.

Here, we are primarily interested in the underlying fundamental principles that guide how computers interact with the physical world. Rather than focus on how to make a temperature measurement, for example, our focus is how to go about sensing continuous-valued properties of the physical world, with the temperature measurement being but one instance.

There are a number of principles that are distinct from traditional, desktop computing.

- **Sensing the physical environment – interacting with transducers.** To interact with the physical world, there must be a literal connection that enables the interaction. A transducer that is impacted by the physical world generates a signal that can be measured by the processor. From the processor's perspective, this can take one of two forms: digital or analog. We will describe digital input mechanisms as well as analog-to-digital conversion. What information transformation takes place as part of an analog-to-digital conversion? What determines the ranges of signals that can be effectively measured?

- **Transforming the transducer signal into knowledge.** It is rare that signals coming directly from transducers are what is practically desired. Voltage signals need to be converted into engineering units that are relevant to the application. Temperature signals need to be converted into degrees Celsius or degrees Fahrenheit. Accelerometer signals need to be processed to detect steps as an input to a pedometer. We will describe the formal mechanisms for many of these conversions and will introduce novice-friendly versions of transformations that are substantially more sophisticated.

- **Impacting the physical environment – interacting with actuators.** In addition to sensing the physical world, we frequently want to have an impact on it. Whether turning heating elements on and off to control the temperature in a space, or engaging drive wheels to make a robot move, there is quite a bit to address when dealing with output devices. As with input devices, outputs can be classified into the

two forms of digital outputs (on/off) or analog outputs (continuous signals). In addition, digital-to-analog conversion takes a number of forms. We will describe not only the mechanisms associated with driving output devices, but also discuss approaches to digital feedback control, in which the computer system is tasked with controlling some aspect of the physical world.

- **Dealing effectively with time.** The correctness of many computer programs is not impacted by how long the computation takes to complete (as long as it completes in finite time). The opposite is true for many circumstances in which a computer is interacting with the physical world. When something happens is often just as crucial to correct operation as what happens. We will describe approaches to software development that include time as a functional property.

In addition, there are a number of principles that are common to embedded computing and desktop computing.

- **Importance of good design.** Design patterns that have a long history of repeated use are a critical thing for new programmers to start learning. We will describe the finite-state machine computational abstraction and provide several examples of its use. In addition, we will describe and contrast polling versus interrupt-driven approaches to digital input.

- **Information representation.** Everything input, processed, stored, or output from a digital computer is represented in some digital form. This includes numbers, letters, images, videos, graphs, maps, and everything else. And to make the world even more complicated, not all computers represent information in exactly the same way. We will describe the commonly used forms of information representation in microcontrollers, starting with a brief history of numbers, then describing integers and floating-point numbers and progressing to text and a very brief introduction to images.

- **User interfaces.** While embedded systems are frequently interacting with the physical world, it is also imperative that they interact with human users. We will describe approaches to the human-computer interface that are common in embedded systems, which mostly use the same technology as the approaches used in desktop machines.

- **Computer architecture.** While one can program a computer with the mental model that the machine directly executes instructions in the

programming language of choice, that is at best a poor illusion of what is really happening. We will describe the basics of how a computer is constructed, how it executes instructions, and what actual instructions really exist.

- **Assembly language.** When exploring how computers actually work, it is common practice to program them directly with the instructions that they actually execute. Assembly language is a human-accessible version of those instructions. We will describe both machine language (the actual instructions), assembly language, and their relationship to one another. This is followed by a discussion of assembly language programs can effectively interact with programs written in a higher-level language such as C.

- **Computer to computer communications.** Computers frequently communicate with one another, using a variety of physical mechanisms, including wired networks, wireless networks, and point-to-point cables of many forms. Common to all of these mechanisms, however, is the concept of a stream of bytes being delivered from one machine to the other. We will describe both the concept of streams and several mechanisms for their delivery. This is followed by a discussion of communication protocols and the sending and receiving of messages of various forms.

We will address each of these principles as part of this book.

Very little prerequisite knowledge is needed to be able to understand and learn the material covered in the book. We assume a familiarity with a procedural language (e.g., Java is commonly taught as a first language and is syntactically very close to the subset of C used on the Arduino platform). We don't assume knowledge of object-oriented programming (i.e., no polymorphism, no inheritance, etc.). We assume a very basic understanding of electricity (voltage, current, how to read a schematic diagram), but you don't need to know Ohm's Law, Kirchoff's Laws, or how to design an amplifier. The mathematical tools required are limited to algebra and base conversions (between decimal, binary, and hexadecimal).

There are a set of appendices that will help the reader that is unfamiliar with some of the required prerequisite knowledge to come up to speed. Appendix A gives a quick overview comparing the Java and C programming languages, including a comparison between standard C and the variant we call Arduino C. Appendix B provides a simple introduction to electricity at the level needed to understand the examples in the book. Appendix C provides a refresher course on base conversions.

This book was written to serve as the textbook for a first-year, second-semester course in the Dept. of Computer Science and Engineering at Washington University in St. Louis [3]. It is a required course for both computer science and computer engineering majors, and it serves as a technical elective for computer science minors and electrical engineering majors. In contrast to traditional embedded computing books and courses, which are typically offered at the junior or senior level, this book and the associated course are aimed squarely at students who are early in their study of computing.

The course at Washington University in St. Louis covers the majority of the book. There are, however, elements that are separable so as to enable a course that does not move at the same pace. Chapters 1 through 8 are considered core material and have a place in any course, although some individual topics (e.g., polling versus interrupt-driven input) can easily be skipped. Optional material includes Chapter 9 – User Interaction, Chapters 10 and 11 – Computer Architecture and Assembly Language, and Chapter 12 – Computer to Computer Communications.

Acknowledgements

1 Introduction

1.1 Beginnings

Years ago, computers were big bulky things that consumed entire rooms, sometimes entire buildings, to house them. Users accessed them by first scheduling time with the owner of the computer (only one program at a time could run) and then coming to the computer room to manually input their program and input data, perform a run, and examine the outputs. The program might have been stored on a paper tape, with holes punched in it to represent the details of the machine instructions. The output invariably was on a stream of fan-fold paper.

We have we come a long way since then. Computers are now ubiquitous in our lives. We carry them around in our pockets, use them to interact with friends (both close friends we see every day and far-off friends we almost never see other than via social media), and trust them to monitor our health (keeping track of our heart rate and how many steps we make each day).

This text will focus primarily on the latter of these three examples of ubiquity. In the modern world, computers are no longer relegated to running programs that have input provided by a paper tape and outputs printed on fan-fold paper. Nor are they relegated to the more recent circumstance of input provided by a keyboard and outputs presented on a desktop screen. No, modern computers frequently take their inputs directly from measurements made from the physical world, and often control aspects of that same physical world.

1.1.1 Why?

We are interested in computers that interact with the real physical world because those computers can do so much more than a computer that is relegated to only have input from humans and output to humans. When a computer that is held in the palm of our hand includes a microphone, a speaker, and a

1

cellular radio, it becomes a phone. When a computer controls the timing of spark plug firings in an internal combustion engine, the engine can run more efficiently, increasing engine power and decreasing fuel consumption.

The examples above are possible when the computer senses one or more properties of the physical world around it and is able to effect change in the physical world as well. In this book, we describe how computers can interact with the real world, and what are the fundamental principles involved in building and programming computer systems that have these capabilities.

1.1.2 The Arduino Platform

The microcontroller that we will use to illustrate the topics we cover is the AVR microcontroller manufactured by Microchip, specifically the ATmega328P. It is an 8-bit processor, and it has 14 digital input/output pins (of which 6 can be used as pulse-width modulated analog outputs), 6 analog input pins (supporting a 10-bit A/D converter), 32 KBytes of program memory, and 2 KBytes of data memory. The term *microcontroller* is frequently used for a chip that contains not only the processor, but additional components as well, such as I/O and built-in memory.

The ATmega328P microcontroller is used on the Arduino Uno, one of a line of experimental boards used extensively by hobbyists. Other boards in the Arduino family use other microcontrollers in the AVR line (all of which share the same instruction set, varying in the number of I/O pins, memory, etc.).

All of the Arduino boards can be programmed using a variant of the C language. Software development is supported via an *integrated development environment* (IDE) that is open source and free to use. Arduino programs are called *sketches* in the hobbyist community, and we will follow that convention. Figure 1.1 shows a very simple sketch that prints a message to the desktop PC.

Every Arduino sketch has at least two components, `setup()` and `loop()`. The code that is in `setup()` is executed once, at the beginning of the run, and the code that is in `loop()` is executed repeatedly thereafter. A sketch does not terminate, but runs until stopped by the user (e.g., by issuing a reset).

Appendix A describes some of the idiosyncrasies of the Arduino C variant that is supported. It is also possible to author programs using the AVR assembly language, a topic that will be discussed in Chapter 10. All of the program code that we use in examples has been tested on the Arduino Uno platform. However, the changes needed for other Arduino boards are quite small (e.g., altering the specific pins used for particular I/O functions).

```
void setup() {
  Serial.begin(9600);              // setup communications
  Serial.println("Hello world!"); // print message
}

void loop() {
}
```

Figure 1.1: Hello world example sketch.

1.2 Digital Systems

In all digital systems, information is represented in binary form. The binary number system is one in which there are only two possible values for each digit: 0 and 1. At different times and for different purposes the 1s and 0s mean different things. One useful meaning is for 1 to represent TRUE and 0 to represent FALSE, allowing us to reason using propositional calculus.

Let's say we are studying at a university that requires all of its students to have taken one or more courses in economics prior to graduation. We will further assume that the economics requirement is to study both microeconomics (how individuals and organizations make economic decisions that effect themselves) and macroeconomics (how economies as a whole operate at a large scale, e.g., at the level of a country). Given the availability of the following three courses:

Econ A   Introduction to Microeconomics
Econ B   Introduction to Macroeconomics
Econ C   Economics Survey: Micro and Macro

we use the symbol A to represent a student having completed Econ A, the symbol B to represent the student having completed Econ B, and C to represent the completion of Econ C. Each of these symbols (A, B, or C) can take on the value 0 or 1, and cannot take on any other value. Under these constraints, these symbols are said to be *Boolean valued* (the name coming from George Boole, 19th century mathematician, who is often considered to be the father of modern digital logic [1]).

If the symbol E represents our student having completed the economics requirement, we can write down an equation that embodies this definition:

$$E = (A \text{ AND } B) \text{ OR } C \tag{1.1}$$

3

where the AND operator and the OR operator are described with precision below, but have meaning that is consistent with the normal English definitions of the terms. In English, we would say that the student needs to take both Econ A and Econ B (one providing microeconomics knowledge and the other providing macroeconomics knowledge) or the student needs to take Econ C (which provides both micro- and macroeconomics training). Clearly, the equation can be interpreted by someone reading it to mean exactly the same thing.

The AND operator and the OR operator are two of three basic logical operations supported in *Boolean algebra*, the third being the NOT operator. Boolean algebra is a mathematical framework that allows us to formally reason about Boolean valued variables, operations on those variables, and equations that utilize those operations. Equation (1.1) is an example of an equation in Boolean algebra.

We will define these three operations (AND, OR, NOT) through complete enumeration of all the possible combinations of values. This is a technique that is available to us primarily because the number of combinations isn't all that large. Since each variable can only have two values, things stay at reasonable sizes as long as the number of variables also stays small. It is common to call the tables that show all possible values *truth tables*. (It should be clear why this name is used, given the frequent interpretation, which we are using here, of 0 representing FALSE and 1 representing TRUE.) Table 1.1 shows the truth tables for the AND operation, the OR operation, and the NOT operation.

Table 1.1: Truth tables for (a) AND, (b) OR, and (c) NOT operations.

x	y	z		x	y	z		x	z
0	0	0		0	0	0		0	1
0	1	0		0	1	1		1	0
1	0	0		1	0	1			
1	1	1		1	1	1			
(a)				(b)				(c)	

As mentioned above, the formal definitions of the operations, as shown in Table 1.1, closely follow the normal English language usage of the words used to name the operations. The AND operation yields a 1 only when both inputs are 1, the OR operation yields a 1 when either input is a 1, and the NOT operation yields a 1 when its single input isn't a 1. It is important to note, however, that these formal definitions are how one resolves potential ambiguity. In English, the word "or" can, in some circumstances, mean "x or y but not both x and y together," but this is *not* the meaning defined in the

truth table.

Using the symbols frequently utilized by logicians, Equation (1.1) can be rewritten as follows:

$$E = (A \wedge B) \vee C \qquad (1.2)$$

where the $\wedge$ symbol is used to represent the AND operation and the $\vee$ symbol is used to represent the OR operation.

Just to illustrate that there are many ways to write down the same notion, the more common notation used in computer engineering and electrical engineering disciplines is to use the traditional addition symbol $(+)$ for OR and the traditional multiplication notation (either $\cdot$ or simply juxtaposition) for AND. Using this approach, the equation now looks like this,

$$E = (A \cdot B) + C \qquad (1.3)$$

or this,

$$E = AB + C \qquad (1.4)$$

where Equation (1.4) has also taken advantage of the normal convention that multiplication takes precedence over addition (in this case, AND takes precedence over OR) to drop the parenthesis from the equation.

Since there are only 3 variables on the right-hand side of the equation, and each variable can have only two values, we can examine this equation with the help of a truth table. Recall that in a truth table, all possible combinations of the input variables are listed, one combination per row. In the truth table for an expression, different columns are frequently used to represent different subexpressions (or the final value). The truth table for Equation (1.4) is shown in Table 1.2.

Table 1.2: Truth table showing all possible conditions for each input variable in Equation (1.4).

A	B	C	AB	$AB + C$	E
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	0	0
0	1	1	0	1	1
1	0	0	0	0	0
1	0	1	0	1	1
1	1	0	1	1	1
1	1	1	1	1	1

It is also possible to build a physical system that implements the logical reasoning encoded into Equation (1.4), and Equations (1.1) to (1.3) as well. Figure 1.2 shows the schematic diagram symbols for each of the 3 operations. The physical implementations of these symbols are called *gates*, and the logical values are encoded as voltages on input and output wires (typically, a HIGH voltage represents a 1 and a LOW voltage represents a 0).
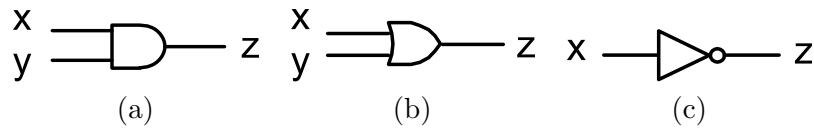
Figure 1.2: (a) AND gate. (b) OR gate. (c) NOT gate.

Using these logic gates as building blocks, Figure 1.3 shows a schematic diagram for a circuit that implements Equation (1.4). The inputs A, B, and C on the left encode whether or not the student has taken the courses Econ A, Econ B, and Econ C, respectively. The output E on the right encodes whether or not the student has met the economics requirement for the degree.
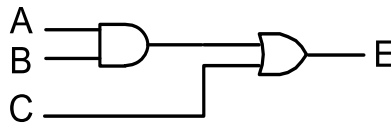
Figure 1.3: Gate-level schematic diagram of circuit that implements economics course requirement check.

Figure 1.3 is a simple example of an application-specific computation. It does a great job of its assigned application (checking whether or not a student has met his/her economics requirement); however, it doesn't really do much of anything else. Most of the time, we are interested in more general purpose computing devices. These are ones that can compute not only the results of Equation (1.4), but many other computations as well.

Computers are simply digital systems that have been engineered to do multiple tasks instead of an individual task. We provide them with a program, which gives specific instructions for the computer to execute. Computers come in many sizes, from small enough to fit in a smart watch, to large enough to fill a room; however, they all do exactly what they are told. They execute specific instructions given to them in the form of a program.

1.3 Authoring Programs

In order to provide a computer with a program, we must first design that program, and author it in some language that the computer can understand. Many are familiar with high-level languages, such as C, C++, Java, etc., that are frequently used to author programs. We will write in a variant of the C language as we develop programs for the AVR microcontroller.

These languages, however, are not the language of the processor itself. Instead, the processor directly executes *machine language*, a much lower-level language that directly encodes the specific instructions to be executed, one after the other, by the computer. Machine language instructions are represented as a string of 1s and 0s stored in the memory of the processor.

It is possible to author programs at the same conceptual level as machine language. To do this, we use *assembly language*, which is a human-readable and -writable language that has a one-to-one relationship with machine language. Instead of representing instructions directly as 1s and 0s, however, assembly language uses mnemonic names for instructions.

Figure 1.4 shows the relationship between machine language, assembly language, and high-level languages. Each assembly language instruction corresponds to an individual machine language instruction (in a one-to-one relationship). Each high-level language statement corresponds to multiple machine language instructions (in a one-to-many relationship).
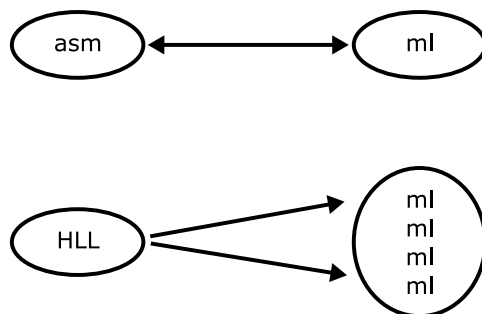
Figure 1.4: Relationship between languages. Assembly language (`asm`), machine language (`ml`), and high-level language (`HLL`).

Various software tools are used to translate between languages. Traditionally, a software tool that translates assembly language programs into machine language is called an *assembler*. The tool that translates a high-level language program into either assembly language or machine language is called a

compiler. Both are built into an *integrated development environment* (IDE).

1.4 Integrated Development Environments

A practitioner in any discipline would tell you that having the right tool for a given task improves the efficiency and enjoyment of completing the task, and that the quality of the completed task is improved as well. The development of software, hardware, or co-designed systems is often a complex undertaking, involving multiple developers, shifting goals and deliverables, and limited resources. Integrated development environments have emerged as the platform of choice for developers because they facilitate the creation of quality systems.

The process of authoring a program, as described above, involves different components of what is typically called the development *stack*. Each layer of the stack plays some role in the creation of the end product. An Integrated Development Environment (IDE) allows the layers of that stack to interoperate beneficially and to provide useful feedback to the product's developers. Here are some examples of the features provided by an IDE:

- The source code of the project can be viewed and edited efficiently. The syntax is typically formatted consistently and highlighted to improve readability. Color and font may distinguish programming language elements (`if`, `then`, `else`) from program-specific concepts (local variables, data types, packages).

  As an example, consider changing a particular method named `foo` to `bar` throughout an entire application. A text editor alone could find occurrences of `foo`, but some of those might be variable names and some may occur in comments. It requires the integration of a compiler with the editor to accomplish a method rename.

- A debugger may help diagnose the cause of a program's undesirable behavior. The IDE can facilitate tracing the program's behavior and relating it to variables, methods, and statements as seen in the editor.

- When a system is tested, one criterion for completeness concerns the *coverage* of a program's statements. Testing in an IDE can show lines of a project that have not been exercised by existing tests.

IDEs typically simplify importing code from a shared repository and exporting the results of modifications back to the repository. IDEs such as `eclipse` offer services and interfaces that allow the open-source community to introduce

new tools into the IDE that integrate well with tools and components already present.

1.5   Interacting with the Physical World

We are interested in computers that interact with the real world. They take measurements of physical phenomena, perform some computation, and optionally trigger some physical action. Examples of computers that perform these types of functions include the temperature and humidity controller in an environmental chamber, the embedded computers that control the flight surfaces (e.g., rudder, elevators, ailerons) in a fighter jet, and the Fitbit^{TM} that you might be wearing on your wrist right now. A self-driving car has embedded computers of this type, of course, or it wouldn't be able to know what was in its immediate environment and react accordingly. Regular cars, however, also have embedded computers doing sensing and control. Examples in include the computer-controlled anti-lock brakes and protective airbags, both safety-oriented subsystems in virtually all modern vehicles.

To accomplish these tasks, the computer cannot be limited to a keyboard for input and a screen for output. Instead, it must interact directly with the physical world. In the next 4 chapters, we will examine 4 specific mechanisms that allow interaction between the computer and the real world.

Another element of interaction that is common in most computers is their ability to communicate with other computers. The Internet of Things (IoT) has developed out of an ability for large numbers of computers to make measurements about the physical world and then communicate those measurements to other computers (e.g., a server in the cloud) so that knowledge can be gained about the entire collection.

1.6   The Role of Design

We are surrounded by objects, mechanisms, and interfaces that are the result of *design*, but ironically design is most successful when it is least apparent. When you approach a door with the intention of opening it, how do you know whether to push or pull on the door's handle? A well designed door makes such interactions obvious, to the extent that its users express no wonder at its ease of operation.

Practitioners of computer science and engineering are often faced with design choices and decisions that affect the efficiency, security, and usability of their products. The most effective designs take into account the context in

which a product will be deployed as well as considerations of how the product's usage might evolve over time. Even our simple door is not so simple: if the door has a latching mechanism, should it be operated via a simple knob? A round, smooth knob is less likely to tangle with clothing, but requires some dexterity and hand strength to operate. A lever can be operated more universally, but it could also interact unpredictably with belt loops or pants pockets.

The point here is that with most design, there is no absolutely right or wrong answer. There are tradeoffs that must be considered, and the value of a given design is measured in terms of its suitability for a particular environment or context. Throughout this text, you will be exposed to design choices, evaluations, and patterns. While you could study this material without any consideration of design, engineers continually grapple with design. By introducing design this early in your studies, you can begin to see the challenges and rewards of considering design alternatives as you develop solutions.

2 Digital Output

A digital output is pretty much exactly like you would expect, given the normal English definitions of "digital" and "output." There are only two possible values, which we will denote as 0 and 1, and the computer is sending one of those two values out into the physical world.

Electrically, one of the pins of the microcontroller is establishing a LOW voltage (for an output value of 0) or a HIGH voltage (for an output value of 1). To be safe (and for a number of other reasons), actually neither output value is a high enough voltage that we need to be concerned about touching it. The HIGH voltage mentioned above is approximately 5 V above the GND potential (for a 5 V microcontroller, it would be about 3.3 V on a 3.3 V microcontroller), and the LOW voltage is approximately 0 V above the GND potential (i.e., it is at the same voltage as GND). If you are unfamiliar with the concept of voltage, see Appendix B.

So, if we have a pin on the microcontroller that can establish a HIGH voltage or a LOW voltage out, what good is that? Let's examine what a digital output is actually good for.

2.1 Why Digital Outputs?

This book is about computing in the physical world, and a digital output is the simplest way that a computer can influence the world around it. If the computer is controlling the light in a room, a digital output is used to turn the light on or off. If the computer is communicating the presence (or absence) of an alarm, a digital output is used to turn the alarm on or off. This is true if the alarm is a buzzer or if the alarm is some large visual indicator. If the computer is controlling the heating element in an oven, a digital output can be used to turn the heating element on or off. If the computer is controlling a conveyor belt, a digital output is used to turn the conveyor belt on or off.

The pattern should be pretty obvious. Whenever there are two output

options, the physical effect can be controlled via a digital output. All that is required is circuitry that transforms the output voltage from the microcontroller (either LOW or HIGH) into the actuator control desired in the real world.

2.2 Software

To control a digital output pin from software, we must first configure the pin as a digital output. Most of the pins on the microcontroller serve multiple purposes (e.g., digital output and digital input), and it is our responsibility to configure the pin prior to use.

Configuring a digital output pin is accomplished using the `pinMode()` function. It takes two arguments, the first is an `int` identifying the pin number and the second is the the constant `OUTPUT` indicating that the pin is now a digital output.

Once the pin has been configured, the `digitalWrite()` function is used to set the output `HIGH` or `LOW`. A `HIGH` output corresponds to 5 V (for a 5 V microcontroller) and a `LOW` output corresponds to 0 V.

Figure 2.1 gives an example sketch that toggles a digital output at 1 Hz (high for 0.5 s, low for 0.5 s, high for 0.5 s, etc.). Figure 2.2 shown an image from a logic analyzer what happens on that output pin

```
const int doPin = 12;          // digital output pin is 17

void setup() {
  pinMode(doPin, OUTPUT);     // set pin to digital output
}

void loop() {
  digitalWrite(doPin, HIGH);  // set the output HIGH
  delay(500);                 // wait for 0.5 s (500 ms)
  digitalWrite(doPin, LOW);   // set the output LOW
  delay(500);                 // wait for 0.5 s
}
```

Figure 2.1: Example digital output sketch.

Figure 2.2: Logic analyzer trace illustrating output square wave.

Practice Problem   What is the period of the output waveform generated by the sketch of Figure 2.1?

Solution   The period is 1000 ms (or 1 s), the sum of the two `delay()` invocations.

2.3   Example Digital Output Use Cases

2.3.1   LED Indicator

One of the simplest digital output devices one can imagine is a light that is either on or off. LEDs (light emitting diodes) are a common light source that are easy to control from a microcontroller's digital output pin. Figure 2.3 shows the schematic for controlling a single LED from pin 17 of the microcontroller (there is nothing special about pin 17, other than it must be usable as a digital output pin and it is the pin number from the example code in Figure 2.1).

If the sketch from Figure 2.1 is executed on the microcontroller that has the schematic from Figure 2.3 constructed, the LED will light up in response to the `digitalWrite(doPin,HIGH)` call. This is because a positive voltage is presented to the anode of the LED and zero volts are presented to the cathode (which is at GND potential). Generally, the HIGH voltage out of the microcontroller (+5 V) is too large for the LED, and it is possible to damage the

13
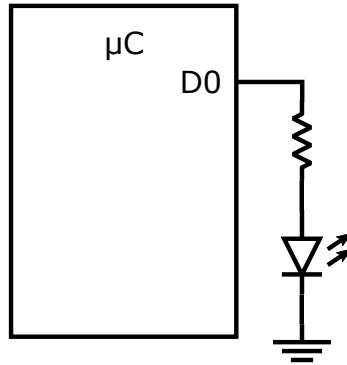
Figure 2.3: Schematic diagram of digital-output-controlled LED with active high control polarity. The anode is the topmost terminal of the LED and the cathode is the bottom terminal.

LED or the microcontroller, so we use the current limiting resistor to ensure that does not happen. One half-second later, the `digitalWrite(doPin,LOW)` call will cause the LED to become dark. With zero volts on the anode and zero volts on the cathode, the LED will not light.

In this example, when the digital output is HIGH, the LED is on, and when the digital output is LOW, the LED is off. This design choice is commonly called *active high*, meaning that the action (here, lighting the LED) happens when the signal is high. That is a completely arbitrary choice, however. Consider the schematic diagram shown in Figure 2.4, which gives an alternative design. In this case, the digital output is connected to the cathode side of the LED, and the anode side goes to +5 V. With this design, an output HIGH, gives 5 V on both the anode and the cathode of the LED, and it will stay dark. An output LOW, on the other hand, provides 0 V to the cathode side of the LED, which will cause it to light up. This design choice is commonly called *active low*. As a result of the alternative schematic connections, the operation of the light as a function of the digital output polarity has been reversed.

What this shows is that while there is a direct one-to-one relationship between the digital output value (HIGH or LOW) and the LED being controlled (on or off), the mapping between these two is determined by the design of the electrical circuit(s) that connect them.
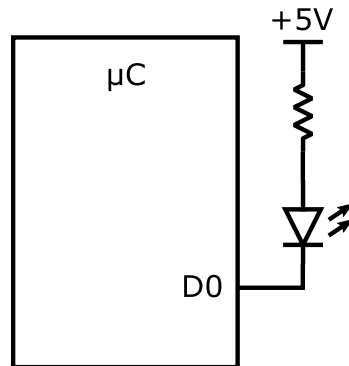
Figure 2.4: Schematic diagram of digital-output-controlled LED with active low control polarity. In this case, the anode of the LED is tied to +5 V and the cathode is tied to the digital output pin.

2.3.2 Buzzer

A commonly used technique to generate an audio signal is through the use of a buzzer. When a voltage is applied across the buzzer, it makes a sound, and is silent otherwise. This is a perfect example of a digital output.

The schematic diagram of a buzzer output is shown in Figure 2.5. When the digital output is HIGH, the buzzer makes sound, and when the digital output is LOW, the buzzer is silent.
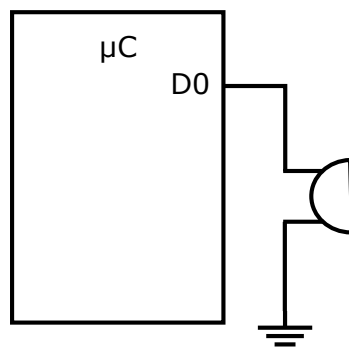
Figure 2.5: Schematic diagram of digital-output-controlled buzzer.

15

2.3.3 Relay

A relay is another device that can be controlled with a digital output. A relay has a low-voltage control side that turns on or turns off a set of mechanical contacts that can be used to control high-voltage devices, such as devices that use 110 V AC power from the wall socket. This might include motors, pumps, heating elements, etc.

The schematic diagram of a relay output is shown in Figure 2.6. When the output is HIGH, the relay coil is energized, which causes the switch to make contact. This forms a closed circuit between the two terminals on the right. When the output is LOW, the switch is open, so the two terminals form an open circuit (i.e., they do not conduct). Note, the diode shown across the relay coil is a protection diode, which often is manufactured as part of the relay assembly. Other times it must be included by the circuit designer.
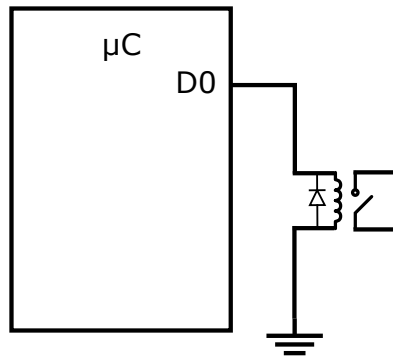
Figure 2.6: Schematic diagram of digital-output-controlled relay.

The purpose of a relay, in many cases, is to allow a low-voltage device (like a microcontroller) to control a high-voltage device (e.g., 110V powered equipment). This has its limitations, however, as a relay is a mechanical device, and will wear out with repeated usage much more quickly than an all-electronic approach.

Note that the relay we used has "normally open" (NO) contacts, meaning that when the coil is not energized, the contacts are open, and they close only when the coil is energized. There are also relays manufactured with "normally closed" (NC) contacts, meaning that when the coil is not energized, the contacts are closed, and they open only when the coil is energized.

16

Practice Problem   Redraw the schematic of Figure 2.6 so that it is controlled with an active low signal (i.e., instead of a HIGH output turning the relay on, a LOW output turns the relay on).

Solution   Here is one possible solution to the problem:
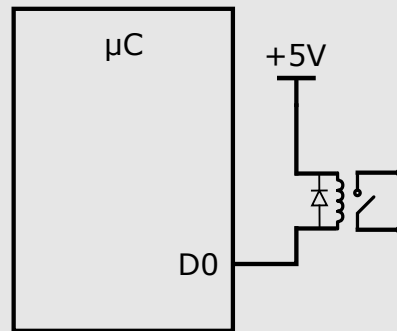
Pump

Figure 2.7 shows the schematic diagram of a relay controlling a pump that is powered by 110 V AC. When the digital output is HIGH, and the relay is energized, the pump is powered.

Heater

Figure 2.8 shows the schematic diagram of a relay controlling a 110 V AC powered heating element. It works very much the same as the pump example above.

Motor

The schematic diagram for controlling a very small motor is shown in Figure 2.9. As with the relay, there is a diode connected across the terminals of the motor for protection of the electronics. Similar to a relay coil, the motor is an inductive load, which means a fast turn-off can be damaging to the controlling electronics if not protected.

As with all of the other digital-output-controlled devices, the motor is on, running at its maximum speed, when the digital output is HIGH and is off,

17
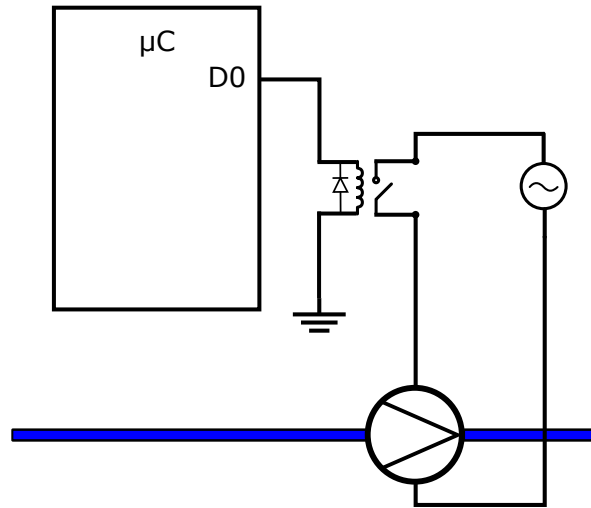
Figure 2.7: Schematic diagram of digital-output-controlled relay driving a pump.
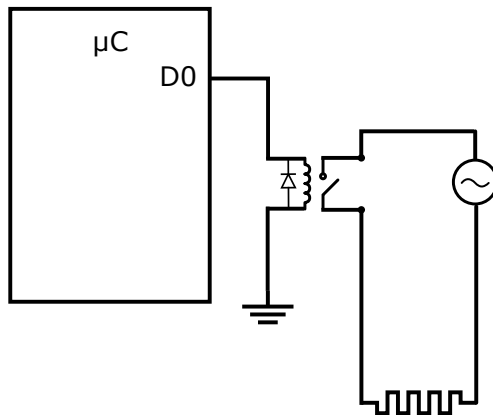
Figure 2.8: Schematic diagram of digital-output-controlled relay driving a heater.

stopped, when the digital output is LOW. In Chapter 4 we describe how to exert more fine-grained control over the speed of the motor.
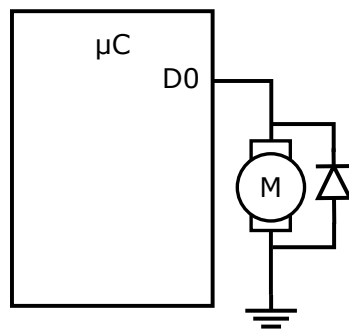
Figure 2.9: Schematic diagram of directly connected 5 V DC motor.

3   Digital Input

Probably the simplest form of interaction between a computer and the physical world is for the computer to sense (i.e., measure) some property of the world. Given that the internal representation of whatever thing we measure is going to be binary, the easiest things to measure are those that are readily represented in a binary system. Sensing opportunities that have this property are those for which there are only two options.

For example, consider a proximity detector that is placed on an assembly line. Its job is to determine whether or not there is a manufactured widget in front of it (i.e., in the *proximity* of the sensor). The answer is either "yes" or "no." Another example would be an emergency stop button on that same assembly line. In this case, a human is either pressing the button or not. Again, the answer is either "yes" or "no."

For each of these possible inputs, the information present can be represented inside the computer using a single binary bit, a 0 or a 1. The meaning of 0 or 1 will depend upon the specifics of the measurement being made. E.g., for the proximity detector, 0 might mean "not present" and 1 might mean "present." Likewise, for the emergency stop button, 0 might mean "not pressed" and 1 might mean "pressed."

3.1   Why Digital Inputs?

Note that the action that the computer will take in response to the example sensor inputs above might very well be radically different. When the computer senses the proximity detector input transitioning from "not present" to "present," it might simply increase an internal counter that is keeping track of inventory. Alternatively, when the computer senses the emergency stop button transitioning from "not pressed" to "pressed," its responsibility at that point is likely to halt the motion of the assembly line (which it would likely do via the use of a digital output). The bottom line is that the computer cannot

do any of these things unless it is making the relevant measurement in the first place. Measuring some property of the physical world has enabled the computer to do things it otherwise could not do.

3.2 Hardware

While the specific hardware required for any particular digital measurement is clearly dependent upon the type of measurement that is being made, a common digital input is a pushbutton or a switch. Figure 3.1 shows a commonly used circuit for interfacing a pushbutton to a microcontroller input pin.
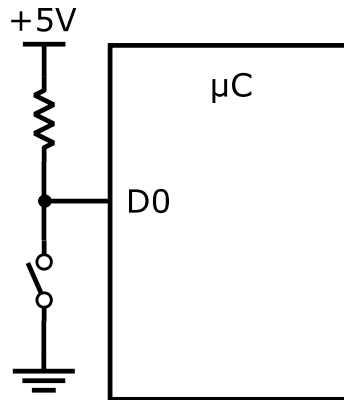
Figure 3.1: Schematic diagram of circuit that interfaces a pushbutton input to a microcontroller digital input pin.

In the figure, when the pushbutton is not being pressed, it creates an open circuit (i.e., no current can flow), because the input pin of the microcontroller is in a high impedance state when configured as an input, and the resistor pulls the voltage at the input pin up to +5 V (the power supply voltage). When the pushbutton is being pressed, it shorts the input pin to 0 V (ground). As a result, the input pin has a low voltage potential when the button is pressed and a high voltage when the button is not pressed.

This is an example of an active low design. The action is pressing the button, which yields a low input voltage on the microcontroller input. It is entirely reasonable to implement an active high design as well, which would involve swapping the positions of the switch and the resistor in Figure 3.1.

3.3 Software

As stated in Chapter 2, most of the pins on the microcontroller serve multiple purposes, and it is our responsibility to configure the pin prior to use. To read a digital input in software, we must first configure the pin as a digital input.

Configuring a digital input pin is accomplished using the `pinMode()` function. It takes two arguments, the first is an `int` identifying the pin number and the second is an `int` indicating the pin mode. For the mode, the constants `INPUT` or `INPUT_PULLUP` indicate the pin is to be a digital input.

The typical use case is to use the `INPUT` pin mode. This would be the appropriate mode to use for the circuitry depicted in Figure 3.1. However, the use of a switch (or some other circuit) to actively pull the voltage low in combination with a resistor that passively pulls the voltage high (i.e., an active low design) is a use case that is fairly common. As a result, microcontrollers often provide the resistor built-in to the chip, and the `INPUT_PULLUP` mode tells the microcontroller to enable the built-in pullup resistor. In this way, the external resistor of Figure 3.1 is no longer needed, as the resistor is internal to the microcontroller. Because of the availability of built-in pullup resistors in many microcontrollers, active low inputs are a much more prevalent design choice, versus the active high alternative.

Once the pin has been configured, the `digitalRead()` function is used to perform the actual reading of the input. If the voltage at the pin is approximately 5 V (relative to the `GND` potential), then the `digitalRead()` function returns the constant value `HIGH` (which is defined as a 1). If the voltage at the pin is approximately 0 V, the the `digitalRead()` function returns the constant value `LOW` (which is defined as a 0). If the voltage at the pin is near the midpoint (≈ 2.5 V), the return value is indeterminate, and either a `HIGH` or a `LOW` might result.[1]

Figure 3.2 gives an example sketch that repeatedly reads from a digital input, writes the value to a digital output, and prints the value. Note that in the sketch, `value` is declared as an `int`. This is because `digitalRead()` returns `HIGH` or `LOW`, which are constants of type `int`.

---

[1]Actually, there are two values specified in the data sheet of the microcontroller that more precisely describe how voltages at the input pin are interpreted. Any voltage less than V_{IL} will return a 0 in software and any voltage greater than V_{IH} will return a 1 in software. Voltages between V_{IL} and V_{IH} give indeterminate results.

```
const int diPin = 16;        // digital input pin is 16
const int doPin = 17;        // digital output pin is 17
int value = LOW;             // input value

void setup() {
  pinMode(diPin, INPUT);     // set pin to digital input
  pinMode(doPin, OUTPUT);    // set pin to digital output
  Serial.begin(9600);
}

void loop() {
  value = digitalRead(diPin);  // read the input
  digitalWrite(doPin, value);  // set the output to value
  Serial.print("value = ");
  Serial.println(value);
}
```

Figure 3.2: Example digital input sketch.

3.4 Example Digital Input Use Cases

3.4.1 Switch

The interfacing of a mechanical switch to a microcontroller digital input was described in Section 3.2. Clearly, one use of mechanical switches is for user input. Alternative uses include limit switches, relays, etc.

3.4.2 Proximity Detector

A proximity detector is an input sensor that is capable of determining whether or not an object is within the "proximity" (nearby space) of the sensor. They can be built using a large number of different physical phenomena, including capacitive sensing, inductive sensing, optical sensing, radar, sonar, ultrasonics, and Hall effect sensing.

3.4.3 Beam Sensor

A beam sensor, sometimes called a breakbeam sensor, is a sensor that transmits a light beam (often infrared, or IR, so that it is not visible to humans) across an open space. A receiver on the opposite side of that space either

detects the transmitted beam (if there is nothing opaque in the light path) or doesn't detect the beam (if something is blocking the light path).

If the receiver is connected to a digital input pin, the microcontroller can sense whether or not the beam is obstructed. Beam sensors are used frequently to count objects moving down a production line or to ensure safety for exclusion zones (e.g., an automatic garage door being closed will reverse direction if the beam is broken).

3.5   Debouncing Mechanical Contacts

For the example use cases described in the previous section, we made the simplifying assumption that the input value can be used effectively in the form it comes to us from the external hardware. This is frequently the case, however, it is not always true. Consider the waveform illustrated in Figure 3.3. It was captured at the input pin of the circuit shown in Figure 3.1.
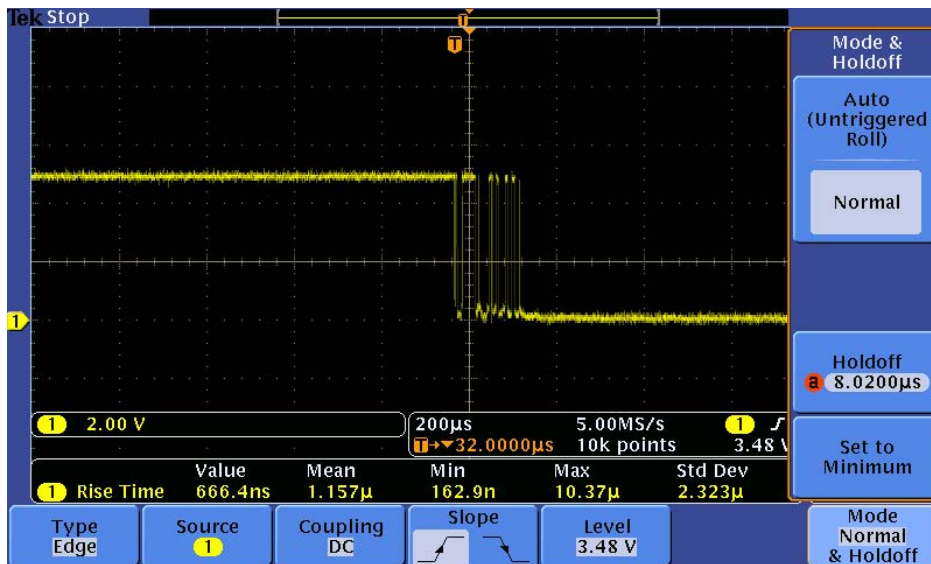
Figure 3.3: Oscilloscope trace illustrating input bounce.

The figure can be interpreted by understanding that the waveform shown is a plot of voltage vs. time, where the pushbutton was depressed at the time shown in the center of the figure. The vertical scale is 2 V/div, with 0 V shown by the marker with a "1" (indicating channel 1) on the left edge. Note that the initial signal voltage is therefore 5 V at the beginning of the waveform.

25

The horizontal scale is 200 μs/div, which implies that the change in the signal voltage starts approximately 1 ms from the beginning of the waveform. This is the time that the pushbutton was pressed. The result of pressing the pushbutton is that the signal makes several rapid changes between 5 V and 0 V, eventually settling at 0 V.

The reason this signal "bouncing" occurs is that the physical switch has mechanical contacts that, when pressed, come together, bounce apart, and then come together again, multiple times. The physical bouncing occurs at time scales of 10s of μs, while we are observing the signal over several hundred μs (almost 2 ms). Compare this to the time scale of the microcontroller, which executes multiple instructions every μs (approximately 16 instructions per μs if the processor's clock speed is 16 MHz).

Now consider what happens in the sketch shown in Figure 3.2. If the microcontroller loops fast enough, the output LED might flash on and off several times before it settles to its final value (the same as the input). However, this happens fast enough that we will never perceive it. We will only see the output LED go on (or go off), we won't see it flash on and off several times as it transitions.

But consider what happens if the sketch isn't just copying the input to the output, but is instead counting the number of times the input goes from high to low. In this case, one throw of the switch (which should be counted once by the sketch) will end up being counted multiple times.

The above describes a circumstance that happens all too frequently when a computer system is interacting with the physical world, especially sensing some property about the physical world. The electro-mechanical interface between the physical world and the microcontroller doesn't always provide information in a form that is immediately usable within software running on the microcontroller. Instead, we must do some computation on the input signal to ensure that it is in a form usable by the high-level software logic. In this example, if we want to count the number of times the switch is thrown, we need to "debounce" the raw input signal so that it correctly reflects the number of times the switch is thrown, not the number of times the mechanical contacts make or break the circuit due to bouncing.

For the switch bouncing illustrated in Figure 3.3, we observed that the time scale over which the input changes is approximately 200 μs. If we do some more investigation and conclude that the bouncing never lasts longer than 2 ms, one approach to safely counting switch throws is to ensure that the switch reads the same thing for 2 ms before the high-level software logic interprets the switch as being that value. A sketch that uses this technique is shown in Figure 3.4.

```
const int diPin = 16;        // digital input pin is 16
int oldValue = 0;            // previous input value
int newValue = 0;            // current input value
int value = 0;               // stable input value

void setup() {
  pinMode(diPin, INPUT);     // set pin to digital input
  Serial.begin(9600);
}

void loop() {
  newValue = digitalRead(diPin);  // read the input
  if (newValue == oldValue) {
    value = newValue;
  }
  Serial.print("value = ");
  Serial.println(value);
  oldValue = newValue;              // update old value
  delay(2);                         // wait 2 ms
}
```

Figure 3.4: Debounce digital input.

In the sketch, the loop executes every 2 ms. Each loop, the digital input is read, and compared to the value from the previous loop. Only when the two values match is the input considered stable.

3.6 Hardware vs. Software

Let's return to the economics requirement example of Chapter 1. If you recall, Figure 1.3 (repeated here as Figure 3.5) is a hardware circuit, constructed using logic gates, that implements the equation

$$E = AB + C \tag{3.1}$$

which is the same as Equation (1.4).

While Figure 3.5 shows a design that computes the economics requirement entirely in hardware, Figure 3.6 illustrates a design that computes the same economics requirement entirely in software. In the sketch, the inputs A, B,
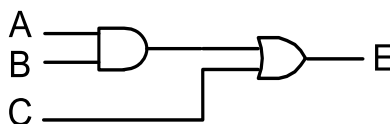
Figure 3.5: Gate-level schematic diagram of circuit that implements economics course requirement check.

and C are provided as digital inputs (on pins 14, 15, and 16) and the output E is made available as a digital output (on pin 17).

At the simplest level, this example illustrates the point that it is possible to build a system that does some logic computation (in this case a check of a student's economics requirement) either in hardware or in software. It is worthwhile to consider some of the differences between these two designs, however.

1. The hardware design only performs the given function, while the software design can have the logic changed without changing the physical system (it does, of course, require a change to the software sketch). This flexibility of function is one of the clear strengths of a design that relies on software for the implementation of the logic.

2. Using modern technology to construct the hardware design, the delay in updating E when one of the inputs changes is only a few nanoseconds, while the software version must execute a full iteration of the loop (maybe a microsecond or more). Whether or not this difference in delay is important is dependent upon the problem; however, it is fairly typical that a software implementation of a design is frequently much slower than a dedicated hardware implementation of the same design.

```
const int Apin = 14; // input pin for A
const int Bpin = 15; // input pin for B
const int Cpin = 16; // input pin for C
const int Epin = 17; // output pin for E

boolean A = false; // student has completed Econ A
boolean B = false; // student has completed Econ B
boolean C = false; // student has completed Econ C
boolean E = false; // student has completed economics requirement

void setup() {
  pinMode(Apin, INPUT);  // A, B, and C are digital input
  pinMode(Bpin, INPUT);
  pinMode(Cpin, INPUT);
  pinMode(Epin, OUTPUT); // E is a digital output
}

// function to read input value and return as boolean type
boolean booleanDigitalRead(int pin) {
  if (digitalRead(pin) == HIGH) {
    return(true);
  }
  else {
    return(false);
  }
}

void loop() {
  A = booleanDigitalRead(Apin); // read input values
  B = booleanDigitalRead(Bpin);
  C = booleanDigitalRead(Cpin);
  E = (A && B) || C;      // economics logic expressed in software
  digitalWrite(Epin, E); // output result
}
```

Figure 3.6: Sketch that computes economics course requirement check.

4   Analog Output

In the previous two chapters, we have talked about outputs that have an impact on the physical world, and we have talked about inputs that sense or measure some property of the physical world. However, in both cases, we only considered two possible values for the input or the output. Internal to the microcontroller, we represented those values as 0 or 1. External to the microcontroller, there were only two physical states represented, "on" or "off," LOW or HIGH voltage, "pressed" or "not pressed" for the emergency stop button, "present" or "not present" for the proximity detector.

Clearly, there are many things we would like to consider controlling or sensing by our microcontroller that have many more than just two values. If I am controlling a motor, I would like the ability to tell it to run faster or slower, not just run or not run. An *analog output* is an output signal that can take on a range of values, not just two.

4.1   Why Analog Outputs?

The purpose of an analog output is to provide a continuously variable signal for the purpose of influencing the external environment in some way. If the analog output is connected to an LED, the brightness of the LED can be directly controlled. If the analog output is connected to a motor, the speed of the motor can be controlled. If the analog output is connected to a heating element, the quantity of heat generated can be controlled. (This last example won't work well directly attaching a heating element to the microcontroller pin, some power delivery circuitry is needed as well, but the principle is exactly the same.)

4.2 Relating Analog Output Values to Physical Reality

A very common form of analog output used in many microcontrollers (including the AVR microcontroller on an Arduino) is called *pulse-width modulation* or PWM. PWM enables a digital output pin to, in effect, provide an analog output value. This is accomplished by quickly changing the digital output back and forth between HIGH and LOW, controlling the fraction of time that the value is HIGH versus LOW so that that average value (averaged over time) corresponds to the desired analog output value. In circumstances where the desired changes in the analog output's value are substantially slower than the rate at which the digital output is being changed HIGH to LOW and LOW to HIGH, this technique can work quite well.

The term pulse-width modulation comes from the fact that to control the average value of the varying digital output, the microcontroller alters the width of output pulses. This is illustrated in Figures 4.1 to 4.3. Each of these figures shows a square wave, varying between 0 V and 5 V at a rate of 500 Hz (for a period of 2 ms).

In Figure 4.1, the width of the pulse is 50% of the total period. As a result, the average value of this output waveform is 2.5 V (it is 0 V for half of each period and 5 V for half of each period).

In Figure 4.2, the width of the pulse has been decreased to 10% of the period, giving an average value of 0.5 V (10% of 5 V). We have decreased the pulse width as the controlling mechanism so as to effect the average value.

Figure 4.3 illustrates the control going the other direction. Here, the pulse width has been set to 90% of the period, resulting in an average value of 4.5 V. By controlling the width of the pulse, we can vary the average value of the waveform so that it has any value we wish between 0 and 5 V.

Another term that describes the width of the pulse is the *duty cycle*, or the fraction of total period that the digital output signal is HIGH. The duty cycle for Figure 4.2 is 10%, for Figure 4.1 is 50%, and for Figure 4.3 is 90%.

For a PWM signal to work effectively as an analog output, the 500 Hz square wave needs to be averaged. The desired output isn't the square wave, it is the average of the square wave. There are a number of options for doing this in the circuits that are connected to the PWM output pin. One of those options is a low-pass filter, a circuit that suppresses the 500 Hz signal but allows the lower frequency signals (the average value) to pass through.

Using a simple filter that has a 20 ms time constant (i.e., 10 times greater than the period of the PWM waveform), the output of the filter is shown on the graph in Figure 4.4, along with the original square wave that is input to the filter.
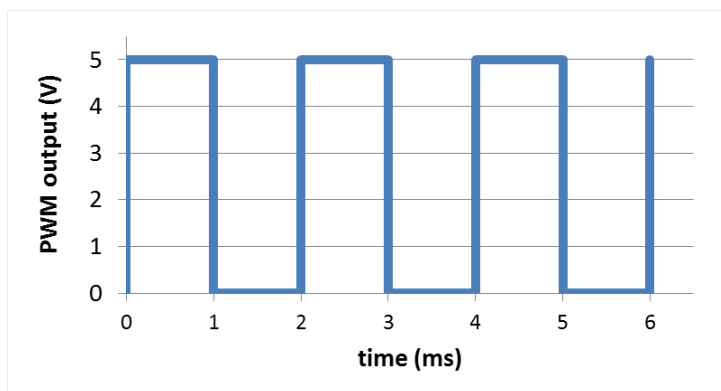
Figure 4.1: Pulse-width modulated analog output at 50% duty cycle.
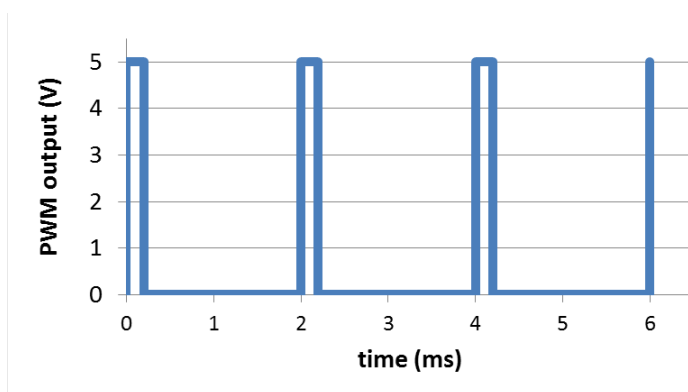
Figure 4.2: Pulse-width modulated analog output at 10% duty cycle.
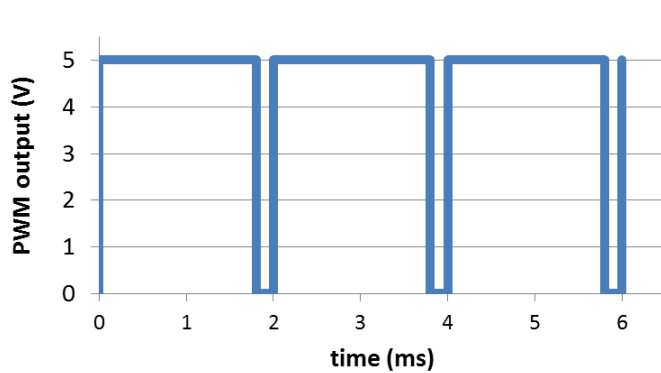
Figure 4.3: Pulse-width modulated analog output at 90% duty cycle.
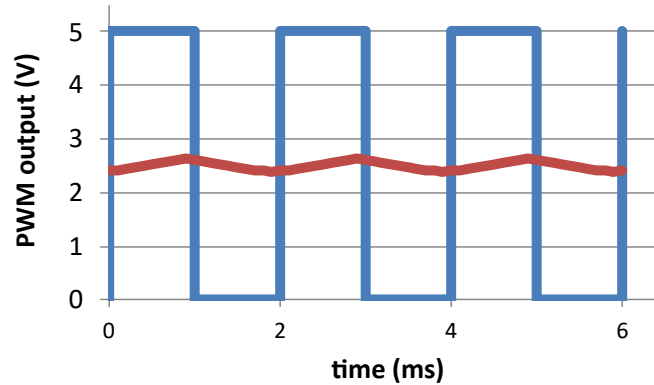
Figure 4.4: Pulse-width modulated analog output at 50% duty cycle, also showing a filtered output.

Next, consider what happens when we provide the waveforms described above to an output pin wired to an LED as in Figure 2.3. In reality, the LED is actually going on and off at a rate of 500 Hz. However, our eyes are nowhere near responsive enough to observe these changes. A recent study gave the fastest visual response measured to date as 13 ms [7]. Instead, our eyes (and brain) respond to the average light intensity, in effect doing the filtering job described above, and as the average value of the voltage varies from low to high, we perceive the LED to be varying in intensity from low to high. In other words, we have controlled the perceived intensity of the LED on an analog scale.

In the above example, the averaging was going on in our visual perception, our eyes and brain. This need not always be the case. If we are controlling the heat generated by a resistive element, the averaging will happen in the thermal response of the element (it is pretty unlikely to switch between hot and cold in less than 2 ms). Each use case needs to be considered individually to make sure that the averaging happens, and the mechanism might be quite different in each case.

4.3   Software

There are several digital I/O pins on the AVR microcontroller that directly support pulse-width modulated analog output functionality. Since the actual

output is a quickly varying digital output, the `pinMode()` routine is used to configure the pin to `OUTPUT` mode.

Once configured, the `analogWrite()` routine is used to control the analog value that is output. The first argument is the output pin, and the second argument is the analog value to be output (which can vary between 0 and 255).

Practice Problem What `analogWrite()` command will give the output waveform of Figure 4.1? Figure 4.2? Figure 4.3?

Solution To achieve an output duty cycle of 50% (for Figure 4.1), the output value should be 50% of 255, or 127. For a 10% duty cycle (matching Figure 4.2), we need to use 10% of 255, or 25, and for a 90% duty cycle (matching Figure 4.3), we need to use 90% of 255, or 229. Instructions to accomplish each of these in succession are shown below:

```
const int aoPin = 3;        // analog output pin is 3
...
analogWrite(aoPin, 127);    // output 50% of full scale
...
analogWrite(aoPin, 25);     // output 10% of full scale
...
analogWrite(aoPin, 229);    // output 90% of full scale
```

4.4 Example Analog Output Use Cases

4.4.1 Variable Speed Motor

Depending on the current required for the motor, one of two circuits can be used for a PWM output to drive a 5 V DC motor. If the motor current is less than 30 mA (this is a very tiny motor), the motor can be driven directly from the output pin of the microcontroller, as illustrated in Figure 4.5 (the same as Figure 2.9 in Chapter 2).

If, on the other hand, the motor current is greater than 30 mA, but less than 250 mA (this is much more common), the motor can be driven using a transistor circuit as shown in Figure 4.6.
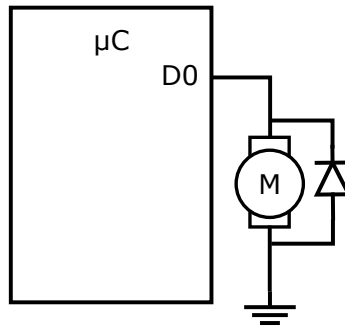
Figure 4.5: Schematic diagram of directly connected 5 V DC motor.
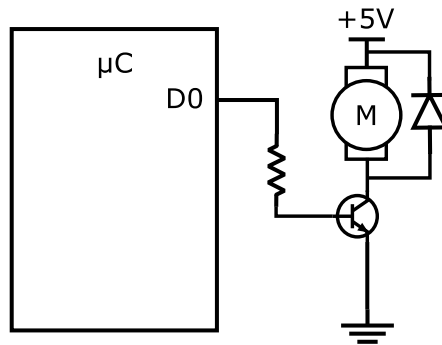
Figure 4.6: Schematic diagram of 5 V DC motor driven by a transistor.

Note than in both schematic diagrams, there is a diode connected across the terminals of the motor. This is important, since without it, there is a very good chance the microcontroller output (when directly connected) or the transistor (when it is used) will be damaged as the motor is turned on and off. Since the motor is an inductive load, it does a very good job of averaging the input square wave and responding to its mean (average) value.

The motor can be controlled as a digital output if desired. The statement

```
digitalWrite(pin,HIGH);
```

will turn the motor on, and the statement

```
digitalWrite(pin,LOW);
```

will turn the motor off.

More generally, we can control the speed of the motor by using a PWM output. The statement

36

```
analogWrite(pin,127);
```

provides half-power to the motor. Here, the averaging is happening in the motor itself. It responds to the average value of the output.

4.4.2 Loudness

The use of a PWM output to control the volume of an audio signal is illustrated in Figure 4.7. In this example, the audio input signal is sent through a variable gain amplifier before being delivered to a speaker. The control input of the variable gain amplifier is set by the microcontroller output pin.
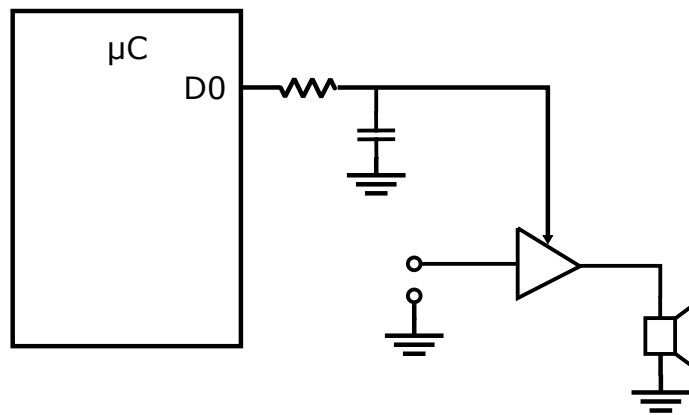
Figure 4.7: Schematic diagram of PWM controlled audio amplitude.

Recall that the frequency of the PWM output is 500 Hz, which is well within the frequency range of human hearing. Unlike the LED drive, in which our eyes cannot respond to the speed of the pulses in the PWM signal, our ears are more than capable of hearing at 500 Hz, and it would significantly interfere with the signal being amplified. As a result, it is insufficient to rely on the output device (e.g., the speaker) to smooth out the 500 Hz PWM pulses.

To address this issue, we insert a circuit between the PWM output pin and the variable gain amplifier's control input pin. This circuit averages the voltage coming out of the microcontroller, and provides a stable signal to the amplifier's control input (smoothing out the 500 Hz variations). This type of filter is called a *low-pass* filter, because it allows lower frequencies to pass through the filter and blocks higher frequencies. The boundary between the low and high frequencies is determined by the value of the resistor and the capacitor in the filter.

Note in the example of Figure 4.4, the output of a filter with a time constant of 20 ms (10 times the frequency of the PWM waveform) is illustrated. It is clear that the output of the filter still has a varying signal at 500 Hz (i.e., filters are not perfect). A practical filter will have a much longer time constant (e.g., 100 times the PWM frequency).

The volume of the audio signal sent to the speaker can now be controlled as follows,

```
analogWrite(pin,volume);
```

where the value of `volume` can range from 0 (minimum gain) to 255 (maximum gain).

5   Analog Input

Given that the previous three chapters have discussed digital output, digital input, and analog output, what else could this chapter possibly cover? As you have no doubt guessed by now, an analog input is a mechanism whereby a continuous signal is input into a computer.

As with the analog outputs described in the previous chapter, it is not possible to represent an infinitely varying signal in a computer, which is limited to binary number representations. As a result, the continuously varying analog signal is discretized as part of the analog input process. The subsystem that does this is called an *analog-to-digital converter* (frequently shortened to *A/D converter*). An A/D converter takes a continuous input (typically a voltage) specified over a given range and translates that input into a (digital) binary number.

We talk about the range of input values by specifying an analog reference voltage (often designated V_{REF}), and the nominal input range is therefore between 0 and V_{REF} V. The range of output values is specified by the number of bits in the resulting binary number. If the A/D converter is described as having n bits, the range of output values is 0 to $2^n - 1$, so an 8-bit A/D converter's output would range 0 to 255 (0 to $2^8 - 1$). The output values of the A/D converter are frequently called *A/D counts*, a convention we will follow.

5.1   Why Analog Inputs?

The purpose of analog inputs is fairly straightforward. Any physical measurement that has a range of possible values is a candidate for using an analog input. This might include temperature, distance, pressure, mass, humidity, acceleration, brightness, pH, force, or any other measurement you might want to consider.

5.2   Counts to Engineering Units

With a 10-bit A/D converter, the values that can result from a conversion range from 0 to 1023 (0 to $2^{10} - 1$). Rarely, however, are we interested in the raw values from the A/D. More often, we wish to convert those raw values (called A/D counts) into engineering units that are meaningful in terms of the physical measurement made in the real world.

Consider the analog input shown in Figure 5.1. It shows a physical sensor (let's assume in this case it is a weight scale), some amplification or signal conditioning, and an input into one of the analog input pins of the microcontroller.
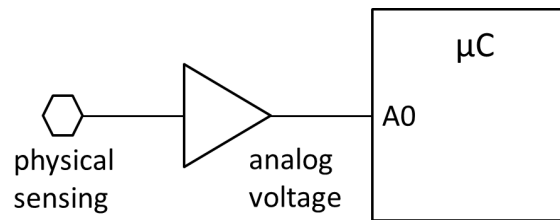
Figure 5.1: General analog input.

5.2.1   Input Range and Linear Transformation

The transformation from weight (in lbs) to voltage (in V, at the analog input pin) is shown in Figure 5.2. For this combination of weight scale and signal conditioning circuitry, at 0 lb the analog voltage is 200 mV and at 100 lb the analog voltage is 4500 mV. This relationship is shown in the figure, with the weight on the x-axis and the analog voltage signal shown on the y-axis, and can be represented mathematically as

$$s = \frac{(4500 - 200)}{(100 - 0)} \cdot w + 200$$

$$s = 43 \left[\tfrac{\text{mV}}{\text{lb}} \right] \cdot w + 200 \, [\text{mV}] \tag{5.1}$$

where s is the analog signal (in mV) and w is the measured weight (in lb). The units for each of the equation coefficients are enclosed in square brackets.

With a 5 V analog voltage reference, the A/D converter maps 0 V input to 0 A/D counts and 5 V input to 1023 A/D counts. This relationship is shown in Figure 5.3. In the figure, the x-axis shows the analog input signal and the y-axis shows the A/D counts. The two points shown correspond to the
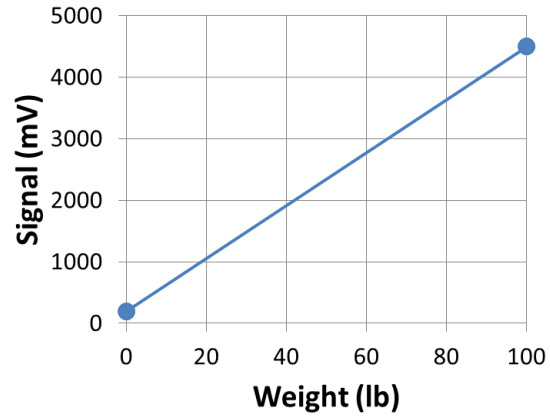
Figure 5.2: Analog circuit response.

values associated with 0 lb on the sensor (200 mV, 41 A/D counts) and 100 lb on the sensor (4500 mV, 921 A/D counts). This relationship is represented mathematically as

$$c = \frac{(1023 - 0)}{(5000 - 0)} \cdot s + 0$$
$$c = 0.2046 \left[\tfrac{\text{cnt}}{\text{mV}} \right] \cdot s \qquad (5.2)$$

where s is again the analog signal (in mV) and c is the A/D counts (cnt).
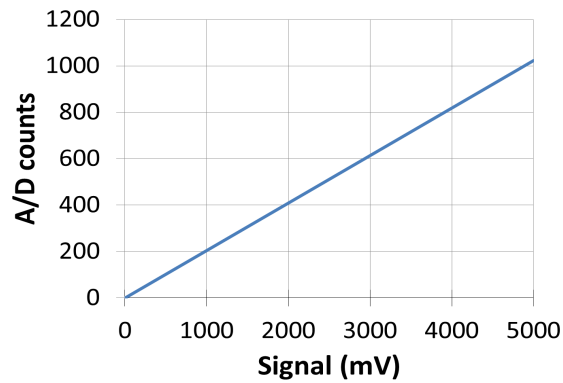
Figure 5.3: Analog to digital converter response.

Table 5.1: Parameters for `analogReference()`.

Parameter	Analog Reference Voltage
DEFAULT	5 V
INTERNAL	1.1 V
EXTERNAL	AREF pin value

Note: These values are for the Arduino Uno and vary for other platforms.

The complete response can now be represented mathematically by substituting Equation (5.1) into Equation (5.2).

$$c = 0.2046 \cdot s$$
$$c = 0.2046 \cdot (43 \cdot w + 200)$$
$$c = 8.7978 \left[\tfrac{\text{cnt}}{\text{lb}} \right] \cdot w + 40.92 \, [\text{cnt}] \tag{5.3}$$

While the above equation describes the A/D counts that will result for a given weight, we are actually interested in the opposite direction. The software would like to know the weight, and what it has are counts. We can get that by simply inverting Equation (5.3).

$$w = 0.1136648 \left[\tfrac{\text{lb}}{\text{cnt}} \right] \cdot c - 4.6512 \, [\text{lb}] \tag{5.4}$$

This gives weight (in lbs) given A/D counts.

The above example makes the implicit assumption that the range of A/D counts (which is 0 to 1023) happens over an input voltage range of 0 to 5 V. This is not always the case. The top end of the voltage range (that corresponds to 1023 A/D counts) is adjustable. We will see how to do this in the following section.

5.3   Software

Figure 5.4 shows a sketch that utilizes the analog input hardware described in the previous section. The scale of the analog range is set using `analogReference()`. With the parameter `DEFAULT`, the analog input range is configured to be 0 to 5 V. Table 5.1 shows other possible settings.

The `loop()` code reads the analog input value, converts the value into engineering units (lbs in this case) using Equation (5.4), and prints both the A/D counts and the weight.

```
const int aiPin = 14;          // analog input A0 is pin 14
int rawValue = 0;              // raw input value
float weight = 0.0;            // weight in lbs

void setup() {
  analogReference(DEFAULT);    // set analog range
  Serial.begin(9600);
}

void loop() {
  // read the input
  rawValue = analogRead(aiPin);
  // compute weight
  weight = (float)(0.1136648 * rawValue - 4.6512);
  // print results
  Serial.print("Raw value = ");
  Serial.print(rawValue);
  Serial.print(" Weight = ");
  Serial.print(weight);
  Serial.println(" lbs");
}
```

Figure 5.4: Example analog input sketch.

5.4   Example Analog Input Use Cases

In this section, we will illustrate the use of analog inputs for three different purposes. To illustrate a variety circumstances, each use case will have some unique property built into the example.

5.4.1   Temperature

This first example illustrates the use of a different reference voltage, which is set using the `analogReference()` call.

Consider a temperature probe that generates an output voltage with the following parameters: 10 mV/°C voltage change with temperature and 0 V at 0 °C. We are interested in measuring liquid water, so the range of temperatures we need to consider are 0 to 100 °C. The above lets us construct an equation

43

for the voltage response of the probe as follows:

$$s = 10 \left[\tfrac{\text{mV}}{\text{°C}} \right] \cdot t \tag{5.5}$$

where s is the analog voltage signal into the A/D converter and t is the temperature in °C.

If we use `INTERNAL` as the parameter to `analogReference()`, the top of the voltage range is 1.1 V, which will be just above the highest temperature we wish to read (1000 mV at 100 °C). The conversion from analog voltage to A/D counts is therefore

$$c = \frac{(1023 - 0)}{(1100 - 0)} \cdot s \tag{5.6}$$

$$c = 0.93 \left[\tfrac{\text{cnt}}{\text{mV}} \right] \cdot s \tag{5.7}$$

which gives

$$c = 0.93 \cdot (10 \cdot t) \tag{5.8}$$

$$c = 9.3 \left[\tfrac{\text{cnt}}{\text{°C}} \right] \cdot t \tag{5.9}$$

as the expression for A/D counts given temperature, and

$$t = 0.1075 \left[\tfrac{\text{°C}}{\text{cnt}} \right] \cdot c \tag{5.10}$$

as the expression for temperature given A/D counts. The code to convert A/D counts into engineering units (temperature in °C) is therefore

```
temp = (float) (0.1075 * rawValue);
```

where `temp` is a `float` representing temperature and `rawValue` is an `int` that has the raw A/D count value.

5.4.2   Level

This second example shows an analog input in which the increasing signal goes the opposite direction. Consider the liquid level sensor of Figure 5.5. In this example the height of the liquid vessel is 4 cm, and the top of the sensor is 5 cm above the bottom of the vessel. The sensor circuit's voltage response is 1 V/cm, measured from the top of sensor to the level of the liquid in the vessel.

As a result of this mode of operation, the circuit will read 5 V (5000 mV) when the vessel is empty and 1 V (1000 mV) when the vessel is full. We are
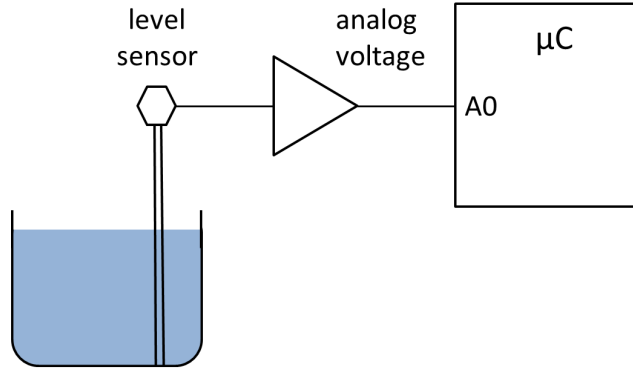
Figure 5.5: Liquid level measurement.

interested in knowing the level of the liquid in the vessel. We can express the voltage signal, s, as a function of liquid level, L, as follows.

$$s = \frac{(1000 - 5000)}{(4 - 0)} \cdot L + 5000 \tag{5.11}$$

$$s = -1000 \left[\tfrac{\text{mV}}{\text{cm}}\right] \cdot L + 5000 \, [\text{mV}] \tag{5.12}$$

using the points (0 cm, 5000 mV) and (4 cm, 1000 mV) to define the linear response. Returning to a 5 V reference, this equation gets substituted into Equation (5.2) to yield

$$c = 0.2046 \cdot (-1000L + 5000) \tag{5.13}$$

$$c = -204.6 \left[\tfrac{\text{cnt}}{\text{cm}}\right] \cdot L + 1023 \, [\text{cnt}] \tag{5.14}$$

as the expression for A/D counts given level, and

$$L = -0.0048876 \left[\tfrac{\text{cm}}{\text{cnt}}\right] \cdot c + 5 \, [\text{cm}] \tag{5.15}$$

as the expression for level given A/D counts. The code to convert A/D counts into engineering units (level in cm) is therefore

```
level = (float)(-0.004876 * rawValue + 5);
```

where `level` is a `float` representing the liquid level in cm.

Note that this analog reading really does work the same as the previous two examples (sensing weight and temperature), with the only distinction being that the slope of the response curve is negative. Therefore, the A/D counts go down as the liquid level goes up.

45

5.4.3 Acceleration

In addition to using the internal A/D converter, it is often the case that we interface a microcontroller to other subsystems that have been optimized for a particular purpose. In the microcontroller world, we frequently use what is known as the I^{2}C bus to connect the microcontroller to peripheral devices, such as sensors and actuators.

As an example, the MMA8451Q is an integrated circuit (manufactured by Freescale) that functions as an accelerometer. The block-level diagram and directional reference are shown in Figure 5.6, which are reproductions of Figures 1 and 2 of the part's data sheet.
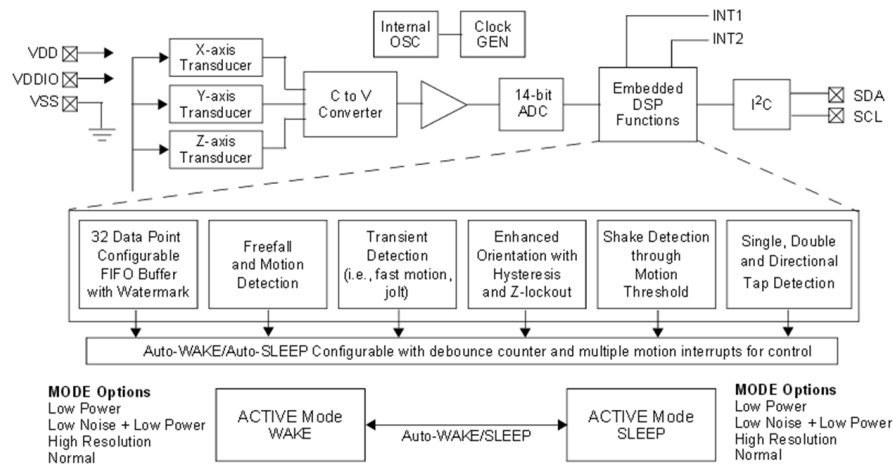
Figure 1. Block Diagram
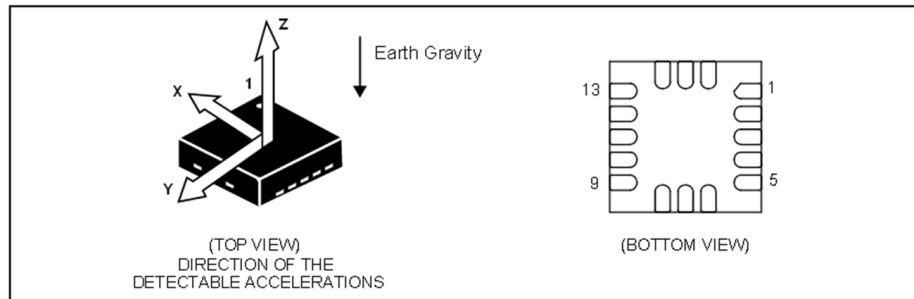
Figure 2. Direction of the Detectable Accelerations

Figure 5.6: Block diagram and directional reference for Freescale MMA8451Q accelerometer(from the data sheet).

Observe that the part is actually noticeably more complex than our Arduino processor. It has a built-in processor of its own (that performs the "embedded DSP functions" on the block diagram), in addition to three transducers (oriented along each axis), analog-to-digital conversion, and various support functions.

The A/D converter that is built in to the accelerometer is 14 bits, so the values range from 0 to 8191 (0 to $2^{14} - 1$). In addition, the processor that is built in to the accelerometer will perform the scale conversions, returning acceleration in m/s^2. In either case, we access the information from the accelerometer using libraries provided by the manufacturer.

6   Timing

There are lots of ways to reason about the passage of time in computer systems, generally. At one end of the spectrum, how much time a program takes to execute is only an issue if it becomes long enough to be distracting to the user. For example, if the task of a program is to add the value of someones assets and subtract the value of his/her debts to determine net worth, until the program takes longer to run than it takes the human to enter the program's inputs and observe the program's outputs, how long it takes to run is almost irrelevant. As a user, what do I care if it completes in 1 millisecond or in 10 milliseconds? TV screens take more than 30 milliseconds to update each frame, and to our human eyes that looks like smooth and continuous motion.

If the amount of time that a program takes to execute is primarily a matter of convenience for the user, we refer to the execution time as a *non-functional* property of the program (i.e., it is not part of the *function* that the program is expected to perform). Another way to say this is that how long the program takes to run is not formally part of the correctness criteria of the program. It is judged to be providing a correct answer even if it takes a long time to get to that answer.

At the other end of the spectrum, there are computer programs for which *when* they provide a result is just as important as the value that they provide. Consider a computer program that is managing the flight control surfaces on a high-performance aircraft. If the program tells the aileron to move up (e.g., because the pilot has moved the control stick), but provides that output too late, the aircraft can crash. This is a much more serious result than simply user inconvenience.

When time is an explicit component in the correctness criteria (i.e., time *is* a functional property), we refer to it as a *real-time* program. Real-time programs are often divided into two classes. The first, called *hard real-time*, are those for which serious dire consequences will result if some timing deadline is missed. This would be the case for our aircraft control example above. The second, called *soft real-time*, are those for which there is some degree

of slack, or forgiveness, in the timing requirements. A good example here is video playback. If you are watching a video and one or two individual frames are missing, you will never perceive it and the playback experience will be a positive one (at approximately 30 frames per second, you'll never miss it). It is not until lots of frames are missing (or delayed) that you will start complaining about the viewing experience. Here, timeliness is clearly part of the correctness criteria for the playback software. However, occasionally missing a few of the timing specifications isn't a life-and-death matter.

For computer systems that interact with the physical world, it is quite common for timing to be an important part of the functional properties of the programs we run. Sometimes they might be hard real-time specifications, other times they might be soft real-time requirements. Most of the time, however, they will include time in some way.

6.1 Execution Time

Any computer program takes time to execute. As described in Chapter 10, it is physically possible to count the individual instructions that the computer executes, and if you know how much time each instruction takes, it is possible to know (with surprisingly good precision) how long a computer takes to execute a specific instruction sequence.

There are two major problems with this approach in practice. First, in many cases we do not know ahead of time how many instructions will execute. As soon as there is a conditional branch in our program (e.g., an `if...then` statement or a `while` loop) for which the condition is dependent upon some input value, then different runs of the program will have different numbers of instructions to execute.

Second, only on the simplest processors do we actually know how much time each instruction takes to execute. On modern processors, there are a whole host of reasons why each instruction can take more or less time to execute. Variations in memory access time, execution pipeline bubbles, out-of-order execution, and contention for needed resources are but a few of the causes that limit our ability to know how much time each instruction takes before it is complete.

As a result, counting of instructions is only very rarely used as an effective mechanism for managing time within programs. In virtually all processors, from the most advanced multicore to the simplest microcontroller, there are dedicated circuits that are tasked with the job of measuring the passage of time. A simple example is a free-running counter that is incremented at a

given frequency. If the counter updates at 1 MHz, each microsecond (μs) the counter value increases by 1. (If f is the frequency, 1 MHz or 1,000,000 Hz in this case, and T is the period, then $T = 1/f = 1$ μs.) The pseudocode in Figure 6.1 then enables the program to know how much time has elapsed between two different points in the code (e.g., before and after a section of code we want to know how long it takes to execute).

```
startTime = readFreeRunningCounter()
// execute timed code
endTime = readFreeRunningCounter()
runTime = endTime - startTime
```

Figure 6.1: Measuring elapsed time with a free-running counter. The variable `runTime` indicates the execution time of the timed code, in time units dependent upon the free-running counter's frequency.

In the Arduino C environment, there are two functions that are available to access the free-running counter on the microcontroller. The first, `millis()`, returns the number of milliseconds since the last processor reset, and the second, `micros()`, returns the number of microseconds since the last processor reset. Both functions return a `long int` type since an `int` will quickly run out of space to store sufficiently large values (see Chapter 8).

6.2 Controlling Time

The discussion above enables us to measure the elapsed time of a section of code; however, frequently the task is to ensure that actions in a program take a specific amount of time, or happen at a given rate. A simple example is the flashing LED of Chapter 2, the code for which is repeated in Figure 6.2. This sketch does a reasonable job flashing the LED at 1 Hz. The `delay()` call takes one argument, the number of milliseconds to delay, and returns from the call approximately that many milliseconds later. We've used this technique several times already, not only in Chapter 2.

There are a pair of (related) limitations to this method of managing time within a program. The first limitation is that this loop will not really run at 1 Hz. Invariably, it will run somewhat slower than 1 Hz (i.e., the total time to execute the loop will be something more than 1 second). This is because there are instructions to be executed in the loop that are outside of the `delay()` call, and those instructions take time to execute. Both calls to `digitalWrite()`

51

```
const int doPin = 17;          // digital output pin is 17

void setup() {
  pinMode(doPin, OUTPUT);      // set pin to digital output
}

void loop() {
  digitalWrite(doPin, HIGH);   // set the output HIGH
  delay(500);                  // wait for 0.5 s (500 ms)
  digitalWrite(doPin, LOW);    // set the output LOW
  delay(500);                  // wait for 0.5 s
}
```

Figure 6.2: Simple timing loop.

are outside of `delay()`, and there is some non-zero overhead associated with the `loop()` construct as well.

The second limitation is that the 1 second loop time will grow any time additional functionality is added to the loop. Consider the addition of a single line of code,

```
Serial.println(millis());
```

which will print the current value of the free-running counter. By observing the sequence of counter values printed, we can then discern actually how long it takes to execute the loop. Since this new code takes additional time to execute, the amount of time spent in the loop has now been altered. Not only is it never going to be precisely 1 second, the amount that it grows from 1 second is dependent upon things like adding diagnostic statements.

Note that the timing errors that result from extra code execution accumulate from one loop to the next. Once we get behind, we never catch back up. We only get further and further behind. There is, of course, a better way to do this, and the next section describes a technique for controlling time in software that is dramatically more robust. It doesn't fix everything, but it works considerably better than the methods described above.

6.3   Delta Time

The use of the `delay()` routine to control time has the issues described above, that all the code that is outside the `delay()` call isn't accounted for in the elapsed time for the loop. We will next examine a more robust timing approach, called *delta time*, that avoids some (but not all) of these issues.

Like the example above, we will assume that the task at hand is to execute the loop once per second. We will simplify the task by no longer having multiple timed events within the loop, but only concern ourselves with the total loop time. The code to implement the delta time approach is shown in Figure 6.3.

```
const long deltaTime = 1000; // loop period (in ms)
long loopEndTime = deltaTime;

void setup() {
}

void loop() {
  if (millis() >= loopEndTime) { // one period is complete
    loopEndTime += deltaTime;
     // code to be executed once per iteration
  }
}
```

Figure 6.3: 1 Hz timing loop using delta time techniques.

Consider how this timing approach works. We maintain a variable that indicates when the next 1 s period will be complete, `loopEndTime`. When the free-running counter has exceeded this value, we know that our loop period has elapsed. When this happens, we update the end-of-loop time and proceed to run the code that should execute once per loop.

This is different than the `delay()` based approach above in several ways. First, as long as the code that is executed once per loop (call it an *iteration*) doesn't take longer than 1 s to run, the code within the `if` condition will faithfully execute once per second. This is true whether the "once per iteration" code takes 1 microsecond, 10 milliseconds, or 900 milliseconds. As long as it is less than 1 second, the timing is preserved.

Second, even an occasional excursion beyond 1 s by the "once per iteration" code doesn't necessarily have dire consequences. While the next iteration will

be delayed (by the amount the previous iteration was late in finishing), the logic of updating the `loopEndTime` by `deltaTime` each iteration ensures that subsequent iterations will revert to the once per second intended rate. This is reasonable operation for many soft real-time tasks (although it is certainly not sufficiently robust for hard real-time operation).

The approach of Figure 6.3 is not the only possible design for delta timing. There are a number of possible designs, each with slightly different properties (primarily of what happens when the timing isn't perfect, or an iteration is late). For example, in Figure 6.3, if an individual iteration is late, the next iteration will be back on the original schedule (i.e., 2 s after the start of the late iteration). Sometimes that isn't what we want to happen. Figure 6.4 illustrates an alternative design that has slightly different properties when an iteration is late. It also introduces another common convention, which is that we make a call to `millis()` once per `loop`, retaining the result of the call, and then use that value throughout the loop. This ensures that "time" (i.e., the software's concept of what time it is) doesn't change in the middle of `loop()`.

```
const long deltaTime = 1000; // loop period (in ms)
long loopEndTime = deltaTime;
long currentTime = 0;

void setup() {
}

void loop() {
  currentTime = millis();
  if (currentTime >= loopEndTime) { // one period is complete
    loopEndTime = currentTime + deltaTime;
    // code to be executed once per iteration
  }
}
```

Figure 6.4: Alternative 1 Hz timing loop using delta time techniques.

In this sketch, if an iteration is late, the 1 s time period restarts at the beginning of the next iteration. This is because we've added `deltaTime` not to the scheduled `loopEndTime`, but instead to the `currentTime`, which might be later than `loopEndTime`. I.e., the 1 s period restarts rather than tries to catch up.

The difference between the above two designs is illustrated in the timeline of Figure 6.5. In the figure, time is increasing to the right. Rectangles represent work performed during an iteration, the top line represents design A (from Figure 6.3), and the lower line represents design B (from Figure 6.4).
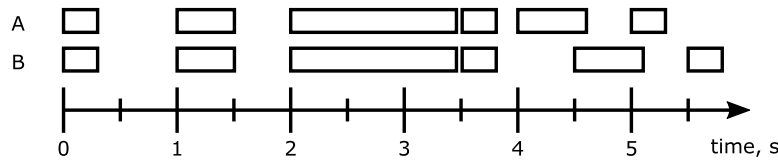
Figure 6.5: Timing diagram showing impact of two different designs for late iterations when using delta time techniques.

You can see that for the first four iterations, both designs do the same thing. The first three iterations are on time, the third iteration (starting at time 2 s) runs long, causing the fourth iteration to start at time 3.5 s. So far so good, but what happens next?

What happens next is that design A does its best to get back on the original schedule, with iterations starting aligned with each second (i.e., 4 s, 5 s, etc.). Design B doesn't do that, however. Instead, it starts a new 1 second period at time 3.5 s, so the next and subsequent iterations start 0.5 s later, at 4.5 s, 5.5 s, etc.

Either of these two designs can be appropriate, depending upon the specific task that is trying to be accomplished. This is one more case in which the correctness of a design depends not just on what goes on in software, but also what is going on in the real physical world and when.

We are now in a position to re-implement the simple LED flashing sketch from Figure 6.2 using delta timing. Figure 6.6 shows the revised sketch. Note that the loop period is now 0.5 s, and there is a new state variable, `LEDState`, to keep track of whether or not the LED is currently on.

6.4 Multiple Time Periods

One of the advantages of the delta time approach is its ability to generalize to multiple timing loops in a fairly straightforward manner. Consider the sketch of Figure 6.7, which supports two distinct things happening with two different time periods. Period A is 200 ms (5 times per second) and period B is 500 ms (2 times per second).

```
const int doPin = 17;         // digital output pin is 17
int LEDState = LOW;           // current LED output value
const long deltaTime = 500;   // loop period (in ms)
long loopEndTime = deltaTime;

void setup() {
  pinMode(doPin, OUTPUT);      // set pin to digital output
  digitalWrite(doPin, LEDState);
}

void loop() {
  if (millis() >= loopEndTime) { // one period is complete
    loopEndTime += deltaTime;
    LEDState = !LEDState;         // toggle LED state
    digitalWrite(doPin, LEDState);
  }
}
```

Figure 6.6: LED flash using delta timing techniques.

Notice that there isn't anything resembling a free lunch here. If both loops trigger at the same time (which they will in this example once per second), the code for the 500 ms period will execute *after* the code for the 200 ms period. If the code for the 200 ms period takes a significant length of time, the 500 ms code will be late. Whether or not this is a problem will, of course, depend on the overall properties of the system.

```
const long deltaTimeA = 200;  // loop period A (in ms)
const long deltaTimeB = 500;  // loop period B (in ms)
long loopEndTimeA = deltaTimeA;
long loopEndTimeB = deltaTimeB;
long currentTime = 0;

void setup() {
}

void loop() {
  currentTime = millis();
  if (currentTime >= loopEndTimeA) { // period A is complete
    loopEndTimeA += deltaTimeA;
    // code to be executed once per 200 ms
  }
  if (currentTime >= loopEndTimeB) { // period B is complete
    loopEndTimeB += deltaTimeB;
    // code to be executed once per 500 ms
  }
}
```

Figure 6.7: Two timing loops using delta time techniques.

7 Design Patterns

While we have been discussing design throughout the book, it is worth noting that one of the ways that we can implement good designs is to follow the lead of good designers that preceded us. Artists of every form, from musicians to painters, like to give credit to those who preceded them and provided inspiration, guidance, and examples to follow. As designers of computing systems, we shouldn't be any different. In this chapter, we will introduce some design patterns that are worthwhile to emulate.

7.1 Finite-State Machines

One of the design patterns that sees quite a bit of use is a *finite-state machine* (FSM), also called a *finite-state automaton*. An FSM is a computational abstraction in which things in the machine to be remembered are represented by individual *states*, and there are a finite number of states (hence the name).

The FSM abstraction can be quite useful, helping to keep a design manageable and straightforward to reason about. We will introduce the concept of a finite-state machine with a simple example. Consider the operation of a super-simple vending machine. There is only one thing for sale, and it costs 20¢. The vending machine can only accept nickels and dimes (no pennies, quarters, half-dollars, dollar coins, or bills). Also, it doesn't provide change (see, we said it was super simple, we weren't kidding).

A finite-state machine diagram that describes the operation of our vending machine is shown in Figure 7.1. It has 4 *states*, which are represented in the diagram as circles. Sometimes, this type of finite-state machine diagram is called a *bubble diagram*, with the circles representing individual states being called *state bubbles*. The table below the diagram gives the meaning associated with each state. In our case, the vending machine needs to remember how much money has been deposited so far (i.e., what is the accumulated value). One state (in our case, the "0¢" state in the upper left of the diagram) is

identified as the initial state, which is where the finite-state machine starts.

State	Meaning
0¢	$0.00 accumulated value (initial state)
5¢	$0.05 accumulated value
10¢	$0.10 accumulated value
15¢	$0.15 accumulated value
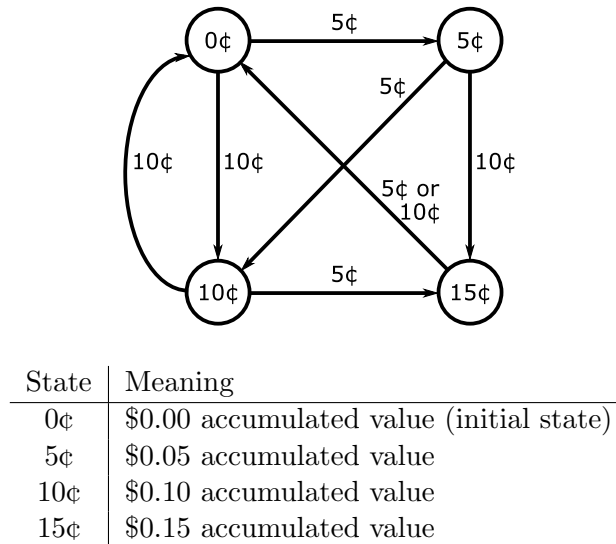
Figure 7.1: Vending machine finite-state diagram.

When someone who is trying to buy the item for sale inserts an individual coin, the operation of the FSM is indicated by the directed edges (arcs with arrowheads) that are labeled with the possible inputs. In this case, the possible inputs are two kinds of coin: (1) a nickel (shown as the label "5¢") or (2) a dime (shown as the label "10¢"). In this example, we are using common labels for both states and inputs (e.g., "5¢" is the name of both a state, representing 5 cents of accumulated value, and an input, representing a nickel). We can keep them apart in our mind by making sure we identify which one we're referring to each time. Alternatively, we could rename one or the other. The primary purpose of the names of both states and inputs is human understanding, so we can call them whatever we wish. In the implementation that is discussed below, we will use distinct labels for states and inputs.

Note that the abstraction only supports one input at a time. When the input arrives (coin is deposited), the FSM follows the edge indicated by the appropriate label, ending up in a new state. From the initial state, "0¢", if our customer inserts a nickel, the FSM moves to the state "5¢". Alternatively, if our customer inserts a dime, the FSM moves to the state "10¢".

Now, in a new state, the FSM is ready to receive another coin. If, for example, the first coin was a nickel and we are now in the state "5¢", if the next coin is another nickel, we will go to the state "10¢", and if the next coin

is a dime, we will go to the state "15¢".

There is an important principle worth noting at this point. If the first coin was a dime, the FSM ended up in the state "10¢", and if the first and second coins were both nickels, the FSM also ended up in the state "10¢". The memory of how the FSM got to the state "10¢" is not retained, however. The FSM only "remembers" that the accumulated value to this point is 10 cents, not whether it came in the form of a dime or two nickels.

If we are in the state "10¢" and the next coin is a dime, the customer has provided enough money to buy what we are selling, so the transaction can be completed (e.g., the vending machine can deliver the item), and the FSM can return to the state "0¢" so as to be ready for the next sale. In the diagram, we are only showing the state transition(s) to state "0¢", not the sales. However, it is possible to show that kind of information on a finite-state machine diagram as well (and we will do so in a later example).

Practice Problem If we start in the initial state "0¢" and three inputs are "5¢", "5¢", and "5¢", which states are visited after the first two inputs and what is the final state?

Solution After the first "5¢" input, the FSM is in the state "5¢". After the second "5¢" input, the FSM is in the state "10¢". After the third "5¢" input, the final state is "'15¢".

As described so far, the finite-state machine is just a computational abstraction. We can implement the finite-state machine in multiple ways. Just like we did in Chapter 3, where we were able to implement the logic for testing the economics requirement either in hardware or in software, we have the same options available to us for implementing finite-state machines. Whether constructed using hardware or software, the same computational abstraction is used.

Here, we will stick with a software implementation, which is shown in Figure 7.2. The state of the FSM is retained in a variable, `FSMstate`, which can have one of the following values:

`STATE_0_CENTS`   `STATE_5_CENTS`   `STATE_10_CENTS`   `STATE_15_CENTS`

and has initial value `STATE_0_CENTS`. There are several design choices for how to do this, but we will illustrate it with the use of an enumeration.

```
enum state {
  STATE_0_CENTS,STATE_5_CENTS,STATE_10_CENTS,STATE_15_CENTS}
};
state FSMstate = STATE_0_CENTS;
```

In the code provided in the figure, the routine `vend()` is invoked when an item has been sold. The `loop` executes the logic of the FSM. It first receives a coin, into `inputCoin`, and then decides what to do (what is the next state, and whether or not a sale has been completed) based on the current state and the value of the coin.

In this example, a `switch` statement is used to separate the logic for each state (one `case` per state), and `if` statements are used to distinguish different input coins. However, there is no requirement that it be implemented this way. We could have used `switch` statements for both state and input, for example.

At the bottom of the loop, the current state, `FSMstate` is updated to take on the value of the next state, `nextFSMstate`, and the loop repeats.

While the code in Figure 7.2 faithfully represents the functionality represented by the FSM diagram in Figure 7.1, notice how much easier it is to follow what is going on in the diagram (as opposed to the code). The abstract representation of the diagram is much clearer, less error-prone, and definitely the preferred way to communicate the operation of the vending machine to humans.

For example, what if we wanted to make a small change to the way the vending machine operates. Our current design doesn't support the return of change to the customer. For example, if we are in state "15¢" and the customer inserts a dime, the sale is made (and the FSM returns to state "0¢"), but the fact that the customer actually provided 25 cents is ignored.

In our alternative vending machine, shown in Figure 7.3, if the customer provides 25 cents, the 5 cents in change is made available to the next customer (OK, this really is pretty silly, but we are just illustrating a point here). In the FSM diagram, this is accomplished by simply adding one more edge (from the state "15¢" to the state "5¢") and changing the appropriate labels on the edges outbound from the state "15¢".

```
void loop () {
    inputCoin = inputNextCoin();
    switch (FSMstate) {        // code to implement FSM
    case STATE_0_CENTS:
        if (inputCoin == NICKEL) {
            nextFSMstate = STATE_5_CENTS;
        }
        if (inputCoin == DIME) {
            nextFSMstate = STATE_10_CENTS;
        }
        break;
    case STATE_5_CENTS:
        if (inputCoin == NICKEL) {
            nextFSMstate = STATE_10_CENTS;
        }
        if (inputCoin == DIME) {
            nextFSMstate = STATE_15_CENTS;
        }
        break;
    case STATE_10_CENTS:
        if (inputCoin == NICKEL) {
            nextFSMstate = STATE_15_CENTS;
        }
        if (inputCoin == DIME) {
            nextFSMstate = STATE_0_CENTS;
            vend();
        }
        break;
    case STATE_15_CENTS:
        nextFSMstate = STATE_0_CENTS;
        vend();
        break;
    }
    FSMstate = nextFSMstate;
}
```

Figure 7.2: Source code for vending machine FSM controller.
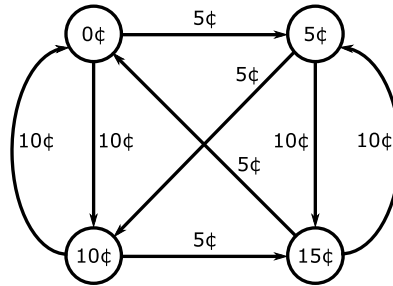
State	Meaning
0¢	$0.00 accumulated value (initial state)
5¢	$0.05 accumulated value
10¢	$0.10 accumulated value
15¢	$0.15 accumulated value

Figure 7.3: Alternative vending machine finite-state diagram.

Practice Problem   The modification of the FSM only impacts the activity in state "15¢". Alter the source code, starting at the line STATE_15_CENTS:, to implement this change.

Solution   The lines of code following STATE_15_CENTS: are now as follows:

```
case STATE_15_CENTS:
    if (inputCoin == NICKEL) {
  nextFSMstate = STATE_0_CENTS;
    }
    if (inputCoin == DIME} {
  nextFSMstate = STATE_5_CENTS;
    }
    vend();
    break;
}
```

Notice how by using an appropriate computational abstraction, a finite-state machine, simple changes to the functionality of the vending machine can be communicated with simple changes to the FSM diagram. In the next

example, we will expand the notion of the FSM computational abstraction to include output functionality as well (which was in the code for our vending machine example, but wasn't shown on the FSM diagram).

One of the things that finite-state machines are really good at is deciding whether or not a sequence of input symbols (i.e., characters) match a specified pattern. Take, for example, a Social Security Number (SSN). It typically is written in the following form:

```
xxx-xx-xxxx
```

where each `x` in the above notation represents a decimal digit between `0` and `9`. The diagram of Figure 7.4 illustrates an FSM that recognizes streams of input characters when they match this pattern.
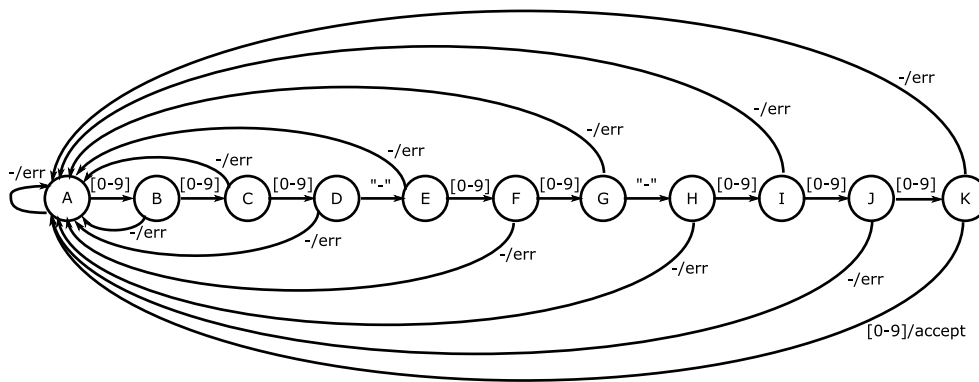
State	Meaning
A	Waiting for digit no. 1 (initial state)
B	Received 1st digit, waiting for digit no. 2
C	Received 2nd digit, waiting for digit no. 3
D	Received 3rd digit, waiting for dash no. 1
E	Received 3 digits and 1st dash, waiting for digit no. 4
F	Received 4th digit, waiting for digit no. 5
G	Received 5th digit, waiting for dash no. 2
H	Received 5 digits and 2 dashes, waiting for digit no. 6
I	Received 6th digit, waiting for digit no. 7
J	Received 7th digit, waiting for digit no. 8
K	Received 8th digit, waiting for digit no. 9

Figure 7.4: SSN finite-state machine diagram.

The states in this FSM are labeled `A` through `K`, and the meaning of each state is provided in the table below the diagram in the figure. The initial state is `A`. The are several important things different about this FSM diagram as compared to the diagrams we have seen so far, and they all relate to the labels on the edges. First, we have introduced a shorthand notation for indicating a range of characters. Take a look at the label on the edge from state `A` to state `B`. We wish to move from `A` to `B` for any character in the range `0` to `9`, and this is indicated in the diagram with the notation `[0-9]`.

Second, for any character not in the range `0` to `9`, we want to follow the other edge (which is a self-loop back to `A`). This is denoted by the `-` symbol (right before the `/`, which we'll address next). The `-` symbol can be interpreted to mean "an input not specified on some other outgoing edge from this state."

Third, for this self-loop edge (from `A` to `A`), after the input designation of `-` there is a `/` followed by the note `err`. The notation after the `/` indicates an *output* of the finite state machine. In this case, we are indicating than an error has occurred (i.e., the input sequence is not in the form of an SSN). Notice that when in state `K`, both outbound edges return to state `A`, and what distinguishes these two edges is whether or not we have an error (the edge with label `-/err`) or we have successfully recognized an SSN (the edge with label `[0-9]/accept`). Edges without a `/` don't generate any output when the edge is traversed by the FSM implementation.

Fourth, given that the `-` symbol represents "anything else" as an input character, when we are explicitly expecting a hyphen (e.g., when in state `D`) we denote that input symbol by putting it in quotes (i.e., `"-"`).

Fifth, while the `/` is used pretty universally to separate input symbols from output actions in finite-state machine diagrams, we pretty much made up the rest of the notation. Some designate ranges with a colon (e.g., `0:9` rather than `0-9`) and don't always include the square brackets. Others designate the "anything else" input by simply not indicating an input (i.e., leaving it blank). The important point here is to remember that the FSM diagram is intended to be read by humans, so as long as the reader has been informed of the conventions in use, he/she should be able to interpret the diagram correctly.

As we did for our first vending machine example, we will also show how the SSN recognizer FSM can be implemented in software. The code is shown in Figures 7.5 and 7.6. As in the earlier example, the current state of the FSM is retained in the variable `FSMstate`, which can only have the values `A` through `K` (as defined by either a set of `#define` statements or `const int` declarations earlier). The `boolean` functions `isNumber()` and `isDash()` determine whether or not the provided character matches the appropriate condition. The outputs are implemented in the routines `error()` and `accept()`. Both the `boolean`

66

functions and output routines need to be provided elsewhere in the sketch (e.g., the function `isNumber()` is available as the library routine `isDigit()`).

In the earlier example we made the point that the diagram was much simpler to follow than the code. While that is certainly still the case, there is an additional point to be made here. Note that the 11-state finite-state machine diagram fit in substantially less than a page, while the code consumed two full pages, and technically only includes the details for 6 of the 11 states.

7.2 Polling and Interrupts

In this section, we will talk about a pair of design patterns that are both quite prevalent in microcontroller systems and overlap to some degree in what they can accomplish. While they can be used for a variety of purposes, we will focus on their use in reading and interpreting input signals that originate external to the processor (mostly digital inputs).

7.2.1 Polling

Polling an input signal refers to the repeated, periodic reading of that signal. Figure 7.7 illustrates a simple example of polling a digital input pin (adapted from Figure 3.2), in which each `loop()` the digital input pin is read, the value retained in `diValue`, and various actions happen depending upon the current and previous digital input values.

One of the salient features of this design pattern is that the input is read only once per iteration. Throughout the rest of the iteration, when the logic depends upon the current input value it takes it from the variable `diValue`. In this way, we are assured that the "logical" value of the input doesn't change part way through the iteration (e.g., the `if` test passes and then the print statement shows the input as `LOW`).

This property is important if we are trying to perform logic as shown in the sketch. At the end of each iteration, the current `diValue` is saved in `prevValue`, so during the following iteration, `prevValue` retains the previous value of the digital input. In this way, testing for the previous value being `LOW` and the current value begin `HIGH` lets us reliably determine in which iteration does the input transition from low to high.

In this example sketch, when the low-to-high input transition is detected, the digital output is sent `HIGH` for 200 ms. Because we used a call to `delay()` to implement the 200 ms wait, the length of time required for each iteration varies. In many circumstances, polling an input with a variable period is not a good choice (e.g., when we are reading an analog input value and we

```
void loop () {
      char inputSymbol = inputNextCharacter();
      switch (FSMstate) {          // code to implement FSM
      case A:
            if (isNumber(inputSymbol)) {
                  nextFSMstate = B;
            }
            else {
                  nextFSMstate = A;
                  error();
            }
            break;
      case B:
            if (isNumber(inputSymbol)) {
                  nextFSMstate = C;
            }
            else {
                  nextFSMstate = A;
                  error();
            }
            break;
      case C:


            ...


      case D:
            if (isDash(inputSymbol)) {
                  nextFSMstate = E;
            }
            else {
                  nextFSMstate = A;
                  error();
            }
            break;
      case E:


            ...
```

Figure 7.5: Source code for SSN finite-state machine. (Continued, next page.)

```
      case G:
            if (isDash(inputSymbol)) {
                  nextFSMstate = H;
            }
            else {
                  nextFSMstate = A;
                  error();
            }
            break;
      case H:


            ...


      case J:
            if (isNumber(inputSymbol)) {
                  nextFSMstate = K;
            }
            else {
                  nextFSMstate = A;
                  error();
            }
            break;
      case K:
            if (isNumber(inputSymbol)) {
                  nextFSMstate = A;
                  accept();
            }
            else {
                  nextFSMstate = A;
                  error();
            }
      }
      FSMstate = nextFSMstate;
}
```

Figure 7.6: Source code for SSN finite-state machine (cont.).

```
const int diPin = 16;          // digital input pin is 16
const int doPin = 17;          // digital output pin is 17
int diValue = LOW;             // digital input value
int prevValue = LOW;           // previous input value

void setup() {
  pinMode(diPin, INPUT);       // set pin to digital input
  pinMode(doPin, OUTPUT);      // set pin to digital output
  digitalWrite(doPin, LOW);
  Serial.begin(9600);
}

void loop() {
  diValue = digitalRead(diPin);  // read the input
  if (prevValue == LOW && diValue == HIGH) {
     digitalWrite(doPin, HIGH);
     delay(200);
     digitalWrite(doPin, LOW);
  }
  Serial.print("input value = ");
  Serial.println(diValue);
  prevValue = diValue;
}
```

Figure 7.7: Polling example sketch.

want samples at regular time intervals). In these cases, we can use delta timing techniques to both: (a) regularize the time period of each iteration, and (b) implement the 200 ms pulse for the digital output signal.

Figure 7.8 illustrates the use of delta timing to accomplish the goals stated above. In this sketch, five iterations happen each second (200 ms per iteration), and the digital input is read precisely once per iteration. When the output signal goes HIGH, due to the low-to-high transition on the input, it is sent LOW one iteration later.

But what happens in the sketch of Figure 7.8 if the digital input signal goes high for 100 ms and then goes back low? Since the digital input is being polled every 200 ms, there is a significant chance that the high pulse that is present on the input pin will be completely missed by the software.

```
const long deltaTime = 200;      // loop period (in ms)
long loopEndTime = deltaTime;
long currentTime = 0;
const int diPin = 16;            // digital input pin is 16
const int doPin = 17;            // digital output pin is 17
int diValue = LOW;               // digital input value
int prevValue = LOW;             // previous input value

void setup() {
  pinMode(diPin, INPUT);         // set pin to digital input
  pinMode(doPin, OUTPUT);        // set pin to digital output
  digitalWrite(doPin, LOW);
  Serial.begin(9600);
}

void loop() {
  currentTime = millis();
  if (currentTime >= loopEndTime) { // one period is complete
     loopEndTime += deltaTime;
     diValue = digitalRead(diPin);  // read the input
     digitalWrite(doPin, LOW);
     if (prevValue == LOW && diValue == HIGH) {
        digitalWrite(doPin, HIGH);
     }
     Serial.print("input value = ");
     Serial.println(diValue);
     prevValue = diValue;
  }
}
```

Figure 7.8: Polling sketch with regular period.

One solution to this issue is to poll the input at a higher rate. If we have knowledge that the shortest pulse that will be present at the input is, say, 100 ms, a period shorter than this minimum pulse duration will always see any low-to-high transition.

Another solution is to use a hardware feature present in microcontrollers that allows us to alter the program flow of control under certain specified

circumstances, including the change in value of a digital input pin. This hardware feature is called an *interrupt* and will be our next subject.

7.2.2 Interrupts

An *interrupt* is a change in program control flow based upon an explicit trigger event. Interrupts can be triggered by a wide variety of things, including both internal events (e.g., a timer) and external events (e.g., a digital input pin). Here, we will consider using interrupts to detect the low-to-high transition on an input pin.

There are a number of things that must be present to accomplish this.

1. A trigger must be unambiguously specified for the interrupt to occur (e.g., a level change on a particular digital input pin).

2. A routine must be authored that is the target of the control flow change (i.e., the code that gets called when the trigger occurs). This routine is commonly called an *interrupt service routine* (ISR), since it "services" the interrupt when it occurs.

3. Because the ISR cannot have parameters or a return value, mechanisms must be put in place for interaction between the main sketch and the ISR. This is typically accomplished through the use of global variables.

4. Care must be taken when accessing these global variables, because interrupt triggers can happen at any time.

We specify an interrupt trigger using the `attachInterrupt()` library routine. The parameters are an interrupt number (provided by a call to the library function `digitalPinToInterrupt()`), the name of the ISR, and one of the modes listed in Table 7.1.

Table 7.1: Interrupt trigger modes.

Mode	Meaning
LOW	trigger interrupt whenever pin is low
CHANGE	trigger interrupt whenever pin changes value
RISING	trigger interrupt whenever pin goes from low to high
FALLING	trigger interrupt whenever pin goes from high to low

When authoring the interrupt service routine, there are a number of limitations that must be followed. As already stated, the ISR cannot have parameters and does not return a function value. In addition, a number of the

timing library routines will not operate as you otherwise might expect within an ISR, because they use interrupts themselves. This includes `delay()` and `millis()`. Finally, since only one ISR can run at a time, and interrupts can be triggered at any time, an ISR should be as short as possible.

When declaring global variables to use with an ISR, include the `volatile` keyword. When accessing (reading or writing) those global variables, make sure that interrupts are turned off (using `noInterrupts()` and `interrupts()`). The code that is executed while interrupts are off is called a *critical section*, implying that it is critical that an interrupt not occur during the execution of this code. As with an ISR, a critical section should also be as short as possible.

If the trigger for an interrupt occurs while the interrupts are turned off (i.e., while the sketch is in a critical section), the interrupt is *pending* and will trigger once interrupts have been turned back on.

Figure 7.9 illustrates a sketch that uses interrupts to detect when a digital input pin goes from low to high. The ISR is called `catchRisingEdge` and is shown at the bottom of the sketch. Note that it is very short, taking as little time to execute as is reasonable. The interrupt trigger is on the rising edge of `diPin`, as specified in the invocation of `attachInterrupts()`. Note that only a limited number of pins support interrupts. The variable `flag` is used to communicate between the ISR and the main `loop()` and it is declared `volatile`. When `flag` is accessed in `loop()` it is in a critical section (which is also very short).

Because `catchRisingEdge()` gets invoked by the microcontroller's hardware interrupt mechanisms, it can happen any time that interrupts are on (the sketch is not in a critical section). This means that even a very short pulse on the digital input pin will cause the interrupt to trigger, the ISR to be called, `flag` to be set `true`, and the rising edge not to be missed.

7.2.3   Discussion

While the descriptions of both polling and interrupts used the example of reading a digital input pin, both design patterns are frequently used for other tasks as well (e.g., regular timing for digital outputs, analog inputs and outputs, etc.). As a general rule, polling is a simpler approach and is therefore less prone to errors in implementation, while interrupts can accomplish some things a time scales that are not readily accessible to polling.

```
const long deltaTime = 200;    // loop period (in ms)
long loopEndTime = deltaTime;
long currentTime = 0;
const int diPin = 2;           // pin 2 supports interrupts
const int doPin = 17;          // digital output pin is 17
volatile boolean flag = false; // input rising flag

void setup() {
  pinMode(diPin, INPUT);       // set pin to digital input
  pinMode(doPin, OUTPUT);      // set pin to digital output
  digitalWrite(doPin, LOW);
  attachInterrupt(digitalPinToInterrupt(diPin),
                  catchRisingEdge, RISING);
}

void loop() {
  currentTime = millis();
  if (currentTime >= loopEndTime) { // one period is complete
    loopEndTime += deltaTime;
    digitalWrite(doPin, LOW);
    noInterrupts();                 // enter critical section
    if (flag) {
      digitalWrite(doPin, HIGH);
      flag = false;
    }
    interrupts();                   // exit critical section
  }
}

void catchRisingEdge() {
  flag = true;
}
```

Figure 7.9: Reading input using interrupts.

7.3   Event-driven Programming

The last design pattern than we will discuss in this chapter is *event-driven programming*. In an event-driven program, the organization is centered around *events* that the program responds to as they happen, as opposed to the program itself controlling the sequence and/or timing of when things happen.

Events can be almost anything that the program is designed to react to. Examples include user actions (keypress or mouse click), elapsed time, external inputs (digital input, receipt of information on a communications link), etc. The event detection can be implemented using polling (Figure 7.7 can be considered an example of an event-driven program where the event is the digital input going from low to high) or using interrupts. The code that gets executed when the event happens is frequently called an *event handler*.

We've already seen several examples of event-driven programming. In Chapter 6, the delta time approach to timing is a case where elapsed time can be considered to be an event. Figure 7.10, which is the same as Figure 6.7, just repeated here, illustrates two different time periods, where the code within the `if` conditional for each delta time test will each be executed when their respective timer has elapsed (either every 200 ms or 500 ms, respectively).

Both of the example sketches that implement finite-state machines above are also illustrations of the design pattern. In Figure 7.2, the routine that accepts a new coin, `inputNextCoin()`, implements the receipt of a coin which the rest of the sketch treats as an event, and the receipt of a character is treated as an event in Figure 7.5.

The use of the event-driven programming design pattern provides some very explicit benefits; however, it also entails some additional responsibilities on the part of the programmer. We will discuss each in turn.

7.3.1   Benefits of Event-driven Programming

The primary benefit of event-driven programming techniques is that they allow the executing program to react to when events happen, and in what order events happen, in a natural way. In the FSM examples in Section 7.1 above, the logic of the sketch did not need to be altered to account for when the input symbols arrived. Instead, the receipt of an input symbol is an event, and the processing of that event happens whenever it is received.

In the delta time example of Figure 7.10, in which there are two different timers that are being used, the sketch does not need to be concerned about which timer will expire next. The reaction to each event (in this case, timers expiring) is implemented separately from one another.

```
const long deltaTimeA = 200;  // loop period A (in ms)
const long deltaTimeB = 500;  // loop period B (in ms)
long loopEndTimeA = deltaTimeA;
long loopEndTimeB = deltaTimeB;
long currentTime = 0;

void setup() {
}

void loop() {
  currentTime = millis();
  if (currentTime >= loopEndTimeA) { // period A is complete
    loopEndTimeA += deltaTimeA;
    // code to be executed once per 200 ms
  }
  if (currentTime >= loopEndTimeB) { // period B is complete
    loopEndTimeB += deltaTimeB;
    // code to be executed once per 500 ms
  }
}
```

Figure 7.10: Two timing loops using delta time techniques.

A subtle benefit that follows from the property that event-driven designs react to events whenever they happen is that the sketch can do other things while waiting for time to elapse. Therefore, if a sketch decides that something is to happen in the future, rather than just inserting a `delay()` call, the programmer can invoke delta timing techniques and proceed to do other things while waiting for the desired time to elapse.

Practice Problem   Outline how one would author a sketch that combines both a delta time based timer and an FSM in the same application. You may assume that there are routines available that will test whether or not an input symbol has arrived as well as receive the next input symbol.

Solution   The sketch below gives the basic structure.

```
loop{} {
    currentTime = millis();
    if (currentTime >= endTime) { // time period is complete
        endTime += deltaTime;
        // event handler for timer-based event
    }
    if (symbolPresent()) { // true if FSM input symbol available
        inputSymbol = inputNextSymbol();
        // event handler that implements FSM
    }
}
```

Notice that the delta time event handler and the FSM event handler will execute in whatever order is dictated by time moving forward and input symbols being received.

7.3.2   Challenges with Event-driven Programming

While the benefits of event-driven programming design are substantial, they are not without challenges as well. Frankly, one of the greatest benefits, that event handlers can be invoked in any order based on the receipt of the events themselves, is also a challenge. The code that goes into an event handler cannot depend on event handlers being triggered in any specific order.

Take, for example, the sketch of Figure 7.10, which has an event handler for the 200 ms period events and a separate event handler for the 500 ms period events. Even though the 200 ms event handler is the first one in `loop()`, one cannot assume that it is executed first. If `loopEndTimeA` is larger than `loopEndTimeB`, the 500 ms event handler will be the first one to execute.

What that means for the programmer is that whenever one is reasoning about the logic of the sketch to ensure everything will work correctly, one has to consider all possible orderings of the event handlers. Of course, the easiest circumstances to do this under are when the event handlers are independent of one another. If they are not independent, make sure to consider all orderings when reasoning about your code.

Another challenge associated with event handlers is that issues can arise whenever too much time is spent in them. If other events happen during the

time that the event handler code is executing, the invocation of the event handlers for those events is delayed.

This can be especially egregious when the programmer inserts a `delay()` call into an event handler. The `delay()` routine does not return program control until the specified time has elapsed, which means that nothing else can happen in the code of the sketch until that `delay()` completes.

If you are tempted to author code in an event handler that is going to take some significant amount of elapsed time, it is appropriate to reconsider the way in which the code is organized. For example, take advantage of the ability to simply schedule some other event later in time. To do this, do the following three things: (1) save the pertinent information for your program logic to continue (e.g., if you are iterating through a list of items, save which item you are currently processing); (2) set a time-based event for the later time using delta time techniques; and (3) continue the logic of the program in the delta time event handler.

Figure 7.11 illustrates this transformation. The first sketch, Figure 7.11(a) is organized around a `for` loop, with a call to `delay()` between each index `i`'s processing. What that implies is that contained within a single entry into `loop()`, the sketch will process the entire list or array (all `N` entries), and nothing else will get the microprocessor's attention. No reading analog or digital inputs, no changing of analog or digital outputs, not receipt of input symbols for a finite-state machine, nothing else will get done.

Now compare that with the event-based design of the second sketch, Figure 7.11(b). Here, the pertinent information required for the sketch to remember where it is has been declared as the index variable `i`, which is initialized to 0 at compile time. The processing of each index is now in the event handler triggered by the delta time `if` test. Within that event handler, the list or array element `i` is processed, the index variable is updated (i.e., `i++`), and the next time for the event to trigger is set.

Consider how this event-based sketch operates. First, and most importantly, during the time that a list or array item is not being processed, the microcontroller is free to do other things. Additional event handlers can be added to `loop()`, and they can trigger between the execution of two different values of the index `i`. Second, the use of delta timing techniques gives more confidence that the actual execution of the items in the list or array is happening every 0.5 s (rather than 0.5 s plus whatever time is spent by the item processing and `for` loop overheads). This latter benefit accrues simply from the use of delta timing techniques, which are inherently event-driven.

78

```
const long deltaTime = 500;  // period (in ms)

void setup() {
}

void loop() {
  for (int i=0; i<N; i++) {
    // process item i in a list or array
    delay(deltaTime);
  }
}
```

(a) Loop-based sketch.

```
const long deltaTime = 500;  // period (in ms)
long endTime = 0;
long currentTime = 0;
int i=0;                       // index into list or array

void setup() {
}

void loop() {
  currentTime = millis();
  if (currentTime >= endTime && i < N) { // time is elapsed
    // process item i in a list or array
    i++;
    endTime += deltaTime;
  }
}
```

(b) Event-based sketch.

Figure 7.11: Transforming loop-based sketch (a) into event-based sketch (b).

Practice Problem   Although the loop-based and event-based sketches of Figure 7.11 are discussed as if they perform the same processing on each index, that is actually not the case. We intentionally left out an important fact that distinguishes the resulting operation of the two sketches. What is that fact?

Solution   The fact that distinguishes the two sketches is that they do not operate in the same way after the processing of all `N` items in the list or array. The loop-based sketch processes all `N` items in one invocation of `loop()`, then re-enters `loop()` and starts processing all of the items again. The event-based sketch processes all `N` items over multiple invocations of `loop()`, and once `i` is equal to `N`, the event no longer triggers due to the `i < N` conditional as part of the `if` test that triggers the event.

   If one wanted to convert the event-based sketch to act the same as the loop-based sketch, one simple way to do that would be to change the update to `i` in the event handler from `i++` to `i = (i + 1) % N`.

8 Information Representation

This chapter will deal with how information is represented within a computer. We will start with numbers, followed by characters and strings, and finish up with how we represent images.

8.1 Numbers

Numbers come in lots of forms. We can talk about the counting numbers, integers, reals, or complex numbers. Algebra allows us to represent relationships between numbers, and reason about those relationships. Here, we are interested in the approaches used to represent numbers with a computer. This includes how to store numbers as well as manipulate them mathematically.

8.1.1 Brief History of Number Systems

We will start with a brief history of number systems. Did you ever wonder how the Romans wrote down the number zero? Think about it, I is one, II is two, III is three, IV is four, etc. But how did they write down zero?

In what follows, we will constrain ourselves to using standard positional notation (i.e., the numerical value of a symbol depends upon its position).

Counting Numbers

One of the earliest uses of written numbers is to count things. This is known to have occurred by the late fourth millennium B.C. in Mesopotamia, present day Iraq [8], although it might have happened even earlier than that.

In the decimal system that we humans typically use these days[1], the first several *counting numbers* are:

    1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 ...

---

[1]This was not always the case, the Babylonians used base 60, which is how we ended up with 60 seconds in a minute and 60 minutes in an hour.

which starts at 1 but continues infinitely far to the right.

The counting numbers are good at measuring how much stuff I have. For example, with just the counting numbers I can compare with my neighbor, and say, "I have 7 cows and 2 goats. How many do you have?" And my neighbor can reply in a meaningful way.

Getting just a bit more formal, we say that the counting numbers are *closed* under addition and multiplication. What this means is that if we know that a is a some counting number and we know that b is also some counting number, without knowing the particular values of a or b we do know that $a + b$ is a counting number and $a \times b$ is a counting number. To say that a number system (such as the counting numbers) is closed over some operation is to say that performing the operation on numbers within the system results in a number that is also within the number system. In other words, adding a pair of counting numbers cannot give you a negative number or a fraction, the result will always be another counting number.

Zero: Natural Numbers

If I've just told my neighbor that I have 7 cows and 2 goats, and I've also politely inquired about how many he has, how does he respond if he has no cows? He can certainly say something like, "I do not have any cows. However, I do have 4 goats." This reply, of course, answers the question, but doesn't help us reason about "no" cows in an algebraic system.

Brahmagupta, an Indian mathematician, addressed the above issue by describing the rules governing the use of the number 0, or zero, in A.D. 628 [2]. By incorporating 0 into our number system, we now have the *natural numbers*, which include all the counting numbers and zero as well. This number system is also known as the *whole numbers*. Like the counting numbers, the natural numbers are closed under addition and multiplication.

Returning to our question earlier, how did the Romans write down zero? They didn't, literally, as a number. Instead, they used the word *nulla* meaning "nothing." They might write *Ego non habent ullam vaccas*, which when translated from Latin means, "I do not have any cows."

Negative Numbers: Integers

If I can count things, you can guess that commerce isn't far behind. In exchange for some thing I value, I might give my neighbor 3 chickens. I can reason about this just fine if I have 3 or more chickens to give, but what about when I promise my neighbor 3 chickens but the chicks haven't hatched

yet (i.e., I don't have 3 chickens to give)? I can, of course, say, "I owe you 3 chickens." Here, I am using words that are outside the number system to differentiate two distinct meanings for the phrase "3 chickens." "I own 3 chickens" means something very different than "I owe 3 chickens."

However, maybe there is a better way. Let's introduce the concept of negative numbers, which gives us the number system called the *integers*. With negative numbers, rather than using different words to talk about 3 chickens, I can use one number system to represent both concepts. "I owe 3 chickens" gets transformed into "I own -3 chickens."

Getting back to formalism, the integers are closed under addition, multiplication, and subtraction. Clearly, the notion of giving my neighbor some of my chickens (either present or future chickens) can be represented using subtraction. Another way we make formal statements about number systems is to describe forms of algebraic equations that can be solved within the number system. For example, if a is a constant integer, we can solve equations of the form

$$x + a = 0 \tag{8.1}$$

and know that the value of x that solves the equation will be an integer.

A quick note on the names of different numbers. With the advent of negative numbers, the numbers that are not negative came to be called positive[2] Humanity has started down a path in which many number systems are named as opposites. That is, the name of the number system borrows opposite labels from the natural language words used to name them.

Rational Numbers

As you can well imagine, the development of number systems was strongly driven by the needs of commerce. People need to know how much of this or that they own, buy, sell, or trade. They also die, and their children inherit.

If I lived in antiquity and owned 2 pigs, when I died tradition held that my 2 pigs were divided among my 3 sons. (Sorry gals, enlightened thinking about equality of the sexes came *much* later than the notion of rational numbers.) Each of my sons now owns 2/3 of a pig, and we have expanded our number system to explicitly include *ratios* between integers. This defines the *rational numbers*. Note that the label "rational" comes from the root "ratio," not the other English meanings associated with the word rational, such as reasonable or logical.

---

[2]Although it is the matter of some debate whether or not 0 is included in the positive numbers, we'll ignore this bit of minutia.

More formally, rational numbers are closed under addition, multiplication, subtraction, and division. If a and b are constant rational numbers, we can solve equations of the form

$$ax + b = 0 \tag{8.2}$$

as long as $a \neq 0$.

Irrational Numbers: Reals

Once folks figured out the rational numbers, they thought they had it all down. Other than this weird issue of not being able to divide by zero, they could do pretty much everything they thought they wanted to. Addition, subtraction, multiplication, and division were all available to them, and the number system handled it all.

Except....

Figure 8.1 was puzzling. Given a right triangle (the angle at the bottom left is precisely $90°$), with adjacent edges each of length 1, how long is the opposite edge, or the hypotenuse? If a is the length of the bottom edge ($a = 1$ in this case), b is the length of the left-most edge ($b = 1$ in this case), and x is the unknown length of the hypotenuse, the Pythagorean theorem tells us

$$a^2 + b^2 = x^2 \tag{8.3}$$

and if we substitute the known values for a and b (and do a little algebraic manipulation), we get

$$x^2 - 2 = 0 \tag{8.4}$$

for which there are two solutions: $x = \sqrt{2}$ and $x = -\sqrt{2}$.
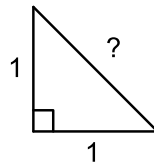
Figure 8.1: Right triangle. How long is the hypotenuse (the edge opposite the right angle)?

The problem is that for centuries mathematicians couldn't find rational solutions to Equation (8.4), because neither solution can be expressed as a ratio of two integers. In other words, the rational numbers are not closed under the square root operation.

The *real numbers* expand the number system beyond the rational numbers to include values that cannot be expressed as a ratio of two integers. Examples include $\sqrt{2}$ (illustrated above), π (the ratio of the circumference of a circle to its diameter), and e (the base of the natural logarithms).

Real numbers that are not rational numbers are called *irrational numbers*. You might notice the pattern mentioned above continuing. In the naming of number systems, numbers that are not rational are called irrational. However, remember that rational came from ratio, not the other possible meanings of rational in English (e.g., logical, reasonable). As a result, irrational means "cannot be expressed as a ratio," not "illogical" or "unreasonable."

Complex Numbers

While they can do quite a bit, real numbers aren't yet the be all and end all of number systems. Consider the following equation:

$$x^2 + 2 = 0 \tag{8.5}$$

There does not exist a real number that will solve it. Instead, we will introduce *complex numbers*.

Consider a vector number system with 2 components: (a,b) where a and b are both real numbers. Our new vector number system obeys the following rules.

1. **Equality:** $(a, b) = (c, d)$ iff[3] $a = c$ and $b = d$.

2. **Addition:** $(a, b) + (c, d) = (a + c, b + d)$.

3. **Multiplication:** $(a, b) \times (c, d) = (ac - bd, ad + bc)$.

Note: numbers in this number system with the second components equal to 0 have the same properties as real numbers:

1. $(a, 0) = (c, 0)$ iff $a = c$.

2. $(a, 0) + (c, 0) = (a + c, 0)$.

3. $(a, 0) \times (c, 0) = (ac, 0)$.

Continuing the naming pattern established earlier, if the first component of the vector number system is called real, it was only a matter of time before the second component came to be called *imaginary*. This name is somewhat

---

[3]The notation iff is shorthand for *if and only if*.

unfortunate, however, as many people then associate the English definition of imaginary (i.e., made up, fake) with the second component of the vector number system, and no such association is warranted. The use of the label "imaginary" is nothing other than a historical accident.

We now return to Equation (8.5). First we rewrite it as an equation in complex numbers (our two component vector number system).

$$x^2 + (2,0) = (0,0) \tag{8.6}$$

Second, we assign $x = (0, \sqrt{2})$.

$$
\begin{aligned}
x^2 + (2,0) &= (0, \sqrt{2})^2 + (2,0) \\
&= (0, \sqrt{2}) \times (0, \sqrt{2}) + (2,0) \\
&= (-2,0) + (2,0) \\
&= (0,0) \quad \checkmark
\end{aligned}
$$

This shows that $x = (0, \sqrt{2})$ is a solution to Equation (8.5).

Another interesting equation is shown below.

$$x^2 + 1 = 0 \tag{8.7}$$

A little bit of algebraic manipulation yields the following,

$$
\begin{aligned}
x^2 + 1 &= 0 \\
x^2 &= -1 \\
x &= \sqrt{-1}
\end{aligned}
$$

which is a number that has intrigued folks for years. Let's now try out Equation (8.7) with $x = (0,1)$:

$$
\begin{aligned}
x^2 + (1,0) &= (0,1)^2 + (1,0) \\
&= (0,1) \times (0,1) + (1,0) \\
&= (-1,0) + (1,0) \\
&= (0,0) \quad \checkmark
\end{aligned}
$$

which tells us that $x = (0,1) = \sqrt{-1}$.

So far we have presented the complex numbers as a two component vector number system. A far more common notation for complex numbers defines the symbol $i = \sqrt{-1}$. With this definition of i, then any complex number

written in the form (a, b) can be rewritten as $a + ib$, which can be understood by the following line of reasoning.

$$a + ib = (a, 0) + (0, 1) \times (b, 0)$$
$$= (a, 0) + (0, b)$$
$$= (a, b)$$

This gives us the traditional form of writing complex numbers.

So, formally, how powerful are complex numbers. It turns out that complex number are rich enough as a number system to solve arbitrary constant coefficient polynomial equations of the form:

$$a_0 x^n + a_1 x^{n-1} + ... + a_{n-1} x + a_n = 0 \tag{8.8}$$

If the a's are complex-valued, $n \geq 1$, and $a_0 \neq 0$, there are precisely n roots to the equation [4]. This result is known as the *Fundamental Theorem of Algebra*, and you know they don't give a theorem that important a name unless it's pretty important stuff.

8.1.2 Positional Number Systems

It is traditional in the modern world to write numbers using a positional system, in which the value of a numerical digit (or digit symbol) depends upon its position within the number as a whole. This is true not only for the decimal system that we most commonly use as humans, but also for the binary system that gets used within digital computers.

Decimal

In the decimal system, which is base 10, the weight associated with each position is a power of 10. If we have a 3-digit number denoted as uvw_{10}, where u is the 1st digit, v is the 2nd digit, w is the 3rd digit (i.e., $0 \leq u, v, w \leq 9$), and the the subscript 10 indicates the number is written in decimal notation, then the overall value is represented by

$$uvw_{10} = u \cdot 10^2 + v \cdot 10^1 + w \cdot 10^0$$
$$= u \cdot 100 + v \cdot 10 + w.$$

Positional notation extends to the right side of the decimal point as well. If we have the 6-digit number $uvw.xyz_{10}$, again each letter is one decimal digit (i.e., $0 \leq u, v, w, x, y, z \leq 9$), the the overall value is represented by

$$uvw.xyz_{10} = u \cdot 10^2 + v \cdot 10^1 + w \cdot 10^0 + x \cdot 10^{-1} + y \cdot 10^{-2} + z \cdot 10^{-3}$$
$$= u \cdot 100 + v \cdot 10 + w + x \cdot 0.1 + y \cdot 0.01 + z \cdot 0.001.$$

Binary

The rules for positional numbers in the binary system are the same as for the decimal system, with only two differences. Instead of digits having values between 0 and 9, in the binary system digits can only have two values, 0 or 1; and the weight associated with each position is a power of 2.

In binary, if we have a 3-digit number denoted as uvw_2, where u is the 1st digit, v is the 2nd digit, w is the 3rd digit ($0 \leq u, v, w \leq 1$), and the subscript 2 indicates the number is written in binary notation, the overall value is

$$uvw_2 = u \cdot 2^2 + v \cdot 2^1 + w \cdot 2^0$$
$$= u \cdot 4 + v \cdot 2 + w.$$

Note, the above expression uses decimal notation, a practice we will continue unless explicitly noted otherwise.

For example if the binary number is 100_2, $u = 1$ (the first digit), $v = 0$ (the second digit), and $w = 0$ (the third digit). The decimal value is therefore:

$$100_2 = (1 \cdot 4) + (0 \cdot 2) + 0$$
$$= 4 + 0 + 0$$
$$= 4_{10}$$

Similarly, the decimal value of 101_2 is 5, the decimal value of 010_2 is 2, and the decimal value of 000_2 is 0.

Binary numbers need not be limited to 3 digits. As the number of digits increases (to the left), the weight of each digit increases by a factor of 2. I.e., to the left of 4 is 8, then 16, 32, etc. As a result, the decimal value of the binary number 10001_2 is $16 + 1 = 17$.

Fractional numbers work as well in binary as in decimal. If we have the 6-digit number $uvw.xyz_2$, again each letter is one binary digit (called a bit), the overall value is

$$uvw.xyz_2 = u \cdot 2^2 + v \cdot 2^1 + w \cdot 2^0 + x \cdot 2^{-1} + y \cdot 2^{-2} + z \cdot 2^{-3}$$
$$= u \cdot 4 + v \cdot 2 + w + x \cdot \tfrac{1}{2} + y \cdot \tfrac{1}{4} + z \cdot \tfrac{1}{8}.$$

We normally call the "." that separates the integer portion of the number from the fractional portion of the number the *decimal point*; however, we are no longer using the decimal number system, so that terminology is technically incorrect. The generalized word for the "." symbol is the *radix point*, which is a term that is appropriate to use whatever base we are using (the *radix* is simply another word for the base of a number system).

Let's look at a couple more examples, this time for binary numbers that are not limited to integers. If the binary number is 011.100_2, then $u = 0$, $v = 1$, $w = 1$, $x = 1$, $y = 0$, and $z = 0$. The decimal value is therefore:

$$011.100_2 = (0 \cdot 4) + (1 \cdot 2) + 1 + (1 \cdot \tfrac{1}{2}) + (0 \cdot \tfrac{1}{4}) + (0 \cdot \tfrac{1}{8})$$
$$= 0 + 2 + 1 + 0.5 + 0 + 0$$
$$= 3.5_{10}$$

In the same way, the decimal value of 111.001 is 7.125. As before, the fraction need not be limited to 3 digits. Moving to the right, the weight of the next digit is 1/16, then 1/32, etc.

While numerical input to a computer and output from a computer might be provided by the user and presented to the user in decimal representation, rest assured that the internal computations are all being performed in binary. Techniques for converting numbers between bases are provided in Appendix C.

Hexadecimal

While numerical representation within the computer is all in binary, this representation is quite cumbersome for humans. It is very difficult for us visually to distinguish between, e.g., 01101110 and 01100110, and as a result, whenever humans are required to deal directly with binary representation, it is a very error-prone endeavor.

Fortunately, there are options that can help us deal with this issue, in a way that make working with binary values dramatically more convenient. The option that is most frequently used is to convert the binary values that we wish to reason about into *hexadecimal* notation, i.e., base 16. Note, it is common practice to shorten the label hexadecimal to just *hex* (which we will frequently do as well). That does not change the fact that the base is 16, not 6!

First, let's examine hexadecimal (or hex) notation itself, and then we'll consider why it is so helpful in terms of humans reasoning about binary. As in all positional systems, the value of a digit depends upon its position. In this case, the base is 16, so the weight associated with each position is a power of 16. In hexadecimal notation, each digit can have values ranging from 0 to 15, and it is conventional to use the first six letters of the alphabet to represent the values 10 through 15 when denoting numbers in hex. So, don't think of the letters `a` through `f` as variables in algebraic notation, but instead think of

them as numerical digits[4]. Table 8.1 gives the value (in decimal) of each of the digits we will use in hexadecimal notation.

Table 8.1: Value (in decimal) of hexadecimal digits.

Hex digit	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
Value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Using these digits, if we have a 3-digit number denoted as uvw_{16}, where u is the 1st digit, v is the 2nd digit, w is the 3rd digit ($0 \leq u, v, w \leq 15$), and the subscript 16 indicates the number is written in hex notation, the overall value is

$$uvw_{16} = u \cdot 16^2 + v \cdot 16^1 + w \cdot 16^0$$
$$= u \cdot 256 + v \cdot 16 + w.$$

For example, if the hexadecimal number is $13a_{16}$, the decimal value is

$$13b_{16} = (1 \cdot 256) + (3 \cdot 16) + 11$$
$$= 256 + 48 + 11$$
$$= 315.$$

Fractional numbers work as well in hex as in binary or decimal. If we have the 6-digit number $uvw.xyz_{16}$, again each letter is one hex digit, the overall value is

$$uvw.xyz_{16} = u \cdot 16^2 + v \cdot 16^1 + w \cdot 16^0 + x \cdot 16^{-1} + y \cdot 16^{-2} + z \cdot 16^{-3}$$
$$= u \cdot 256 + v \cdot 16 + w + x \cdot \tfrac{1}{16} + y \cdot \tfrac{1}{256} + z \cdot \tfrac{1}{4096}.$$

For example, the hex number $1f.4_{16}$ has the decimal value

$$1f.4 = (1 \cdot 16) + 15 + (4 \cdot \tfrac{1}{16})$$
$$= 16 + 15 + 0.25$$
$$= 31.25.$$

There are two very strong reasons why we use hex extensively instead of binary. First, hex is visually much closer to the familiar decimal representation, so we make fewer human errors when reading and the number of digits is closer to what our brains expect to see and understand. Second, it is very

---

[4]We will use `a` through `f` (lower case), but it is also common to use `A` through `F` (capitals) to represent the values 10 through 15.

straightforward to convert back and forth between binary and hex representations. As a result, it is quite common to use hex as a shorthand for binary, to simplify our ability to copy, compare, etc., numbers, but recall that inside the machine it really is binary all the time. Hex is nothing more than a convenience for us as humans.

To convert from hex to binary, we start at the radix point and translate each hex digit into 4 binary digits, moving both to the left and right of the radix point. This gives

$$1fc7.2d4_{16} =$$
$$0001\ 1111\ 1100\ 0111.0010\ 1101\ 0100_2$$

where space has been added between groups of 4 binary digits to help see the correspondence between each hex digit and each group of 4 binary digits. It is traditional to assume the radix point is to the far right of a number if it is not explicitly shown (i.e., the number is a whole number). A second example is

$$86eb01_{16} =$$
$$1000\ 0110\ 1110\ 1011\ 0000\ 0001_2$$

Converting from binary to hex simply reverses the process. Group the binary digits into groups of 4, starting from the radix point and moving out to the left and the right. If the number of binary digits on either side of the radix point is not an even multiple of 4, pad the binary number with zeros until it is an even multiple of 4. Then convert each group of 4 binary digits into the equivalent hex digit.

Both Java and C support the specification of hexadecimal constants by prepending the number with the symbols `0x`, so that the number $1f3_{16}$ would be written `0x1f3`. We will use this notation going forward to indicate that a number is understood to be in hexadecimal form.

8.1.3   Supporting Negative Numbers

When writing numbers down on a page, there is a straightforward notation that we are all used to when we wish to denote that a number is negative, the "−" symbol, or the minus sign. This technique doesn't work, however, within digital systems that can only use 0 and 1 as symbols. In the sections below, we describe a number of techniques that are currently used in computer systems for representing negative numbers.

Table 8.2: Sign-magnitude integers.

Binary number	Decimal value
00	$+0$
10	-0
01	$+1$
11	-1
0101	$+5$
1101	-5
01000000	$+64$
11000000	-64

Sign-Magnitude

The first technique for representing negative numbers within a computer draws its origins straight from the written notation that we are all familiar with. The first digit (or bit) of a number is simply designated as a sign bit, and the bits that follow represent the value (magnitude) of the number. The normal convention is to have 1 represent a negative number and 0 represent a positive number (this is because positive numbers then are interpreted the same way as regular unsigned binary values).

Table 8.2 shows the value (in decimal) for several integers represented in what is called *sign-magnitude* form. The first two entries illustrate the biggest issue with the sign-magnitude representation, there are two ways to designate 0, since $+0$ and -0 are really the same thing. With two different representations, however, the circuitry and other logic necessary to manipulate numbers (e.g., check for equality) become more complicated.

When a fixed number of bits are used to store a sign-magnitude number, we can easily discern the range of values that are supported. With n bits in the number, the range of possible values that can be represented is $-(2^{n-1}-1)$ to $+(2^{n-1}-1)$.

Excess or Offset

The second technique is called either *excess* notation or *offset* notation. In this approach, a fixed amount (that must be agreed to ahead of time) is logically subtracted from each number. For example, if the regular binary value of a number was 3, and the offset was specified as 7, the value of the number is $3 - 7 = -4$.

Table 8.3: 4-bit numbers and their value in several number systems.

4-bit number	Binary value	Sign-magnitude value	Excess-7 value	Two's complement value
0000	0	0	−7	0
0001	1	1	−6	1
0010	2	2	−5	2
0011	3	3	−4	3
0100	4	4	−3	4
0101	5	5	−2	5
0110	6	6	−1	6
0111	7	7	0	7
1000	8	−0	1	−8
1001	9	−1	2	−7
1010	10	−2	3	−6
1011	11	−3	4	−5
1100	12	−4	5	−4
1101	13	−5	6	−3
1110	14	−6	7	−2
1111	15	−7	8	−1

When a fixed number of bits are used to store a number in excess notation, the offset amount is typically chosen to be near the midpoint of the range of representable values. For example, Table 8.3 shows both the unsigned (binary) values and the excess notation values for a 4-bit excess number system with an offset of 7 (along with sign-magnitude and two's complement values).

While this notation has the nice property that value comparisons work the same way as regular binary notation (if $a < b$ using normal binary conventions, $a < b$ in excess notation as well); however, arithmetic manipulation (addition, subtraction) is substantially complicated.

Radix Complement

While the sign-magnitude and excess notation do get used in computer systems (see the description of floating-point numbers below), by far the most common approach to representing negative numbers in computers is known as *two's complement* notation, or more generally *radix complement* notation (with two as the binary radix).

Two's complement notation is constrained to number systems with a fixed number of bits. Like regular binary numbers, the two's complement number

system is a positional system, with weights associated with each position. What is unique about two's complement numbers is that the left-most digit (the first bit of the number) has a weight that is negative.

In two's complement, if we have a 4-digit number denoted as $uvwx$, where u is the first digit, v is the second digit, w is the third digit, and x is the fourth digit, the value is

$$uvwx = u \cdot -8 + v \cdot 4 + w \cdot 2 + x$$

where the weight associated with the first digit has the same magnitude that it would have in regular unsigned binary notation, but its weight is negative.

For example, if the two's complement number is 0100, the decimal value is

$$
\begin{aligned}
0100 &= (0 \cdot -8) + (1 \cdot 4) + (0 \cdot 2) + 0 \\
&= 0 + 4 + 0 + 0 \\
&= 4
\end{aligned}
$$

which is the same as in regular binary. As a second example, if the two's complement number is 1101, the decimal value is

$$
\begin{aligned}
1101 &= (1 \cdot -8) + (1 \cdot 4) + (0 \cdot 2) + 1 \\
&= -8 + 4 + 0 + 1 \\
&= -3
\end{aligned}
$$

which is negative.

The two's complement number system has several properties that make it attractive for use in computer systems:

1. The least significant $n-1$ bits (of an n-bit number) have the same meaning in two's complement notation as in the standard binary positional notation.

2. The weight of the most significant bit is negated; however, it retains the same magnitude as its weight in the standard binary positional notation.

3. There is only one zero (and every bit of zero is 0).

4. All negative numbers have a 1 in the first bit, and all non-negative number (0 and positive numbers) have a 0 in the first bit. As a result, this bit is commonly called the *sign bit*.

5. Arithmetic circuits that perform addition work equally well for standard binary notation and two's complement notation.

For an n-bit two's complement number system, the range of values that can be represented is $-(2^{n-1})$ to $+(2^{n-1} - 1)$. Virtually all integer numbers in computers are represented using two's complement representation.

8.1.4 Integer Data Types in Programming Languages

When numbers are represented inside a digital computer, they are stored in fixed-size memory locations and manipulated using arithmetic circuits that support a fixed number of bit positions. As indicated by the above discussion, the range of integer values that can be represented in a fixed number of bits depends on the number of bits. Assuming a two's complement representation, the range of integer values that can be represented in n bits is between -2^{n-1} and $2^{n-1} - 1$. By convention, the least significant bit is designated as bit 0 and the most significant bit is designated as bit $n - 1$, such that a 16-bit number with binary digits b_i would be as follows.

$$b_{15}b_{14}b_{13}b_{12}b_{11}b_{10}b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0$$

The number of bits supported by a digital computer often depends both on the architecture of the computer and the language used to express the program. The virtually all cases, however, the number of bits is some even number of *bytes* or groups of 8 bits. In Java, variables declared as type `int` hold 32-bit (4-byte) two's complement numbers. In addition to the `int` data type, Java supports the `short` data type, which holds a 16-bit (2-byte) two's complement number, as well as the `byte` data type, which holds an 8-bit (1-byte) two's complement number.

In C, the size of variables declared as type `int` depends upon the architecture and compiler. For the AVR microcontroller and `gcc` compiler, a C `int` holds a 16-bit (2-byte) two's complement number. In addition to integers, C also supports data types that hold natural numbers (non-negative numbers). An `unsigned int` is of the same size as an `int` (2 bytes on the AVR microcontroller using `gcc`); however, it holds values ranging from 0 to $2^{16} - 1$.

We frequently use the term "signed" to refer to integer data types that use two's complement to represent negative numbers and the term "unsigned" to refer to data types that only support non-negative integers.

8.1.5 Fractional Numbers

How positional number systems represent fractions was described in Section 8.1.2. Like negative representations, however, internal to computer systems there is no way to explicitly represent the radix point. As a result, alternative techniques need to be developed to represent fractions in computer systems.

The simplest method of representing fractions is to have a fixed position for the radix point. This is called a *fixed point* number system. For example, if a number system uses 8 bits total, and by convention the radix point is in the middle (between bits 3 and 4 if the least significant bit is called bit 0 and the most significant bit is called bit 7), there are then 4 bits to the left of the radix point (representing the integer portion of the number) and 4 bits to the right of the radix point (representing the fractional portion of the number. In this example, the least significant bit (bit 0) has weight 2^{-4} or 1/16, and the most significant bit has weight 2^3 or 8.

To be clear, one could reasonably call the integer number system a fixed point system as well, since the radix point is fixed to be immediately to the right of the least significant bit. This terminology, however, is almost never used. If you hear someone describe a number system as a fixed point system, they invariably mean a fractional system in which the radix point is at some position within the bits of the number, not on the right end as is the case for an integer.

There is a notation that is commonly used (and, unfortunately, commonly abused) to denote fixed point fractional number systems, called *Q notation*. In one version, the notation Qm, n means that the fixed point number system has $m+n$ bits, with m bits to the left of the radix point and n bits to the right of the radix point. The example in the previous paragraph would therefore be a Q4.4 number system.

The notation gets less precise when fixed point numbers are combined with negative representations. It is common to use two's complement in combination with fixed point numbers (which works quite well). However, their isn't good consistency in how Q notation is used in these circumstances.

Take our 8-bit fixed point numbers above. If, in addition, they use two's complement, bit 7 is now the sign bit. So far, so good. The range of representable values is from -8 (`1000.0000`) to $7\frac{15}{16}$ (`0111.1111`). (We are showing the radix point in the previous illustration to help the reader understand the fixed point notation. Remember that the only way we know it is there is because it is defined to be there as part of the number representation.)

The confusion comes in when trying to denote this fixed point, two's com-

plement number system using Q notation. Some would still call this a Q4.4 system, and simply add that it uses two's complement. Others call this a Q3.4 system, using the logic that since bit 7 is a sign bit, it shouldn't be included in m, the count of the number of bits to the left of the radix point.

Since virtually all number systems used in computers are some multiple of 8 bits (an integral number of bytes), a reasonable guess when one is unclear which form of Q notation is being used is to make the assumption that the total number of bits is a multiple of 8.

To complicate matters even further, the most common use of fixed point numbers is in digital signal processing applications, in which it is conventional to place the radix point between the most significant bit and the next most significant bit and also to use two's complement representation. This gives a range of values that is approximately ± 1.

As an example, for a 16-bit number, the radix point is between bits 15 and 14, and the precise range of values is -1 (`1.000000000000000`) to $+\frac{32767}{32768}$ (`0.111111111111111`).

Rather than call this a Q1.15 (or Q0.15) number system, many have used a shorthand notation, arguing that the m is already known (or ambiguous, see above) and our 16-bit, two's complement fixed point number system should be denoted Q15. So, if you see a fixed point number system described as Q15, you should interpret that to be a 16-bit number with the radix point between bits 15 and 14, and if you see a number system described as Q31, you should interpret that to be a 32-bit number with the radix point between bits 31 and 30.

8.1.6 Real Numbers

A clear limitation of any fixed point fractional number system is simply the fact that the position of the radix point is fixed (i.e., it cannot vary from one number to the next). To better approximate a wider range of numbers on the real line, while maintaining the constraint that numbers must fit in a given number of bits, computer systems use a more complicated number system that includes the ability to move the radix point to the left and to the right. Not surprisingly, this type of number system is called a *floating point* system.

Floating point numbers use the conventions that we commonly understand as scientific notation. Staying for the moment in decimal notation, we can

represent any number we wish by specifying a *mantissa* and an *exponent*.

$$100 = 0.1 \times 10^3$$
$$3470 = 0.347 \times 10^4$$
$$0.0000072 = 0.72 \times 10^{-5}$$

In the examples above, the mantissa represents the significant digits and is constrained to be in the range $0 \leq$ mantissa < 1. The exponent represents the order of magnitude and is an integer. The general form is

$$M \times 10^E$$

where M is the value of the mantissa and E is the value of the exponent.

Switching from decimal to binary representation, virtually all floating point numbers in modern computer systems conform to a standard notation denoted IEEE-754 [5]. This standard describes two forms of floating point representation. The first, called *single precision*, is a 32-bit representation and the second, *double precision*, is a 64-bit representation. In both Java and C, variables declared as `float` use the IEEE-754 single precision representation and variables declared as `double` use the double precision representation.

Figure 8.2 shows a pictorial bit-level illustration of a single precision floating point number. Bit 31 is the sign bit, s, with 0 indicating the number is non-negative and 1 indicating the number is negative. Floating point numbers use sign-magnitude representation for the number as a whole. Bits 30 down to 23 are the eight bits that represent the exponent. The exponent uses excess-127 notation to represent a value, E, that can range from -126 to +127 (the bit patterns `00000000` and `11111111` will be discussed below). That is, if e is the unsigned value of bits 30 down to 23, the value of the exponent is $E = e - 127$, as long as $e \neq 0$ and $e \neq 255$. Bits 22 down to 0 are 23 bits that represent the mantissa; with the bits themselves representing the fractional part of the mantissa and an implied 1 also part of the value (i.e., if f is the value of the 23 fraction bits, with the radix point to the left of bit 22, the value of the mantissa is $M = 1 + f$).

31	30 $\cdots$ 23	22 $\cdots$ 0
sign	exponent bits	fraction bits

Figure 8.2: Layout of IEEE-754 single precision floating point numbers.

When $e \neq 0$ and $e \neq 255$, we call this a *normalized* floating point number, and the overall value is given by the following,

$$(-1)^s \times 2^{e-127} \times (1 + f)$$

where s designates the sign, e is the unsigned value of the exponent, and f is the fractional part of the mantissa.

The value with the smallest magnitude that can be represented using normalized single precision has $e = 1$ and $f = 0$, to give a value of 2^{-126}. When $e = 0$, the interpretation of the mantissa is altered, and the implied 1 is no longer included. This is called a *denormalized* floating point number. For denormalized numbers, the value of the exponent is a fixed -126, and the value of the mantissa is $M = 0 + f$, which gives an overall value given by the following expression.

$$(-1)^s \times 2^{-126} \times f$$

Denormalized numbers allow the value to get closer to zero, at the cost of fewer effective bits of precision (since the leading fraction bits are zeros). When $s = 0$, $e = 0$, and $f = 0$, the value is zero, as indicated by the expression above.

When $e = 255$, a number of special case values are supported by the standard. When $f = 0$, that is a designation for infinity (either $+\infty$ if $s = 0$ or $-\infty$ if $s = 1$). When $f \neq 0$, that is a designation that means *not a number*, which is frequently shown as `NaN`.

The layout of double precision floating point numbers closely follows the form of single precision numbers, with the only exception being that the number of bits assigned to the exponent and to the fraction are larger. This is shown in Figure 8.3. As before, there is one sign bit (now bit 63). Bits 62 down to 52 now form an 11-bit exponent, which is interpreted using excess-1023 notation. Bits 51 down to 0 now form a 52-bit fraction.

63	62 $\cdots$ 52	51 $\cdots$ 0
sign	exponent bits	fraction bits

Figure 8.3: Layout of IEEE-754 double precision floating point numbers.

Normalized numbers ($e \neq 0$ and $e \neq 2047$) have their overall value given by

$$(-1)^s \times 2^{e-1023} \times (1 + f)$$

and denormalized numbers ($e = 0$) have their overall value given by

$$(-1)^s \times 2^{-1022} \times f.$$

As in single precision, $e = 2047$ is used to indicate the special cases of infinity and `NaN`.

8.2   Text: Characters and Strings

Numbers are far from the only information that we wish to represent using binary form. Far more prevalent than numbers is text, both individual characters and sequences of characters that form words, phrases, sentences, paragraphs, and books. We will start by describing common representations for individual characters, and follow that with a description of string representations, or sequences of characters.

8.2.1   ASCII

Unlike numbers, where there is a firm mathematical foundation on which to base our binary number systems, characters are a bit more ad hoc. Typically, the representation of characters is table driven, where some sequence of binary bits corresponds to an individual character, and the relationship between the bit sequence and the character is arbitrary and defined in a table.

An early character table that still gets used extensively is the American Standard Code for Information Interchange (ASCII). It was developed in the 1960s for use with teletype machines. The basic ASCII character set corresponds to codes that are 7 bits long (we'll talk about extensions below), which each 7-bit combination representing an individual character.

The table of ASCII codes is shown in Table 8.4. It is shown using groups of three columns: the first showing the character that is represented, the second showing the value of the code in hex, and the third showing the value of the code in decimal. The hex and decimal values shown in the table are, however, merely for the benefit of us humans reading the table. In fact, the code for the letter `A` is `0100001`.

When ASCII characters are stored in a byte, which is typical, the most significant bit is set to `0`. This helps us understand why the table only goes up to `0x7f` in hex values; the leading bit is always zero.

There are a few things that are important to note about the ASCII code. First, the initial codes (and final code) don't represent characters at all, but instead are various control codes. For example, code `0x07` (`BEL`) would ring a bell on the old physical teletype machine. Table 8.5 gives the descriptions for each of the control codes; however, only a very few of them get used with any consistency.

Second, there are many possible characters that are not included in the code. For example, one cannot represent accented characters such as é or á, nor can one put the tilde over an `n` such as ñ. Neither can one represent many common currency symbols such as £, €, or ¥.

100

Table 8.4: Table of ASCII codes.

Char	Hex	Dec	Char	Hex	Dec	Char	Hex	Dec	Char	Hex	Dec
NUL	00	0	SP	20	32	@	40	64	`	60	96
SOH	01	1	!	21	33	A	41	65	a	61	97
STX	02	2	"	22	34	B	42	66	b	62	98
ETX	03	3	#	23	35	C	43	67	c	63	99
EOT	04	4	$	24	36	D	44	68	d	64	100
ENQ	05	5	%	25	37	E	45	69	e	65	101
ACK	06	6	&	26	38	F	46	70	f	66	102
BEL	07	7	'	27	39	G	47	71	g	67	103
BS	08	8	(	28	40	H	48	72	h	68	104
HT	09	9	)	29	41	I	49	73	i	69	105
LF	0a	10	*	2a	42	J	4a	74	j	6a	106
VT	0b	11	+	2b	43	K	4b	75	k	6b	107
FF	0c	12	,	2c	44	L	4c	76	l	6c	108
CR	0d	13	-	2d	45	M	4d	77	m	6d	109
SO	0e	14	.	2e	46	N	4e	78	n	6e	110
SI	0f	15	/	2f	47	O	4f	79	o	6f	111
DLE	10	16	0	30	48	P	50	80	p	70	112
DC1	11	17	1	31	49	Q	51	81	q	71	113
DC2	12	18	2	32	50	R	52	82	r	72	114
DC3	13	19	3	33	51	S	53	83	s	73	115
DC4	14	20	4	34	52	T	54	84	t	74	116
NAK	15	21	5	35	53	U	55	85	u	75	117
SYN	16	22	6	36	54	V	56	86	v	76	118
ETB	17	23	7	37	55	W	57	87	w	77	119
CAN	18	24	8	38	56	X	58	88	x	78	120
EM	19	25	9	39	57	Y	59	89	y	79	121
SUB	1a	26	:	3a	58	Z	5a	90	z	7a	122
ESC	1b	27	;	3b	59	[	5b	91	{	7b	123
FS	1c	28	<	3c	60	\	5c	91	|	7c	124
GS	1d	29	=	3d	61	]	5d	93	}	7d	125
RS	1e	30	>	3e	62	^	5e	94	~	7e	126
US	1f	31	?	3f	63	_	5f	95	DEL	7f	127

Table 8.5: Control codes and descriptions.

Code	Description		Code	Description
NUL	null		DLE	data link escape
SOH	start of heading		DC1	device control 1 (X-ON)
STX	start of text		DC2	device control 2
ETX	end of text		DC3	device control 3 (X-OFF)
EOT	end of transmission		DC4	device control 4
ENQ	enquiry		NAK	negative acknowledgment
ACK	acknowledge		SYN	synchronous idle
BEL	bell		ETB	end of transmission block
BS	backspace		CAN	cancel
HT	horizontal tabulation		EM	end of medium
LF	line feed		SUB	substitute
VT	vertical tabulation		ESC	escape
FF	form feed		FS	file separator
CR	carriage return		GS	group separator
SO	shift out		RS	record separator
SI	shift in		US	unit separator
SP	space		DEL	delete

It is clear to see that the ASCII code was not designed with international use in mind. It is very (American) English centered, which is not surprising given its origins in the U.S.; however, the limitations illustrated above strongly motivate expansion.

In spite of its limitations, ASCII is the original character representation used in the C language, and the `char` data type in C is one byte, sized to hold an individual ASCII character.

8.2.2   Unicode

A number of expansions to the ASCII code have been proposed. We will focus our attention on the Unicode family of character encodings. The Unicode standard is an attempt to handle most of the planet's languages consistently, and number of Unicode Transformation Format (UTF) encodings are defined for Unicode characters, including UTF-8, UTF-16, and UTF-32.

UTF-8 is a variable length encoding of the Unicode character set that maintains backward compatibility with ASCII. It uses 8-bit code units, in which the first 128 codes are the same as their ASCII counterparts. Additional characters are encoded as multi-byte sequences.

UTF-16 is a variable length encoding of the same Unicode character set; however, it uses 16-bit code units, meaning that the minimum size of any

character is 2 bytes. As a result of this decision, many more characters fit into a single code unit than is the case when using UTF-8. This includes all the characters from almost all Latin alphabets as well as Greek, Cyrillic, Hebrew, Arabic, and several others. Characters in many oriental languages (e.g., Chinese, Japanese, Korean) require 4 bytes per character when encoded in UTF-16.

UTF-32 is a fixed length encoding of the Unicode character set. As such, every character requires 4 bytes to be encoded.

Since a `char` in C is one byte, it can reasonably store UTF-8 single-byte characters. Java's internal representation is UTF-16, and the `char` data type in Java is 2 bytes.

8.2.3 String Representations

Strings are composed of sequences of characters, and characters can be represented in any of the ways indicated above. Independent of the character encoding, however, there two additional design decisions that must be made when representing strings, "How do we indicate the length of the string?" and "What data structure do we use to store the individual character codes?"

In general, there are two approaches to representing string length, and different languages use both of these approaches. In both cases, it is conventional to store the characters themselves in an array whose type is appropriate for the character set employed (e.g., the `char` type in C, which is one byte in size, or the `char` type in Java, which is two bytes in size).

1. **End marker** – The first approach to representing string length is to use a designated symbol to mark the end of the string in the array storing the characters. Note that this mechanism relies on the existence of a position available in the array (i.e, the array length must be at least one greater than the string length).

   This is the approach used in the C language, with a NULL character (0x00, `'\0'`) used as the end marker.

2. **Explicit count** – The second approach is to use an explicit count of the characters in the string. This count is maintained separately from the array storing the actual characters.

   This is the approach used in the Java language, in which the `String` class maintains (internally, as `private` instance variables) both an array of characters and a count for each `String` object that is created.

This is also the approach used for transmitting UTF-8 strings in a stream. A 16-bit character count is followed by the sequence of individual UTF-8 characters.

8.3 Images

Consider the following sequence of bits: 0x002400081881423c. In binary, this is:

```
0000 0000 0010 0100 0000 0000 0000 1000
0001 1000 1000 0001 0100 0010 0011 1100
```

If 0 represents a white spot, and 1 represents a black spot, this yields the sequence of spots shown in Figure 8.4.

Figure 8.4: Sequence of spots that result when 0 represents a white spot and 1 represents a black spot.

Next we will arrange these white and black spots in rows, one byte (8 bits) per row. This results in the image shown in Figure 8.5.
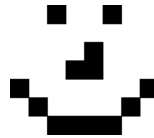

Figure 8.5: Image that results when spots are arranged in rows.

While fairly simple, this example illustrates many of the conventions used generally in image representation.

1. Each bit of the example image specification corresponds to one position in the image, commonly called a *pixel*. This will be generalized below to more than one bit per pixel.

2. To correctly recreate the image, the number of pixels per row must be known. In the case of the example it was 8 pixels per row. Other images are, of course, much larger.

3. The sequence of image pixel data typically starts in the upper-left corner, which is designated as coordinate position $(0,0)$, and proceeds across the first row. This is followed by the pixel data for the second row, starting at coordinate position $(1,0)$, and continuing until the final row.

4. For an image that is n pixels tall and m pixels wide, the upper left coordinate is $(0,0)$, the upper right coordinate is $(0, m-1)$, the lower left coordinate is $(n-1, 0)$, and the lower right coordinate is $(n-1, m-1)$.

The above conventions apply to raw, or uncompressed, images. It is very common to use compression techniques to reduce the storage requirements of images. A frequently used technique is specified in the JPEG standard [9], and images compressed using this technique normally are stored with a `.jpg` file extension.

8.3.1 Monochrome Images

In the example image of Figure 8.5, each pixel is represented by a single bit, and a 0 encodes a white spot while a 1 encodes a black spot. The next step to more interesting images happens when, instead of a single bit per pixel, each pixel is represented by a number that encodes shades of gray (between white and black). If each pixel is represented by a byte, the possible values range from 0 (white) to 255 (black). Such images are called *monochrome* images, since they only include a single color (black), and vary its intensity.

An example of a 512 by 512 pixel monochrome image is illustrated in Figure 8.6. With one byte dedicated to each pixel, and 262,144 ($= 512 \times 512$) pixels, the memory required to store the image is 262,144 bytes. It gives a much more realistic view than the simple image of Figure 8.5; however, it still leaves quite a bit to be desired.

8.3.2 Color Images

We can extend the concept of monochrome images to include color by adding information to each pixel that represents the color of that pixel. The most common approach to doing this is to use three values for color representation: red, green, and blue. With one byte for each color at each pixel, our 512 by 512 image will now require 786,432 bytes of memory.

The color image that corresponds to the same picture as Figure 8.6 is shown in Figure 8.7.

Figure 8.6: Monochrome image that is 512 pixels tall and 512 pixels wide. (Photo courtesy Tracy L. Chamberlain, © 2013.)
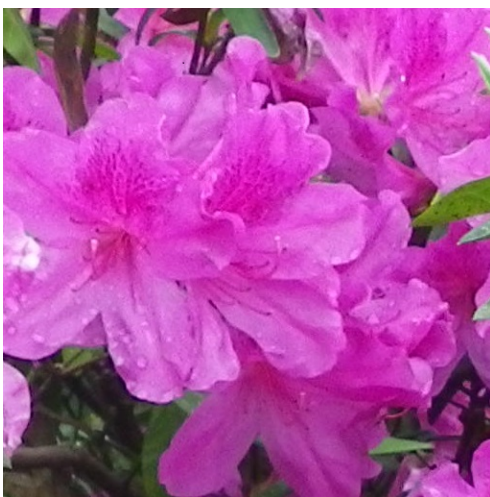

Figure 8.7: Color image that is 512 pixels tall and 512 pixels wide. (Photo courtesy Tracy L. Chamberlain, © 2013.)

9 User Interaction

Any book that claims to describe how computers can interact with the real world should not neglect the fact that users are an important aspect of the real world that deserve special attention. Some of the interface mechanisms described in earlier chapters (e.g., LED indicators, pushbuttons) constitute a primitive form of user interaction. However, we have come to expect a much richer form of interaction with the various digital devices that frequently help define our day-to-day lives.

In this chapter, we will describe both physical mechanisms that enable interaction (input and output) between the computer and its human users as well as common techniques for exploiting the interaction capabilities.

9.1 Visual Display

The first set of techniques we will describe are those that support a visual display, the output from the microcontroller is to be viewed by a human user.

9.1.1 Display Technologies

We have already discussed a simple visual display technology in Chapter 2, an individual light-emitting diode (LED). Figure 2.4, repeated here as Figure 9.1, illustrates the configuration in which the LED will be on when the digital output pin is LOW.

As described in Chapter 4, the brightness of the LED can be controlled if it is connected to one of the pulse-width modulated output pins. If we retain the schematic configuration of Figure 9.1, the LED will have maximum brightness for an analog output value of 0 and minimum brightness for an analog output value of 255.

An alternative display technology that takes significantly less power to operate is a liquid crystal display (LCD). An LCD exploits the polarization
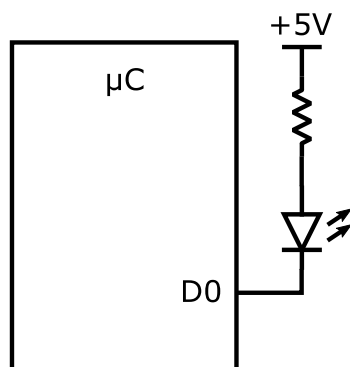
Figure 9.1: Schematic diagram of LED digital output with active LOW control.

properties of light to either transmit or block light transmission at each specific position of the display. As such, an LCD does not emit light itself, it relies on either reflected light from in front of the display or a separate light source positioned in back of the display.

Unlike an LED, which simply requires a current flowing through it to be illuminated, an LCD requires significantly more complex control circuitry to operate. As a result, it is more common to interface LCD displays using the I^2C bus.

9.1.2 7-segment Displays

A common display on small, hand-held devices is the *7-segment display*, named by the fact that there are 7 individual elements (or segments) that can be independently on or off. An illustration of a 7-segment display is shown in Figure 9.2. They can be constructed using either LED or LCD technology, and both are frequently used for numeric display purposes.
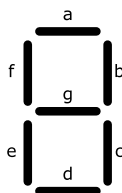

Figure 9.2: Organization of 7-segment displays.

The organization of a 7-segment display is such that it is fairly straight-forward to display any of the decimal digits, 0 through 9, by turning on the appropriate selection of segments. Given the segment labeling shown in the figure (which is fairly standard), we can form a 0 by turning on segments 'a', 'b', 'c', 'd', 'e', and 'f', leaving segment 'g' off. Alternatively, we can form a 1 by turning on segments 'b' and 'c', leaving all of the other segments off. All of the combinations of segments on and off to show the digits 0 through 9 are compiled in Table 9.1.

Table 9.1: Indication of on and off segments for 7-segment display.

Digit	a	b	c	d	e	f	g	Display
0	on	on	on	on	on	on	off	
1	off	on	on	off	off	off	off	
2	on	on	off	on	on	off	on	
3	on	on	on	on	off	off	on	
4	off	on	on	off	off	on	on	
5	on	off	on	on	off	on	on	
6	on	off	on	on	on	on	on	
7	on	on	on	off	off	off	off	
8	on	on	on	on	on	on	on	
9	on	on	on	on	off	on	on	

If we encode the contents of Table 9.1 into an array, with the least significant 7 bit positions of each byte corresponding to the 7 segments of the display, and the 7-segment display configured so that a LOW output on a pin lights the segment (e.g., like in Figure 9.1), Figure 9.3 shows how to control the display in a sketch.

There are a few things worth looking at carefully in this sketch. First, the array segments[] is a direct transcription of the information in Table 9.1, just coded in hexadecimal. Second, the use of the segments[] array in the if statement illustrates a common design practice in the C language. Here, we

```
// segment pins are 17 to 23 ('a' to 'g')
const int segmentPin[] = {17, 18, 19, 20, 21, 22, 23};

// encoding of segment table
const byte segments[] = {0x7e, 0x30, 0x6d, 0x79, 0x33,
                         0x5b, 0x5f, 0x70, 0x7f, 0x7b};

void displayDigit(int digit} {
  for (int i=0; i<7; i++) {
    if (segments[digit] & (1 << i)) {
      digitalOut(segmentPin[6-i], LOW); // turn segment on
    } else {
      digitalOut(segmentPin[6-i], LOW); // turn segment off
    }
  }
}

void setup() {
  for (int i=0; i<8; i++) {
    pinMode(segmentPin[i], OUTPUT); // set pin to digital output
  }
}

void loop() {
  // remainder of sketch, calling displayDigit() when desired
}
```

Figure 9.3: Fragment of sketch to control 7-segment display.

perform a bit-wise AND (the '&' symbol) of the entry in the segments array with a 1 that has been left-shifted (the '<<' symbol) i times. The result of this operation will be zero (all bits are 0) if the ith bit (counting from the right) of segments[digit] is 0, and will be non-zero (the ith bit will be 1) if the ith bit of segments[digit] is 1. Third, the index into the array segmentPin[] in the digitalOut() invocations is 6-i because the segmentPin[] array is ordered 'a' to 'g' and we extract the segments in the opposite order. We could change this index to i by reordering the pins in the segmentPin[] array.

With a 7-segment display, the most common usage is for numeric informa-

tion; however, if we are displaying hexadecimal numbers, the digits 0 through 9 are not sufficient. We can, of course, build any symbol that we wish using the individual segments of the display. Table 9.2 extends Table 9.1 (i.e., adds additional rows), illustrating how one can use a 7-segment display to represent hexadecimal output as well.

Table 9.2: Hexadecimal digits in a 7-segment display.

Digit	a	b	c	d	e	f	g	Display
A	on	on	on	off	on	on	on	
B	off	off	on	on	on	on	on	
C	on	off	off	on	on	on	off	
D	off	on	on	on	on	off	on	
E	on	off	off	on	on	on	on	
F	on	off	off	off	on	on	on	

Another thing to notice about a 7-segment display is that the number of segments is less than the number of bits in a byte. When we encoded which segments are to be on or off in the `segments[]` array of Figure 9.3, the high-order (most significant) bit of every entry was 0, because there was no meaning associated with that bit.

Designers have taken advantage of that fact and added an additional indicator to the traditional 7-segment display to serve as a decimal point. Illustrated in Figure 9.4, the eighth segment is denoted by the label `h` and, like each of the other segments, can be turned on and off individually.
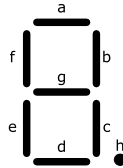
Figure 9.4: 7-segment display including decimal point.

The above approach works well for a single-digit display, but frequently we are interested in displays with more capability. In the following section,

we will discuss approaches to arbitrary characters (and images), but first we will just extend our 7-segment display to more digits.

If we want to implement a 4-digit, 7-segment display, clearly the easiest thing to do is simply position 4 copies of an individual 7-segment display right next to one another, as illustrated in Figure 9.5. The difficulty comes in when we try to use a sketch like that of Figure 9.3 to control all 4 digits, we don't have enough output pins to control all $4 \times 7 = 28$ digital outputs required.
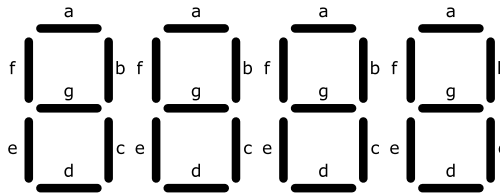
Figure 9.5: A 4-digit, 7-segment display.

We can solve this problem by time multiplexing between the 4 digits. Instead of wiring the anode of each segment's LED through a limiting resistor to $+5$ V, we wire the anode to a digital output pin (4 pins for 4 digits, assuming the 7-segment display has a common anode) and the limiting resistor is wired in series with the cathode. The cathodes of segment 'a' for each digit are then wired together and to a single digital output, with similar wiring for the remaining cathodes of each segment. This is illustrated in Figure 9.6, in which the 7-segment displays are shown, the common anode wires to the upper left are to be connected to digital output pins, and the bused cathode wires to the lower left are to be connected to digital output pins. Note that this requires $7 + 4 = 11$ pins, instead of the earlier 28 pins.
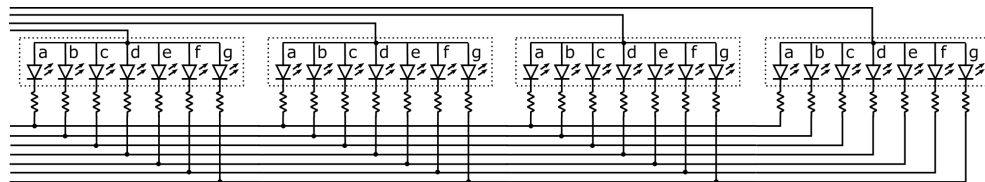
Figure 9.6: Wiring for 4-digit, 7-segment, common-anode LED display.

In this way, we can send a `HIGH` to one (and only one) of the digits' anodes while sending a `LOW` to the segments that we wish to be on for that digit. A short time later, we switch the `HIGH` output to the anode to the next digit,

with the third and fourth digits following on a similar schedule. A sketch that does this is shown in Figures 9.7 and 9.8.

What results is a repeating sequence where one quarter of the time each 7-segment display is showing the digit that is desired. If the time associated with each digit is approximately 5 ms, the entire 4-digit number is illuminated every 20 ms, which is way too fast for our eyes to observe the fact that it is flashing. To us it looks like each 'on' segment is on continuously.

9.1.3 Pixel-oriented Displays

The obvious limitation of 7-segment displays is that they are severely limited in the things that they can effectively portray. While OK for numbers, and usable for hex numbers in a pinch (assuming our users don't get confused with a `b` and think it is a `6`), there are many other things we would like to be able to output for users. The first, and most obvious, is arbitrary alphanumeric characters.

The rest of the display technologies that we will consider are all *pixel-based*. By this, we mean that individual spots in the display (called *pixels*) can each be on or off, and the collection of all of the pixels forms an image to be seen by the user. In virtually all pixel-oriented displays, the pixels themselves are organized in a rectangular grid (e.g., an m by n display has m columns and n rows of pixels), and hence they are sometimes referred to as a *matrix display*.

The technology at each pixel can vary quite a bit. The simplest pixel-oriented displays have an LED at each pixel (see Figure 9.9, which shows the internal schematic of a 5 by 7 display, an appropriate size for one character). However, other technologies include LCDs, plasma displays, vacuum fluorescent displays (not too common anymore), e-paper, and MEMS micro-mirrors (used in DLP projectors). Some are monochrome (only support a single color at each pixel), while others support a range of colors at each pixel.

The number of pixels also varies widely, from as small as 5 by 7 pixels for displaying an individual character to high-resolution 1080p displays (with 1920 by 1080 pixels) for HDTV, and even larger. We will constrain our discussion to the smaller end of the range.

Let's return to Figure 9.9. The first thing to note is that the display does not provide individual connections to each pixel's LED, but rather ties all the anodes of the LEDs in each column together and ties all the cathodes of the LEDS in each row together. Connections are then made to the columns and to the rows, $5 + 7 = 11$ connections, rather than the $5 \times 7 = 35$ connections

113

```
// digit pins are 24 to 27 (left digit 0 to right digit 3)
const int digitPin[] = {24, 25, 26, 27};

// segment pins are 17 to 23 ('a' to 'g')
const int segmentPin[] = {17, 18, 19, 20, 21, 22, 23};

// encoding of segment table
const byte segments[] = {0x7e, 0x30, 0x6d, 0x79, 0x33,
                         0x5b, 0x5f, 0x70, 0x7f, 0x7b};

int digitCounter = 0;
int digitDivisor = 1000;
int valueToDisplay = 0;

void setDigit(int whichDigit) {
  for (int i=0; i<4; i++) {
    if (i == whichDigit) {
      digitalOut(digitPin[i], HIGH);
    }
    else {
      digitalOut(digitPin[i], LOW);
    }
  }
}


void displayDigit(int digit} {
  for (int i=0; i<7; i++) {
    if (segments[digit] & (1 << i)) {
      digitalOut(segmentPin[6-i], LOW); // turn segment on
    } else {
      digitalOut(segmentPin[6-i], LOW); // turn segment off
    }
  }
}
```

Figure 9.7: Sketch to control 4-digit, 7-segment display (continued in Figure 9.8).

```
void setup() {
  for (int i=0; i<4; i++) {
    pinMode(digitPin[i], OUTPUT); // set pin to digital output
  }
  for (int i=0; i<8; i++) {
    pinMode(segmentPin[i], OUTPUT); // set pin to digital output
  }
}

void loop() {
  // set valueToDisplay to something interesting
  setDigit(digitCounter);
  displayDigit(valueToDisplay / digitDivisor);
  digitCounter++;
  digitDivisor /= 10;
  if (digitCounter == 4) {
    digitCounter = 0;
    digitDivisor = 1000;
  }
  delay(5);
}
```

Figure 9.8: Sketch to control 4-digit, 7-segment display (continued from Figure 9.7).

that would be required otherwise. (Some displays swap the anode and cathode connections, so pay attention to the schematic for your particular display.)

The second thing to note is that the display only contains the LEDs, yet driving an LED from a microcontroller requires current limiting resistors. Don't just connect the columns and rows directly to your microcontroller, as this can damage either the display, the mircrontroller, or both.

We control the display in essentially the same way that we controlled the 4-digit, 7-segment display in the previous section, we time multiplex groups of LEDs and cycle through the groups faster than our eyes can respond. There are a number of possible design options for doing this. In what follows below, we will cycle through the columns (with cycling through the rows being an alternative design option).

To light an individual LED, the voltage on the column wire must be higher
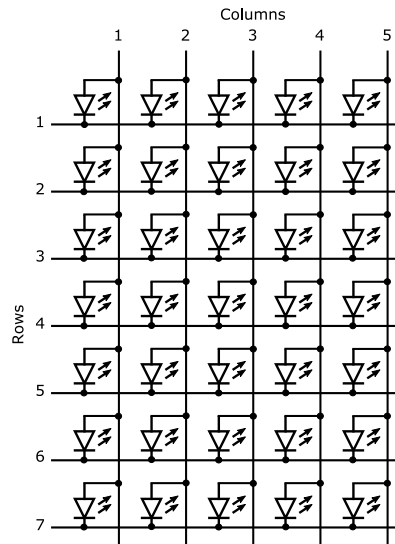
Figure 9.9: 5 by 7 matrix display.

than the voltage on the row wire. We therefore cycle through the columns by connecting each column to a digital output pin and ensuring that one column output is `HIGH` and the other four column outputs are `LOW`.

With the columns under appropriate control, we now need to send the appropriate rows `LOW` to light the LEDs that are to be lit (if the row is `HIGH`, the LED will not light). We connect a current limiting resistor to each row, and connect the other side of the resistor to a digital output pin. Now, for each column, we can light the desired LEDs in each row by sending the output connected to that row `LOW`.

9.2 Hearing and Other Senses

9.2.1 Sound

The second set of techniques we will describe are those that support audio output. In this case, the microcontroller is generating output that the human is intended to hear.

It is important when considering sound to make a clear distinction between two different approaches to sound generation. In the first approach, the microcontroller is managing a sound waveform that is generated by some

116

peripheral device (e.g., a buzzer), and the control of the sound is limited to turning it on or off, or possibly controlling its volume.

An on/off control is reasonable to accomplish with a digital output, as described in Chapter 2 (see Figure 2.5). Controlling the volume of a sound can be accomplished with an analog output, as described in Chapter 4 (see Figure 4.7). Here, we have control over some of the properties of the sound signal, but "what it sounds like" is not under direct software control.

In the second approach, the waveform itself is created in software. This isn't practical on the Arduino Uno platform we've been using up to this point, for reasons we will describe in a moment, but the approach is worthy of discussion in any event.

Sound is a pressure wave that is continuous in intensity and also is continuous in time. In the physical world, that implies that no matter how small a difference one considers between two values (either pressure or time), there is always a differential that is half of what was just considered. In both pressure and in time, any differential value (i.e., difference between two values) can be decreased to something smaller.

In the digital world, both of these continuous axes are approximated with discrete values. Digital-to-analog converters are commanded by an integer value parameter (in our case, the `analogWrite()` can take on values between 0 and 255, but does not support a non-integer value such as 12.75). Also, when two subsequent digital-to-analog conversions happen, there is no way to instruct the analog output value between the two conversions. The conversions are at two discrete points in time. As such, the digital world approximates an analog waveform by a series of discrete *samples*.

This is illustrated in Figure 9.10, which shows two full cycles of a sine wave at 0.5 kHz frequency (the period of the sine wave is 2 ms). The points on the graph are the output values (which happen at discrete points in time with a sample period of 100 μs, or 10 kS/s). The values on the vertical axis are also constrained to be integers (although that isn't as easy to see at this scale).

This approximation is normally perfectly acceptable. The values between discrete samples are shown as the curve in Figure 9.10. Under a few (normally reasonable) assumptions, e.g., the sample rate is sufficiently faster than the highest frequency present in the waveform, the continuous-valued waveform actually doesn't contain any new information that isn't already present in the discrete samples.

The assumptions mentioned in the previous paragraph are our first clue as to why this doesn't work very well on the Arduino Uno platform. The sample rate supported by the digital-to-analog output on the Arduino Uno is

117
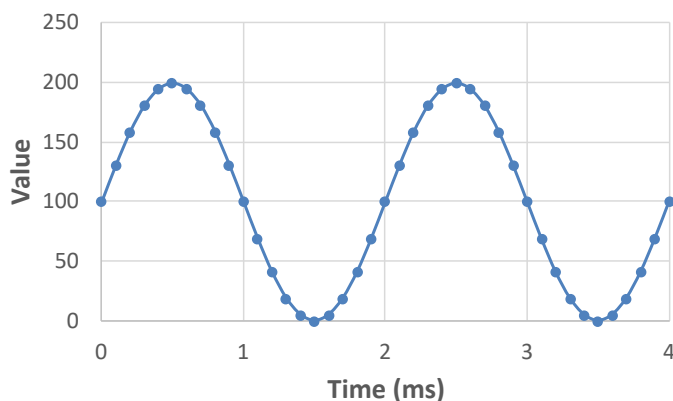
Figure 9.10: 500 Hz sampled sine wave.

no where near fast enough to support the frequency range needed for sound that can be perceived by humans.

The second reason this approach doesn't work very well on the Arduino Uno platform is that the amount of memory needed to store the sample values for an interesting waveform (not just a single tone generated by a sine wave) is much larger than the quantity of data memory available on the microprocessor.

The reader might have noticed at this point that while throughput the book we have been referring to the microprocessor platform as an Arduino, but in this section we became much more specific and referred to it as the Arduino Uno. This is because there are different versions of the Arduino platform, and some of the more capable versions have faster digital-to-analog converters and more data memory. These more capable Arduinos can, reasonably, generate sound by sending a series of samples to the analog output (and then driving a speaker with the resulting waveform).

9.2.2   Other Senses

9.3   User Input

9.4   User Interface Design

10 Computer Architecture

The instruction set architecture (ISA) of a processor is traditionally seen as the boundary between the hardware world and the software world. It is essentially the abstraction that allows hardware designers and software designers to co-exist without constantly having to re-engineer everything they do because of choices made within the other discipline.

In this chapter, we will consider the underlying computer architecture that makes up the AVR microcontroller family. Included in this is the set of *machine instructions* that are directly executable by the microcontroller. These machine instructions constitute the *machine language* of the microcontroller. We will also introduce the human-readable and -writable variation of the machine instructions, commonly called *assembly language*, which is primarily the subject of the next chapter.

10.1 Basic Computer Architecture

A high-level view of the AVR microcontroller family computer architecture is shown in Figure 10.1. When all of these components are included within a single chip, the chip is referred to as a *microcontroller*. In a larger, more complex processor, the program memory, data memory, and peripherals are typically off-chip, and the chip is referred to as a *microprocessor*. These definitions, however, are far from ubiquitous.

10.1.1 Architecture Components

There are a number of components that make up an AVR microcontroller. Items that are included in a microcontroller that are external to a microprocessor are the program memory, the data memory, and the peripherals. We will start our discussion with these components.

The AVR has what is called a *Harvard architecture*, in which the program memory and the data memory are physically separate memory subsystems,

119
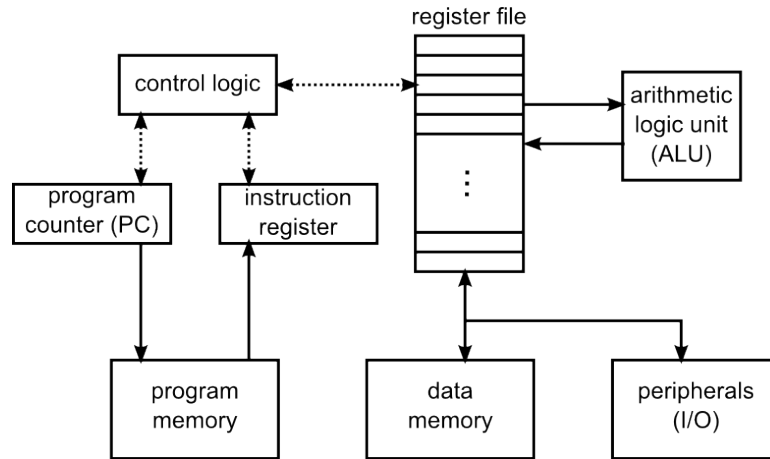
Figure 10.1: AVR microcontroller architecture. All of these components reside within an individual integrated circuit chip.

often with distinct properties. In the case of the AVR, the machine instructions are each 16 bits wide, and the program memory is addressable at the instruction level (i.e., each address in the program memory references a 16-bit storage location which holds one instruction). The data memory is logically 8 bits wide, meaning that each data memory address references an 8-bit storage location which holds one byte.

Microcontrollers generally also incorporate some number of peripherals onto the chip as well, and the AVR is no exception. The AVR microcontroller includes digital inputs, digital outputs, analog inputs, and analog outputs in its peripheral set.

The remaining components that make up the microcontroller are sometimes referred to as the *processor* components. The register file is the set of memory elements that are internal to the processor yet visible to the programmer (not necessarily a high-level language programmer, but a programmer who is writing machine language or assembly language). The arithmetic/logic unit (ALU) is the component that is responsible for most data manipulation operations (e.g., addition, subtraction, logical and, logical or, etc.).

Two additional memory elements are the program counter (PC) and the instruction register (IR). The program counter is responsible for keeping track of the next instruction to be executed, and the instruction register holds the contents of that instruction inside the processor. The control logic is responsible for coordinating all of the operations of the microcontroller. The set of

operations that result in running code is frequently called the *fetch-decode-execute cycle*.

10.1.2 Fetch-Decode-Execute Cycle

The fetch-decode-execute cycle proceeds as follows:

Fetch The address that is currently in the program counter is used to access the program memory and retrieve (or "fetch") the instruction to be executed. This instruction is placed in the instruction register.

Decode The instruction currently in the instruction register (that has just been fetched) is provided to the control logic, which interprets (or "decodes") the instruction to decide three things:

1. What *operation* is to be performed by the instruction. E.g., is it an addition operation or is it a conditional branch operation? This is frequently called the *opcode*.

2. What *operands* comprise the data to be operated upon and the location to store any results.

3. What is the next instruction to be executed. Unless explicitly changed, the default next instruction is the one at the next address following the current instruction. Control flow instructions, however, can alter this default.

Execute The actual actions to be performed by the instruction are carried out (or "executed"). If the instruction is an arithmetic or logical operation, the ALU is involved. If the instruction is a load or store, memory is accessed. If the instruction is a branch, the program counter is altered to a new value.

While there are many variations from one processor to another, the above notion of a fetch-decode-execute cycle is common to almost all of them.

10.2 Instruction Set Architecture (ISA)

The instruction set architecture (ISA) is an abstraction boundary that essentially forms a contract between the hardware world and the software world. It defines what is observable and directly controllable by software, yet does not prescribe how the hardware implements the functions.

Traditionally, an ISA comprises the following four components:

- **register file** – the programmer-visible storage within the processor (visible to the machine language programmer, not a high-level language programmer).

- **memory model** - the logical organization of the memory (as viewed by the machine language program).

- **instruction set** – the collection of machine language instructions that are directly executable by the processor.

- **operating modes** – some processors have subsets of the instructions that are privileged based on being in a given "mode".

We will consider each of the above ISA elements in turn. By necessity, we will not cover the complete ISA of the AVR microcontroller, but will instead provide a representative subset. Full details of the AVR can be found in the instruction set manual from Microchip [6].

10.2.1 Register File

The register file is the machine language program's view of storage that is internal to the processor. In the AVR microcontrollers, this is comprised of 32 general purpose registers (named `r0` to `r31`) and a status register (named `SREG`).

The general purpose registers are each 8 bits wide. To store 16-bit values, they are normally paired (e.g., `r31:r30`) with the high-order bits of the 16-bit value going in the odd-numbered (larger label) register and the low-order bits going in the even-numbered (smaller label) register.

The general purpose registers can hold data or addresses; however, registers `r26` through `r31` are commonly used for addresses, and as such have synonyms associated with each of three register pairs. The register pair `r27:r26` is also known as X, `r29:r28` is also known as Y, and `r31:r30` is also known as Z. We will see examples of the use of these register pairs for addressing later, for now it is sufficient to know that they have more than one name.

The status register, `SREG`, is an 8-bit register that gives information about what has happened previously on the processor. Informally, what is the processor's "status"?

More precisely, it comprises 8 individual status bits, each of which has its own dedicated function. Tabletbl:sreg gives the meaning and shorthand label for each bit of the status register.

Table 10.1: Bits of status register `SREG`.

Bit	7	6	5	4	3	2	1	0
Label	I	T	H	S	V	N	Z	C
Meaning	interrupt enable	transfer bit	half carry	sign bit	signed overflow	negative flag	zero flag	carry flag

As an example of how the status register operates, when the processor executes an `add` instruction, if the result of the addition is negative (i.e., if the most significant bit of the result is 1), the `N` bit in the `SREG` will be set to 1, otherwise it will be set to 0. Likewise, if the result of the addition is zero, the `Z` bit will be set to 1, otherwise it will be set to 0. Subsequent instructions (e.g., conditional branches) can then test the value of individual bits in `SREG` and act accordingly.

10.2.2 Memory Model

The memory model is essentially the machine language programmer's view of system memory. In the AVR microcontroller family, there are several items that can be accessed through the memory interface.

Program Memory

The AVR's Harvard architecture means that the program memory is separate from the data memory. The program memory is constructed using flash memory technology, which has the advantage that it is non-volatile. It retains its contents even when the power is removed.

As a result, if you provide power to an Arduino board and don't download a new program, it will run the last program that was downloaded, since that is what the processor finds when it fetches instructions from the program memory.

Because most instructions in the AVR family are 16 bits in length, the designers chose to have the program memory organized around 16-bit words. This means that an individual address that points to a single location in the program memory is referring to a 16-bit value.

The program memory is further divided into application program memory and a boot loader. The boot loader is responsible for receiving new programs from the USB link, loading them into application program memory, and starting them up. If no new program comes from the USB, then the boot loader starts up the program that is currently resident in the application program memory.

123

Data Memory

The primary memory used by programs is the data memory. It is the memory that is accessed on load and store instructions. Data memory is *byte addressable*, meaning that each address refers to an individual byte, or 8 bits.

The largest region of data memory is SRAM that is internal to the chip and available to store variables or anything else desired by the program. SRAM, or Static Random Access Memory, is a memory technology that supports single-cycle read and write operations to any location (address) in the memory. SRAM is volatile, so when power is lost, the contents stored in memory are not retained.

In addition to the internal SRAM, addresses in data memory are used to access other structures, described below.

Non-volatile Memory

The third memory type is program-accessible non-volatile memory, which is constructed using EEPROM technology. EEPROM, or Electrically Erasable Programmable Read Only Memory, is a memory technology that, like flash, does not lose its contents when power is removed. Like the data memory, it is 8 bits wide and byte addressable. The major difference between EEPROM and flash is that EEPROM can be altered (erased and rewritten) one byte at a time, where flash is typically bulk erased and then rewritten.

Peripherals

Chapters 2 through 5 discussed approaches to send signals in and out of the microcontroller. Devices that are attached in this way are called *peripherals*. In the AVR instruction set architecture, there are two mechanisms for accessing peripherals: (1) `in` and `out` instructions, and (2) through the memory interface.

The `in` and `out` instructions allow for quick (single execution cycle) access to 64 addresses on what is called an *I/O bus*. Individual peripherals are assigned to unique addresses on this bus, and it is common to call individual locations on the bus *I/O registers*. With 64 addresses on the I/O bus, the range of I/O addresses is from 0x00 to 0x3F.

In addition to being available on the I/O bus, the I/O registers can also be accessed via the memory interface (i.e., they also have addresses in the data memory). The I/O registers start at memory location 0x0020, so that I/O address 0x00 accesses the same I/O register as memory location 0x0020, I/O

address 0x01 is the same I/O register as memory location 0x0021, and I/O address 0x3F is the same as memory location 0x005F.

Because the number of needed addresses has grown over time, the AVR family supports an extended set of I/O registers, beyond the original 64. These extended I/O registers are are not available via the `in` and `out` instructions, but are only available via the data memory, starting at data memory location 0x0060 and running through 0x00FF. The internal SRAM then starts at memory location 0x0100.

In addition to the above items, the memory interface can also be used as an alternative path to access the processor's state. This includes not only the register file described in Section 10.2.1, but also additional registers that comprise part of the processor's workings. The register file is accessible as data memory addresses 0x0000 through 0x001F, and the remaining registers are included in either the I/O registers or the extended I/O registers.

Table 10.2 shows some of the peripherals that can be accessed by the program through the memory interface. Both the data memory address and the I/O address are shown along with the label and description of the peripheral.

Several of the table entries are worth specific mention. Memory addresses 0x0023 through 0x002B (I/O addresses 0x03 through 0x0B) comprise three groups of addresses, and within each group there are three addresses associated with an *I/O port*. These 8-bit registers are the interface to the digital output and input pins described in Chapters 2 and 3. Each port is associated with up to 8 physical pins on the chip.

Within each group, `DDRx` is the data direction register, which is used to set whether each pin is an `INPUT` or an `OUTPUT` in response to a call to `pinMode()`. The `PINx` addresses are used to read the input values for `digitalRead()`, and the `PORTx` addresses are used to write output values using `digitalWrite()`.

Memory address 0x005F (I/O address 0x3F) is the status register, `SREG`, described in Section 10.2.1. Memory address 0x0060 is a watchdog timer control register. A *watchdog timer* is a circuit that independently keeps track of time and is used to ensure that the software on the processor continues to operate. Based on a settable timeout value, the software is required to "reset" the watchdog time prior to the timeout. If this watchdog timer reset does not happen, the watchdog timer circuitry will reset the processor, in an attempt to correct whatever problem caused the software to miss its deadline.

Other entries in the I/O address space (not shown in the table) provide access to the free-running counter used for timing that was described in Chapter 6 and control other aspects of the processor's operation.

125

Table 10.2: Peripherals accessible through the memory interface.

Memory Address	I/O Address	Label	Meaning
0x0000	–	r0	general purpose register r0
0x0001	–	r1	general purpose register r1
.	.	.	.
.	.	.	.
.	.	.	.
0x001E	–	r30	general purpose register r30
0x001F	–	r31	general purpose register r31
0x0020	0x00	–	reserved
0x0021	0x01	–	reserved
0x0022	0x02	–	reserved
0x0023	0x03	PINB	input pins port B
0x0024	0x04	DDRB	data direction register port B
0x0025	0x05	PORTB	data register port B
0x0026	0x06	PINC	input pins port C
0x0027	0x07	DDRC	data direction register port C
0x0028	0x08	PORTC	data register port C
0x0029	0x09	PIND	input pins port D
0x002A	0x0A	DDRD	data direction register port D
0x002B	0x0B	PORTD	data register port D
.	.	.	.
.	.	.	.
.	.	.	.
0x005D	0x3D	SPL	stack pointer (low byte)
0x005E	0x3E	SPH	stack pointer (high byte)
0x005F	0x3F	SREG	status register (see Table 10.1)
0x0060	–	WDTCSR	watchdog timer control register
.	.	.	.
.	.	.	.
.	.	.	.
0x0078	–	ADCL	analog-to-digital conv. register (low byte)
0x0079	–	ADCH	analog-to-digital conv. register (high byte)
.	.	.	.
.	.	.	.
.	.	.	.
0x00FF	–	–	reserved

Memory Map

The memory model can be shown visually in a diagram known as a *memory map*. Figure 10.2 shows the memory map for the AVR microcontroller used in the Arduino Uno.
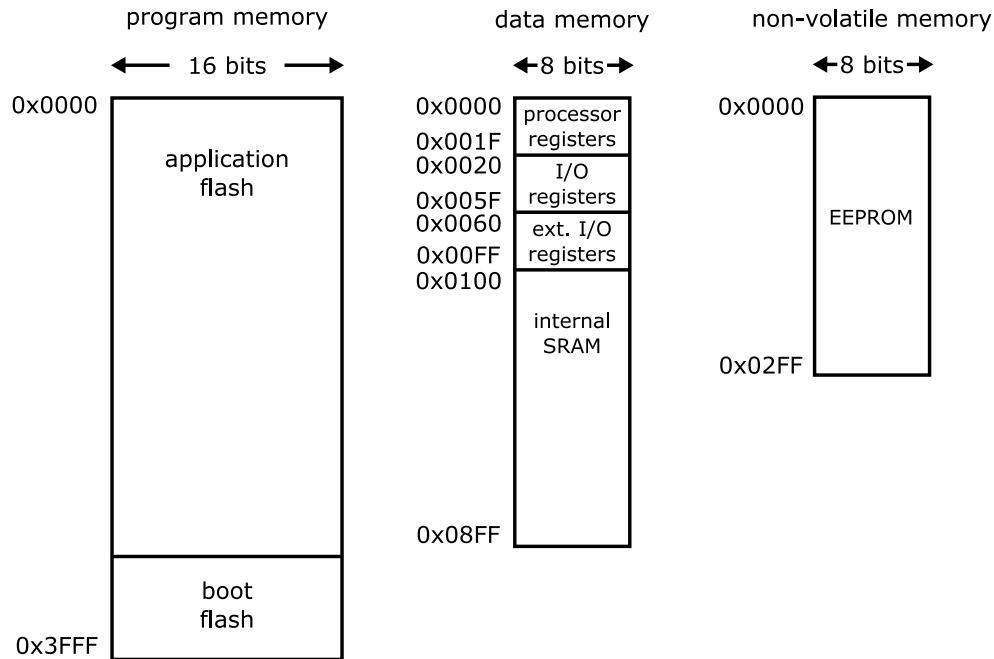
Figure 10.2: AVR microcontroller memory map.

There is a separate rectangle for each memory subsystem, the program memory, data memory, and non-volatile memory. The width (in bits) of each memory subsystem is shown across the top of the rectangle that represents that memory. Addresses (starting at the top and increasing as you look down) are indicated immediately to the left of the rectangle, and different regions within the memory are indicated within the rectangle.

10.2.3 Instruction Set

The third component of the instruction set architecture, after the register file and the memory model, is the actual instructions themselves. These are the specific instructions that can be directly executed by the processor, the machine language of the AVR family of microcontrollers.

The AVR family is what is known as a *2-address, load/store machine.* As such, most instructions operate to and from the register file, rather than interact with memory. The "2-address" label means that instructions reference 2 registers, and one of the registers acts both as a source of data and the destination for the result. The "load/store" label means that explicit load and store instructions move data back and forth between memory and the register file and the operations happen on data in the register file.

Classes of Instructions

We will separately consider instructions as members of several classes, or groups: arithmetic operations, logical operations, control flow, data movement, and system operations.

The following syntactic conventions will be used as part of the descriptions of individual instructions and the addressing modes that follow:

- `Rd` – destination register (one of `r0` to `r31`)

- `Rs` – source register (one of `r0` to `r31`)

- `k` – constant

- `X, Y, Z` – index register (`X` is `r27:r26`, `Y` is `r29:r28`, `Z` is `r31:r30`)

The arithmetic operations include addition, subtraction, increment, decrement, complement, and multiplication (but not division). Let us examine the `add` instruction in some detail. The syntax for a register to register addition is as follows:

```
add Rd,Rs
```

The operation that gets performed is

$$Rd \leftarrow Rd + Rs$$

in which the initial value in `Rd` is added to the value in `Rs` and the result is stored as the new value of `Rd`.

Also, the individual bits of the status register, `SREG`, are set or cleared based upon the value of the result. In particular, six of the eight bits of `SREG` will be altered based on the results of the add instruction. Table 10.3 shows how they are effected. The remaining two bits in `SREG` are not changed by the instruction.

Table 10.3: `SREG` bits effected by `add` instruction.

Label	Name	Definition
H	half carry	set if carry from bit 3
S	sign bit	$N \oplus V$, for signed tests
V	signed overflow	set if two's complement overflow
N	negative flag	equal to most significant bit of the result
Z	zero flag	set if result is 0x00
C	carry flag	set if carry from bit 7 (msb)

Similar to the `add` instruction, the `adc` (add with carry) instruction is used to perform additions on data elements that are larger than 8 bits. The syntax is

        adc Rd,Rs

and the operation that gets performed is

$$Rd \leftarrow Rd + Rs + C.$$

The value in `Rd` is added to the value in `Rs`, plus the carry bit, `C`, of the status register, `SREG`. The result is stored in `Rd`. If we wanted to add the 16-bit value in the register pair `r7:r6` to the 16-bit value in the register pair `r9:r8`, the code would be as follows:

        add r6,r8
        adc r7,r9

in which the `add` instruction sums the lower order bits in `r6` and `r8`, with any carry out being placed in the `C` bit of `SREG`. The subsequent `adc` instruction sums the higher order bits in `r7` and `r9`, including any carry from the lower order bits' sum.

Some instructions require only one operand. For example,

        inc r17

increments (adds one to) register `r17`.

The logical operations include AND, OR, NOT, exclusive-OR, shift, and set/clear bits. Expressing logical operation instructions is very similar to arithmetic operations. Consider the logical AND operation. The syntax is

        and Rd,Rs

while the operation that gets performed is

$$\texttt{Rd} \leftarrow \texttt{Rd} \wedge \texttt{Rs}$$

where each bit of `Rd` is combined with each bit of `Rs` using the logical AND operation with the result stored in `Rd`.

The normal instruction flow is once an instruction has completed its execution, the next instruction in program memory is fetched. Control flow instructions are those that have the potential to change this normal flow. These include unconditional control flow operations, such as jumps, subroutine call, and return, as well as conditional control flow operations, often called *branch* instructions.

Branch instructions will either branch or not branch, based on a Boolean condition that is tested as part of the instruction. Most branch instructions are conditional on one or more bits in the status register, `SREG`. For example, the `brne` instruction (branch if not equal) will branch if the `Z` bit of `SREG` is clear. This instruction might come immediately after a compare instruction, e.g., the instruction sequence

```
cp   r3,r22
brne loop
```

first compares register `r3` with `r22`, which will set the `Z` bit of `SREG` if $\texttt{r3} - \texttt{r22}$ is equal to zero (i.e., if `r3` and `r22` are equal). The conditional branch instruction then branches to the program location labeled `loop` if the two registers are not equal to one another (i.e., the `Z` bit is *not* set).

Data movement instructions includes register to register transfers, load and store instructions that move data back and forth between registers and memory, and I/O instructions that move data to/from peripheral devices.. The simplest data movement instruction copies data from one register to another (leaving the source register unaltered). E.g.,

```
mov r23,r22
```

copies the contents of `r22` into `r23`. Note, `mov` might make you think of the word "move"; however, it leaves the source register unchanged, so "copy" is a better way to think about what the instruction actually does. We can load a constant value into a register using the load immediate instruction:

```
ldi Rd,k
```

which loads the constant value `k` into register `Rd`.

When a load instruction is reading a value from memory, there are a number of methods that can be used to specify the address in memory to be read. The simplest of these is to directly specify the address as part of the instruction, with the load direct from data space instruction:

```
lds Rd,k
```

which accesses the memory at address `k` and copies the contents of memory at address `k` into register `Rd`. Other methods for specifying the address to be accessed are called *addressing modes* and are described below.

Similar to the load direct from data space instruction is the store direct to data space instruction:

```
sts k,Rs
```

which copies the data in register `Rs` and stores it in the memory location at address `k`.

While it is possible to access all the peripheral devices via the memory system (see memory map in Figure 10.2), there are also dedicated instructions for accessing the I/O registers in less time (and smaller instruction size) than load and store instructions. The `in` instruction supports reading from an I/O peripheral:

```
in Rd,k
```

reads data from I/O register `k` ($0 \leq k \leq 63$) into general purpose register `Rd`, and the `out` instruction writes to an I/O peripheral:

```
out k,Rs
```

sends the contents of general purpose register `Rs` to I/O port `k`.

Finally, there are a number of system operations that don't conveniently fit into any of the categories above. For example, the `nop` is "no operation" (it is an instruction that has no effect other than to take time to execute) and the `wdr` instruction is the watchdog timer reset instruction.

Addressing Modes

Instructions that operate on data must first identify the input data to the operation and identify the location in which to store the result. Generally,

techniques used to specify either data source or destination are called *addressing modes*, and the AVR has a number of addressing modes which we will describe below.

Unfortunately, there is great disparity in the naming of addressing modes, so we will use the commonly used name in the description, but will also include the specific name that Atmel uses in its documentation for the AVR microcontroller family.

The simplest addressing mode is *immediate*. Here, the value to be operated on is included as part of the instruction itself. As an example, if we have a value in register `r9` and we wish to subtract 7 from that value, the 7 is part of the instruction, e.g.,

```
subi r9,7
```

in which the operation to be performed is as follows:

$$r9 \leftarrow r9 - 7$$

and the fact that the addressing mode is immediate is conveyed as part of the instruction, subtract immediate, `subi`.

The most commonly used addressing mode is *register* addressing . (Atmel calls this "register direct.") In this case, the source (or destination) of data is one of the 32 general purpose registers. In the example above, the 7 is immediate addressing and the register `r9` is register addressing. In this case, `r9` is both a source and destination. The majority of operations to be performed on data use the register addressing mode, enabling efficient data movement from the register file to the ALU and back to a register.

The first addressing mode we will cover that accesses memory is *direct* addressing. (Atmel calls this "data direct" or "I/O direct.") In direct addressing, the address in memory is directly specified in the instruction. In the case of a load instruction, the source is a memory address, e.g.,

```
lds r12,0x0500
```

loads the value in memory location 0x0500 into register `r12`. In the case of a store instruction, the destination is a memory address, e.g.,

```
sts 0x1000,r12
```

stores the value currently in `r12` into memory location 0x1000. Direct addressing is also appropriate for accessing the I/O registers using the `in` and `out` instructions as well. In this case, the address specified is not a memory address, but rather an I/O address (in the range 0 to 63).

While direct addressing works well when the address to be accessed is known when the program is written, it isn't as useful if the address depends on the program input. Consider the case of an array reference. Which specific address we wish to access depends upon the value of the array index.

To enable the address to be computed at run time, we use the *indirect addressing mode*. (Atmel calls this "data indirect.") Recall that 3 registers pairs (`r27:r26`, `r29:r28`, and `r31:r30`) have alternate names (`X`, `Y`, and `Z`, respectively). The register pairs are sometimes called *index registers*. The indirect addressing mode uses the contents of one of these register pairs as the address of the data to be loaded (read) or stored (written).

For example, the instruction sequence below:

```
ldi r26,0x00
ldi r27,0x05
ld  r12,X
```

first puts 0x0500 into the index register `X` (the least significant byte into `r26` and the most significant byte into `r27`) and then loads the value at memory location 0x0500 into register `r12` (using the `ld` instruction). Similarly, the instruction sequence

```
ldi r28,0x00
ldi r29,0x10
st  Y,r12
```

puts 0x1000 into the index register `Y` and then stores the value initially in register `r12` into memory location 0x1000.

The indirect addressing mode can be extended to also alter the index register used to access memory. In the *post-increment* mode, the index register is incremented after being used to access memory. In the *pre-decrement* mode, the index register is decremented prior to being used to access memory. The syntax for a post-increment indirect access is

```
ld Rd,Y+
```

for a load, and the syntax for a pre-decrement indirect access is

```
st -Z,Rs
```

for a store.

There are a few specialized addressing modes for program memory access; however, we will not discuss them here.

10.2.4 Operating Modes

Complex processors often have different *operating modes*, in which different subsets of the instruction set are enabled or not enabled. For example, the control mechanisms for the virtual memory subsystem on an Intel processor in a PC cannot be accessed by a regular user's program.

The Atmel AVR family does not effectively have operating modes, so we can avoid the subject, other than to include it in the list of what constitutes a processor's ISA.

11 Assembly Language

In the examples of instructions provided in Chapter 10, we actually cheated a bit. In the real machine, instructions are binary values (typically 16 bits in length) that are stored in the program memory. The examples we provided are actually written in a more human-readable form called *assembly language*.

In this chapter, we will clarify the distinction between assembly language and machine language, as well as discuss the options available to the assembly language programmer that do not get translated (directly) into machine language. Once a program has been expressed in assembly language, a tool (called the *assembler*) translates the assembly language source code into machine language (often called *object code*). We will also spend some time discussing how to author good assembly language code, including how to mix assembly language and high-level language code in one sketch.

11.1 Machine Instructions

The individual instructions that are stored in the program memory and are directly executed by the processor constitute the *machine language* of the processor. Machine language instructions are encoded in binary, with the bit pattern both specifying the operation itself (i.e., the opcode) and the operands (i.e., source and destination of the data).

For example, the encoding for several representative instructions is given in Table 11.1. Some of the table entries are for specific operands, and others show how to encode the operands in the instruction word. When encoding operands, the general purpose registers are numbered 0 to 31, requiring 5 bits to specify. Register `Rd` is specified using the 5-bit value $d_4d_3d_2d_1d_0$ and register `Rs` is specified using the 5-bit value $s_4s_3s_2s_1s_0$. Constants are specified in a similar way, with the distinction that different constants have different numbers of bits allocated in the instruction format. Almost all instructions are 16 bits, with a few having an additional 16-bit word (that immediately

Table 11.1: Machine language encoding of several instructions. In the table, d_i is bit i of Rd, s_i is bit i of Rs, and k_i is bit i of k.

Instruction	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
nop	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
wdr	1	0	0	1	0	1	0	1	1	0	1	0	1	0	0	0
inc Rd	1	0	0	1	0	1	0	d_4	d_3	d_2	d_1	d_0	0	0	1	1
inc r12	1	0	0	1	0	1	0	0	1	1	0	0	0	0	1	1
ld Rd,X	1	0	0	1	0	0	0	d_4	d_3	d_2	d_1	d_0	1	1	0	0
ld r6,X	1	0	0	1	0	0	0	0	0	1	1	0	1	1	0	0
mov Rd,Rs	0	0	1	0	1	1	s_4	d_4	d_3	d_2	d_1	d_0	s_3	s_2	s_1	s_0
mov r3,r1	0	0	1	0	1	1	0	0	0	0	1	1	0	0	0	1
add Rd,Rs	0	0	0	0	1	1	s_4	d_4	d_3	d_2	d_1	d_0	s_3	s_2	s_1	s_0
add r7,r4	0	0	0	0	1	1	0	0	0	1	1	1	0	1	0	0
subi Rd,k	0	1	0	1	k_7	k_6	k_5	k_4	d_3	d_2	d_1	d_0	k_3	k_2	k_1	k_0
subi r21,2	0	1	0	1	0	0	0	0	0	1	0	1	0	0	1	0
lds Rd,k	1	0	0	1	0	0	0	d_4	d_3	d_2	d_1	d_0	0	0	0	0
	k_{15}	k_{14}	k_{13}	k_{12}	k_{11}	k_{10}	k_9	k_8	k_7	k_6	k_5	k_4	k_3	k_2	k_1	k_0
lds r8,0x100	1	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0

follows in program memory).

Note an interesting feature of the subi instruction, subtract immediate. It only has 4 bits to specify Rd, the destination register. Instead of being available to use on all 32 general purpose registers, the subi instruction can only operate on registers r16 to r31, and the register Rd is specified by the bits $1d_3d_2d_1d_0$.

Restrictions like the one above are fairly common in the machine language of many processors. With only 16 bits to specify the full instruction, the designer of the instruction set often must choose whether to give access to the full range of registers (requiring 5 bits of the 16 available in the instruction word) or limit the range of registers accessible and increase the range of constants that can be specified (in this case, enabling constants of length 8 bits, which can range in value between -128 and $+127$).

11.2 Assembly Language Instructions

In the code examples of Section 10.2.3 and in the left-most column of Table 11.1, we have been using the assembly language expressions for the instructions of interest. The general form for assembly language instructions

is:

```
label: opcode operands ;comments
```

where the `label` in an optional identifier (specifically, an identifier of the address in program memory where the instruction is stored), the `opcode` is the name of the instruction itself, sometimes called an operation code, the `operands` specify the source and/or destination of the data needed to execute the operation, and the `comments` are ignored by the program (they are intended for the human reader).

It is conventional for the semi-colon (`;`) to delimit comments in assembly language. However, in modern assemblers C/C++ style comments (either `//` or `/* */`) are also commonly recognized.

11.3 Labels and Symbols, Constants and Numbers

Whenever the programmer inserts a label into the assembly language source, that declares a symbol (within the assembly process) that can then be used elsewhere in the program. An immediately obvious use of this capability is to provide labels whenever a program might wish to do conditional branching.

Consider the following code snippet:

```
      ldi r3,2   ; r3 <- 2
loop: add r7,r8  ; r7 <- r7 + r8
      dec r3     ; r3--
      brne loop  ; branch if r3 <> 0
```

in which `r3` is initialized to 2, and after `r8` has been added to `r7`, `r3` is decremented by one. The decrement instruction, `dec`, sets the `Z` bit in `SREG` to 1 if the result of the decrement is zero. The first time that `dec` is executed, the result is not zero, so the `brne` instruction (BRanch if Not Equal to zero) will take the branch to `loop`, which has been declared to be the address of the `add` instruction. Once `r3` does equal zero, the `brne` instruction no longer branches, and the code executes the instruction that immediately follows `brne`.

Another observation to make is that when initializing `r3` to the value 2, we did not specify the base. By default, when constants are specified in assembly language, they are in base 10 (as is the case in higher-level languages like C/C++ and Java). There are a variety of ways that alternative bases can be expressed, and the assembly language programmer must check the manual of the specific assembler being used to know which is correct. We will stick

with the same convention used in C/C++ and Java, that a `0x` preceding the numeric constant means that the constant is being expressed in hexadecimal.

11.4 Assembly Language Pseudo-operations

In addition to assembly instructions that translate directly in machine instructions, the assembler also recognizes instructions that in reality are aimed at the assembler itself. These *pseudo-operations* (or *pseudo-ops*) are also sometimes called assembler *directives*.

To distinguish pseudo-ops from regular instructions, it is conventional for them to begin with a period (`.`) as the first character of the operation. For example,

```
.equ portd,0x0b
```

uses the `.equ` pseudo-operation to define the symbol `portd` as equivalent to the expression `0x0b`. That symbol can then be used in place of a numerical constant in a subsequent `in` or `out` instruction, such as

```
in r7,portd  ;read from I/O port 0x0b
```

which reads from the I/O port `0x0b` but references the port using the symbolic name `portd`.

11.4.1 Sections

One of the tasks required of the assembly language programmer is to specify what items are to be assigned to the various memories. On the AVR family, the code goes into the program memory (often called the *text section* or *text segment*), and the data is either in the non-volatile memory (EEPROM) or data memory (SRAM). The SRAM is, in assembly language terms, called the *data section* or *data segment.*

11.4.2 Data Section Pseudo-ops

The data section starts with the following pseudo-op:

```
.data
```

which declares that the assembly language statements to follow are associated with the data memory (SRAM).

138

There are a collection of pseudo-ops that are supported in the data section for reserving space in the data memory. The simplest is `.byte`, which reserves space and gives it an initial value, i.e.,

```
label: .byte expression[,expression]
```

which reserves one byte for each `expression` (the second and subsequent ones are optional) and initializes it with the value of `expression`. The inclusion of the label associates the symbol `label` with the address of the first byte allocated.

We can make `label` into a global symbol (visible to the linker) through the use of the `.global` pseudo-op:

```
.global label
```

however, be careful to remember that the linker does not have any notion of type, so it is only the address of `label` that is available to other files.

We can allocate a C-style string (null-terminated) via the `.asciz` pseudo-operation, e.g.,

```
errstr: .asciz "Error"
```

reserves 6 bytes of data memory and initializes it to the ASCII characters 'E', 'r', 'r', 'o', 'r', and '\0'. The label `errstr` is associated with the first address in the string.

Arbitrary sized chunks of data memory can be allocated with the `.space` pseudo-op, which takes two arguments, the `size` of the chunk of memory to reserve (in bytes) and the `value` to store in each byte. For example,

```
array1: .space 10,0
```

will reserve 10 bytes of memory (initialized to 0) and associate the label `array1` with the first allocated byte.

11.4.3   Text Section Pseudo-ops

The text section starts with the following pseudo-op:

```
.text
```

which declares that the assembly language statements to follow are code (program instructions).

If we wish a label declared in the `.text` section to be available to the linker (i.e., enabling it to be called from another file), we again use the `.global` pseudo-op. The `.equ` directive is also available to use in the `.text` section.

Another commonly used directive is `.include`, which enables another file to be directly inserted at the point of the directive, e.g.,

```
.include "header.h"
```

will insert the contents of the file `header.h` at the current point in the assembly language source.

11.4.4 Macros

With the `.macro` directive, it is possible to declare macros that can do fairly sophisticated symbolic processing at assembly time. We will not describe the general technique for declaring macros; however, we will illustrate the use of two very useful macros that are built-in to the assembler.

When using an 8-bit processor, it is common to manipulate 16-bit values (e.g., addresses) one byte at a time. The assembler provides two macros that are useful in this process, i.e,

```
lo8(value)
```

takes a 16-bit `value` and returns the least significant 8 bits, while

```
hi8(value)
```

takes a 16-bit `value` and returns the most significant 8 bits. These macros can be very useful in address manipulation, especially for array indexing.

For example, if we wish to load the address of `array1` into the `X` index register (recall `X` is a name for the register pair `r27:r26`), we can use the following instruction sequence:

```
ldi r26,lo8(array1)
ldi r27,hi8(array1)
ld  r2,X
```

which loads the value stored at address `array1` into register `r2`.

11.5 Authoring in Assembly Language

It is true that when authoring applications in assembly language, the programmer can do pretty much whatever he or she wants to do, given the only the constraints of the instruction set. That approach, however, is rarely a good idea. The conventions that have come in to common practice using high-level languages (even simple things like `if-then-else` statements and `while` loops) came into being for good reasons. Next, we will examine a number of those conventions, and explore how to author assembly language application code in such a way as to reap the benefits of those conventions.

11.5.1 Accessing Data

When accessing data in a high-level language like C or Java, the programmer declares variables of a given type, and the compiler is responsible for assigning storage locations for those variables. The programmer can then read the data by simply referencing the appropriate variables in expressions.

In assembly language, things happen at a lower level of abstraction. What is directly accessible to the assembly language programmer is the register set and memory model of the AVR microcontroller. What data are to be stored in registers vs. what data are to be stored in memory is the responsibility of the programmer. In addition, each register is only 1 byte (8 bits) in size, and each individual memory location is also only 1 byte wide. A C `int` is 2 bytes (16 bits) in size, so it requires two registers or two memory locations.

Since the AVR is a load/store architecture, accesses to memory are all via load or store instructions. Data manipulations, such as add, subtract, etc., read from registers and write their results to registers. As such, if we want to add the contents of two memory locations, we must first load the data into registers, perform the add operation, and then store the result to memory.

Consider the following code snippet:

```
lds r18,0x0200  ;M[0x0250] <- M[0x0200] + M[0x0210]
lds r19,0x0210
add r18,r19
sts 0x0250,r18
```

its purpose is indicated in the comment on the first line, add the contents of memory location `0x0200` to the contents of memory location `0x0210` and store the result in memory location `0x0250`. It accomplishes this by loading the two values into registers `r18` and `r19`, adding `r19` to `r18` (leaving the result in `r18`) and storing the result in memory location `0x0250`.

141

The above is an example of how an operation that only takes one line of source code in a high-level language actually takes 4 assembly language instructions to accomplish the same thing. And the above example only works for single-byte data (e.g., a `char` in C). What about an `int`, or a `long int`?

Multi-byte Primitive Data Elements

There are two approaches commonly used to store multi-byte primitive data elements in byte-sized locations. Either approach uses adjacent locations for storage (e.g., if in the register set, use a pair of registers such as `r16` and `r17` to store a two-byte value), they differ in which byte gets stored in which location. The AVR microcontroller is what is called a *little-endian* machine. What this means is that the low-order bits of the multi-byte value get stored in the location with the smaller address or label. In our earlier example, if the two-byte value `0x1f32` were being stored in the registers `r16` and `r17` (denoted as register pair `r17:r16`), the low-order byte, `0x32`, gets stored in `r16` and the high-order byte, `0x1f`, gets stored in `r17`.

In the register set, we reference pairs of registers by listing them both (e.g., the `r17:r16` example above). When talking about multi-byte values stored in memory, the convention is to discuss the data value as if it were stored in one address, even though it *actually* consumes more than one memory location. So, a two-byte value that is referred to as being stored at location `0x02f4` actually consumes memory location `0x02f4` and location `0x02f5` (recall that memory addresses are 16 bits long on the AVR microcontroller, it is the memory cells themselves that are 8 bits wide). Since it is a little-endian machine, the low-order byte gets stored in location `0x02f4` and the high-order byte gets stored in location `0x02f5`.

The second approach (which is the opposite of a little-endian machine) is a big-endian machine. In a big-endian machine, the high-order byte of a multi-byte value is stored in the location with the smaller address. The NXP ColdFire processors (originally developed by Motorola) are an example of big-endian machines. The big-endian convention is also known as *network order*, as the high-order byte of multi-byte values are typically sent first in data communications (we will return to this issue in Chapter 12).

On our little-endian AVR microcontroller, if we wish to load a 16-bit value from memory address `0x2f4` into the register pair `r17:r16`, the code would be as follows:

```
lds r16,0x02f4
lds r17,0x02f5
```

and it takes two load instructions, one per byte. We can accomplish the same goal in a slightly more general way by using one of the address registers:

```
ldi r26,0xf4    ;X <- 0x02f4
ldi r27,0x02
ld  r16,X+      ;r17:r16 <- M[02f4]
ld  r17,X
```

which has a number of things going on. First, we are using address register X, which is actually the pair of registers `r27:r26`, so it must be initialized to `0x02f4`, which happens in the first two instructions. Second, take a look at the comment associated with the first two instructions. See what it is doing? It is telling us higher-level information than the literal statements in assembly language. The instruction with the comment, and the instruction immediately below it, serve the high-level purpose of storing the value `0x02f4` in the X address register. Third, the next instruction (after initializing the address register) loads the low-order byte into `r16`. It uses the post-increment addressing mode, so that after address register X has been used to access memory, it is incremented by 1 (so it now has the correct value to access the high-order byte of our two-byte data). Fourth, we finish by loading the high-order byte from location `0x02f5` into register `r17`. Again, the comment associated with the last two instructions describes the high-level intentions of the programmer, specifically, to load the contents of memory at address `0x02f4` into the register pair `r17:r16`.

In addition to two-byte elements (such as `int`), we also want to occasionally deal with four-byte elements (such as `long int` or `float`). The general rule that applies to two-byte values extends just as one would expect for larger values. The least significant byte is stored in the lowest memory address and the most significant byte is stored in the highest address. Figure 11.1 illustrates the memory layout for both little-endian and big-endian machines.

Arrays

While the order in which the individual bytes of a primitive data type are stored in memory depends upon whether a machine is little-endian or big-endian, the order in which array elements are stored has no such dependency. In C, array indices start at `0`, and increment up. The memory locations used to store array elements do exactly the same.

Consider the C string declared as follows:
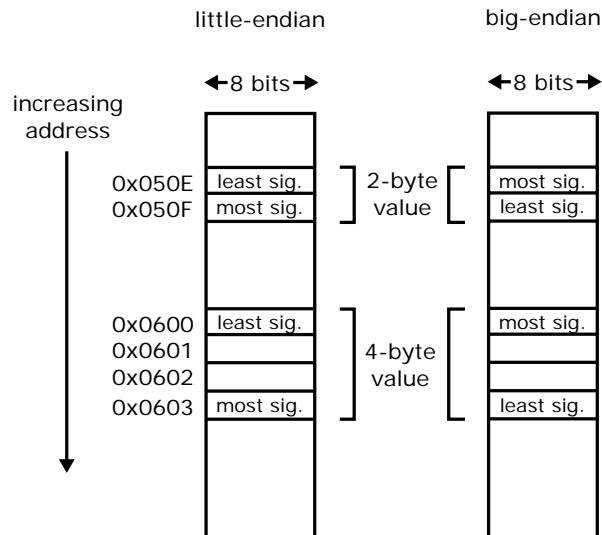
```
char str[5] = "read";
```

143

Figure 11.1: Two-byte and four-byte memory layout for little-endian and big-endian machines. The 2-byte value is located at address `050e` and the 4-byte value is at address `0x0600`.

Table 11.2: Memory layout of array `str`.

C reference	Address	Value
`str[0]`	`str + 0`	`'r'`
`str[1]`	`str + 1`	`'e'`
`str[2]`	`str + 2`	`'a'`
`str[3]`	`str + 3`	`'d'`
`str[4]`	`str + 4`	`'\0'`

it will have 5 array elements, and the memory layout will be as shown in Table 11.2.

In assembly language, this array would be declared (in the data segment) and initialized as follows:

```
        .data
str: .asciz "read"
```

which essentially does close to the same thing as the C declaration above. It allocates 5 bytes in memory, initializes those bytes to the characters in the string `"read"` (including the terminating NULL character), and declares the symbol `str` to refer to the address of the first element. The primary

144

difference is that C understands the type of `str` to be an array of `char`s, while in assembly language, the symbol `str` merely represents the address of the memory location, it has no type information.

If we want to write assembly language code to implement the following C statement,

```
str[3] = 'l';
```

which will edit the string from `"read"` to `"real"`, we again use an address register. This time we'll use address register `Z`, just to be different.

The assembly language code is shown in Figure 11.2, in which the first two instructions put the address of the array `str` into the address register `Z` (taking advantage of the macros `hi8()` and `lo8()` described earlier in the chapter), the third instruction changes `Z` to be the address of the 4th element of `str` (by adding 3, remember that the first element is designated by index 0), the next to last instruction puts the ASCII code for the letter l into `r16`, and the last instruction stores the letter into `str[3]`.

```
ldi  r30,lo8(str)  ;Z <- &str[3]
ldi  r31,hi8(str)
adiw r31:r30,3
ldi  r16,'l'       ;r16 <- 'l'
st   Z,r16         ;str[3] <- r16
```

Figure 11.2: Assembly language array access.

If the array is an array of multi-byte values (e.g., an array of `int`s), then the order of elements in memory is the same (start at index `0` and increase); however, each element now consumes more than one byte of storage. If, for example, the elements are each two bytes, then the address increase for each element must be two. The element itself is stored with its normal byte ordering (least significant byte first for the AVR microcontroller).

Consider the 5-element integer array `item`, which would be declared in C as follows:

```
int item[5];
```

and declared in assembly language as:

```
 item: .space 10,0
```

Table 11.3: Memory layout of `item` array.

C reference	Address	Contents
item[0]	item + 0	least significant byte of item[0]
	item + 1	most significant byte of item[0]
item[1]	item + 2	least significant byte of item[1]
	item + 3	most significant byte of item[1]
item[2]	item + 4	least significant byte of item[2]
	item + 5	most significant byte of item[2]
item[3]	item + 6	least significant byte of item[3]
	item + 7	most significant byte of item[3]
item[4]	item + 8	least significant byte of item[4]
	item + 9	most significant byte of item[4]

each reserving 10 bytes of storage. The memory layout of the array is shown in Table 11.3.

Accessing an individual element is pretty much the same as before, with the added twist that we must account for the fact that each element is two bytes long, and therefore consumes two addresses. Again using the `Z` address register, we can read `item[2]` into register `r17:r16` using the assembly language code shown in Figure 11.3.

```
ldi  r30,lo8(item)  ;Z <- &item[2]
ldi  r31,hi8(item)
adiw r31:r30,2
adiw r31:r30,2
ld   r16,Z+          ;r17:r16 <- item[2]
ld   r17,Z
```

Figure 11.3: Assembly language array access for 2-byte elements.

As before, we load the address of `item` into `Z`. Next, we add the index offset, but we do it twice since we want 2 times the offset (since each element is 2 bytes long). Finally, we read the array element (both bytes).

11.5.2 Control Flow Templates

There are many reasons people use high-level languages all the time, and the ability to easily express control flow is but one of those reasons. However, it

is true that anything you can express in a high-level language, you can also express in assembly language, it just takes more instructions to do so. We will next consider how to implement several examples of control flow in assembly language.

Control Flow: `if-then`

The most straightforward example of control flow in a high-level language is the `if-then` statement. Consider the C code of Figure 11.4(a), where `a`, `b`, and `c` are all assumed to be one-byte data types (e.g., `char` or `byte`). We can write the same thing in assembly language as shown in Figure 11.4(b), where we have made the arbitrary decision to use `r16` to hold `a`, `r17` to hold `b`, and `r18` to hold `c`. Also, the comments roughly indicate where, in the assembly language code, the high-level language constructs are being implemented.

```
if (a == b) {
    c++;
}
```

(a) C code.

```
      lds r16,a   ; if (a == b) {
      lds r17,b
      cp  r16,r17
      brne end_if
      lds r18,c   ;      c++;
      inc r18
      sts c,r18
end_if:           ; }
```

(b) Assembly language code.

Figure 11.4: Assembly language `if-then`.

There are a few things to note about this code. First, we used the `cp` (ComPare) instruction to compare `a` and `b` (actually, we compare the values that were initially in memory locations `a` and `b`, which are by now located in registers `r16` and `r17`). What the `cp` instruction does is subtract the two values, but throws away the result, and only retains the conditions codes (the bits of `SREG`). In our case, we are interested in the Z bit (the zero bit) of `SREG`, because we want to test whether or not the two values are equal (in which case

subtracting them gives a result of zero). After the `cp` instruction executes, we have the convenient circumstance that if the two values are equal, the `Z` bit is set (has the value `1`), and if the two values are not equal, the `Z` bit is clear (has the value `0`).

Second, we use a conditional branch instruction, `brne` (BRanch if Not Equal), that is actually the opposite of the `if` test in the C code. We branch because we want to skip over the `then` clause and not execute it.

Third (and this is not required, but convention), the target of our conditional branch, `end_if`, is on a line by itself, rather than with code that follows the `if`. This is simply to make it easier to edit the logic later, the `end_if` label will still point to the next assembly language instruction, even if it is not on the line with the label itself.

As the above example illustrates, the general approach is the following four steps.

1. Write code to test the conditional (`if`-clause), leaving the result of the test in a place where a conditional branch instruction can evaluate it.

2. Use the conditional branch instruction that yields the negation of the `if`-test, branching to a label at the end of the code snippet.

3. Write code that implements the `then`-clause.

4. Finish with the target label of the conditional branch instruction.

Remember that all labels in assembly language must be unique, so it is common to see assembly language programmers use labels that have numbers in them, just to make them unique (e.g., `end_if12` if this was the 12th `end_if` label he/she was creating). There is, of course, nothing magic about the `end_if` either, you also see fairly unimaginative labels like `l12` often as well.

Control Flow: `if-then-else`

An only slightly more complicated construct is the `if-then-else`, which we will consider next. Starting with the C code of Figure 11.5(a), a simple extension of the earlier example, the equivalent assembly language can be written as shown in Figure 11.5(b), in which the first seven instructions are the same as Figure 11.4(b), other than the fact that the destination of the conditional branch instruction is the `else` label instead of the `end_if` label. At the end of the code for the `then`-clause, a `jmp` (JuMP) instruction sends us to the `end_if` label. As you would likely expect, the code for the `else`-clause follows the `else` label.

```
    if (a == b) {
        c++;
    }
    else {
        c--;
    }
```

(a) C code.

```
    lds r16,a    ; if (a == b) {
    lds r17,b
    cp  r16,r17
    brne else
    lds r18,c    ;      c++;
    inc r18
    sts c,r18
    jmp end_if  ; }
else:            ; else {
    lds r18,c    ;      c--;
    dec r18
    sts c,r18    ; }
end_if:
```

(b) Assembly language code.

Figure 11.5: Assembly language `if-then-else`.

Starting from the code above, one might recognize that a few changes are possible which make it (1) a bit smaller, and (2) a bit harder to follow. Notice that two instructions, the `lds r18,c` and the `sts c,r18`, are present in both the `then`-clause and the `else`-clause. We can move the first one above the test code and the second to the end of the code snippet to get the alternate code shown in Figure 11.6, which is two instructions smaller, yet does the same thing.

Note, though, that the comments don't line up quite as well, associating bits of assembly language with C. This code, while smaller, is a tad bit harder to follow, logic-wise. Not much, but a little. These kinds of transformations, moving loads forward and stores later, are commonly performed by compilers.

```
    lds r16,a    ; if (a == b) {
    lds r17,b
    lds r18,c
    cp  r16,r17
    brne else
    inc r18      ;      c++;
    jmp end_if   ; }
else:            ; else {
    dec r18      ;      c--;
end_if:
    sts c,r18    ; }
```

Figure 11.6: Alternative assembly language `if-then-else`.

Control Flow: while

For our example `while`, we will be a bit more generic and start with the C code
of Figure 11.7(a), where `done` is assumed to be a one-byte flag (e.g., of type
`boolean`) and the code inside the `while` eventually changes `done` from 0 to 1.

```
    while (!done) {
        // do something
    }
```

(a) C code.

```
while:
    lds r16,done    ; while (!done) {
    cpi r16,0
    brne end_while
    ...             ;      // do something
    jmp while
end_while:          ; }
```

(b) Assembly language code.

Figure 11.7: Assembly language `while`.

Assembly language that does this can be written as illustrated in Fig-
ure 11.7(b), in which the `while` test is performed at the top, the body follows,

and at the bottom of the body there is an unconditional `jmp` back to the `while` test. When the `while` test fails the code jumps to the `end_while` label.

Control Flow: `for`

At this point we are going to stop giving explicit examples, and make a different point. A `for` loop can always be written in terms of a `while` loop, so if we had the following C code:

```
for (i = 0; i < N; i++) {
    // do something
}
```

a perfectly good way to author this in assembly language is to first transform it into the equivalent `while` loop, as below:

```
i = 0;
while (i < N) {
    // do something
    i++;
}
```

and then convert to assembly language.

11.6   Interfacing with C

Frequently, we would like to write code that is partially in assembly language and partially in C. The most straightforward way to do this is to author assembly language routines and C routines separately. Then, as long as certain conventions are followed when authoring the assembly language routines, C routines can call assembly language routines and assembly language routines can call C routines. The text below describes those conventions (i.e., what you need to do as an assembly language programmer to ensure compatibility with C).

11.6.1   Calling Conventions

When the C compiler translates C code into machine language, it has various conventions that it follows, so as to ensure that C routines are compatible with one another. As we write assembly language routines, as long as we follow those conventions, our assembly language routines will then be compatible with C.

The primary three things we need to consider are: register usage, parameter passing, and function return values.

Register Usage

When authoring programs exclusively in assembly language, we can, for the most part, use whatever registers we want for almost any purpose. Yes, we have to follow the requirements of the instruction set, e.g., only certain register pairs can be used as index registers for indirect addressing, but other than that, register usage is up to the programmer.

This is no longer the case if we wish our assembly language code to be compatible with C. When one C routine invokes another C routine, the calling routine is known as the *caller* and the invoked routine is known as the *callee.* Each routine has different responsibilities and obligations with respect to register usage. The C compiler divides up the registers into three groups: fixed registers, caller-save registers, and callee-save registers.

The *fixed registers* are `r0` and `r1`. Register `r0` is a temporary register that can be used for intermediate results, but can be overwritten by C code, so don't assume it will retain its value if a C routine is called from assembly language.

Register `r1` is assumed to always be zero (`0`) by the C compiler (i.e., it is set to `0` before `setup()` and then never written to after that). As a result, anytime you wish to have handy access to the value `0`, read from `r1`. Don't write to `r1` unless you are writing `0` (and why you would want to do that, I have know idea).

The *caller-save registers* are `r18-27` and `r30-r31`. Caller-save essentially means that if a routine is about to call another routine (i.e., it is the caller), it cannot assume that these registers will retain their values. It is the responsibility of the *caller* to *save* the value of the registers, somewhere other than the register itself.

The most common place to save the contents of the registers is on the system stack. For example, if the caller has important data that must be retained in registers `r26` and `r27`, they can be saved by pushing them onto the stack using the `push` instruction and then invoking the called routine:

```
push r26            ; save r26 and r27 on the stack
push r27
call some_C_routine
pop  r27            ; restore register values
pop  r26
```

after which the register values can be restored from the stack using the `pop` instruction. Note that since the stack operates in a "last-in, first-out" manner, the registers must be restored in the opposite order that they were saved.

From the perspective of the called routine, it can change caller-saved registers at will, since it can be assured that the calling routine has saved their values.

The *callee-save registers* are `r2-17` and `r28-r29`. In this case, it is the callee (the called routine) that must save the contents of any registers that it uses, because the calling routine assumes they are not changed.

From the perspective of the calling routine, if it has data stored in callee-saved registers, it can be assured that the called routine will save those values and they will persist across the routine's invocation.

Passing Parameters

When passing parameters, how the calling routine places arguments in registers depends upon the size of the arguments. Arguments are assigned to registers starting with register pair `r25:r24` for the first argument (assuming it is either 8 bits or 16 bits in size), then register pair `r23:r22` for the second argument, etc., down to register pair `r9:r8`. Arguments that are 8 bits long only use the even-numbered register of a register pair (e.g., `r24`, `r22`, etc.), leaving the odd-numbered registers unused. Arguments that are 32 bits long consume two consecutive register pairs (four consecutive registers), moving subsequent arguments to lower-numbered registers.

Function Return Values

As with parameters, the mechanism for returning values from function calls also depends upon the size of the return value. Table 11.4 indicates the registers used to provide function return values of various sizes. If the return value is 8 bits long, register `r25` is either zero-extended or sign-extended. Note that all of the registers that are used for return values are all caller-save registers. If the calling routine wishes to retain the value in those registers across the function invocation, it must save the values prior to calling the function (which is the callee in this case).

Now that we are following C conventions for register usage, parameter passing, and function return values, we can call C routines from assembly language as well as call assembly language routines from C. We will describe each of these next.

Table 11.4: Registers for function return values.

Return value data size (bits)	Registers (most sig. to least sig.)
8	r24
16	r25:r24
32	r25-r22

11.6.2 Calling C Routines from Assembly Language

When authoring code in assembly language, it is often the case that we would like to invoke a routine in C (maybe one that already exists), so that we don't need to rewrite it in assembly. An example of this might be calling a routine available in the C library, such as `digitalWrite()` or `digitalRead()`.

Consider the assembly language code in Figure 11.8. It does the same thing as the `loop()` code of Figure 2.1. While it uses registers `r22` through `r25` to pass parameters to the C routines that it calls, it doesn't save their values (even though they are caller-save registers) because it doesn't need those values retained after the call to each C routine.

```
      .global foo
foo:  lds r24,doPin       ; digitalWrite(doPin,HIGH)
      lds r25,doPin+1
      ldi r22,1
      mov r23,r1
      call digitalWrite
      ldi r24,lo(500)     ; delay(500)
      ldi r25,hi(500)
      call delay
      lds r24,doPin       ; digitalWrite(doPin,LOW)
      lds r25,doPin+1
      ldi r22,0
      mov r23,r1
      call digitalWrite
      ldi r24,lo(500)     ; delay(500)
      ldi r25,hi(500)
      call delay
      ret
```

Figure 11.8: Calling C from assembly language.

The C routines we called in this example didn't return a value. However, if they did, the return value would be in registers `r25:r24` (for values of 16 bits or less).

11.6.3 Calling Assembly Language Routines from C

The assembly language routine of Figure 11.8 is already in appropriate form so that we can call it directly from C. This is because: (1) we included the pseudo-op `.global` to indicate that the label `foo` needs to be known to the linker (part of the compiler), (2) we provided the label `foo` to indicate the first instruction to execute, (3) we didn't alter any callee-save registers, and (4) we included a `ret` (RETurn) instruction at the end.

Figure 11.9 illustrates a sketch that essentially does the same thing as Figure 2.1, only calling the assembly language routine `foo()` to do the work within `loop()`. Note we require an `extern` statement at the top to inform the linker that the routine `foo()` is found elsewhere (not in the same file as this sketch).

```
extern "C" {                    // tell linker about foo()
    void foo(void);
}


const int doPin = 17;      // digital output pin is 17


void setup() {
  pinMode(doPin, OUTPUT);  // set pin to digital output
}


void loop() {
  foo();                        // call assembly language foo
}
```

Figure 11.9: Calling assembly language from C.

12 Computer to Computer Communications

It has been said in the past that a "smart" device was one that was capable of some level of computation. That definition is dated. Raj Jain, one of the early developers of congestion control protocols for networks, describes a "smart" device as one that is connected. Connected to other devices, connected to the Internet, connected to the world. In this chapter, we will investigate techniques for communicating between computers, both microcontrollers and desktop computers. We'll address byte stream concepts, as well as communication protocols that enable higher-level abstract communications.

One of the fundamental notions that has to be considered when dealing with computer to computer communications is that each computer (whether it be a simple microcontroller or a sophisticated desktop or server machine) is executing its own program, and both programs are running at the same time. In effect, there is *concurrency* present in the complete system.

This situation is different than everything we have seen so far. Up to this point, an individual program was running on the microcontroller, and while the actions that the computer takes are fast, only one thing is happening at a time, and they have a strict ordering. This is no longer the case. When two actions happen on distinct computers, it is entirely possible that we don't know (or can't know) precisely which one happened first.

The presence of concurrency in the system often brings with it more complicated reasoning about the correctness of the system as a whole. Here, we will address these issues by constraining the scope of designs that we consider, staying within the realm of operations that are relatively easy to reason about. A word of caution, however, as arbitrary concurrent techniques can be very difficult to understand.

In addition, communications between two different computers also encompasses the likelihood that the two computers are not just two copies of the

157

same type of computer, but are in reality two different types of computer as well. To help us understand and deal with these issues, in this chapter we will assume that our Arduino microcontroller is communicating with a desktop machine (or maybe a laptop machine) that is running a Java program. For those unfamiliar with Java, Appendix A compares the Java language with the Arduino C language.

12.1 Stream Concepts

We will organize our description of computer to computer communications in terms of an abstract concept called a *stream*. A *stream* is an arbitrary sequence of bytes, delivered from one computing entity to another. A stream has a source, which generates the sequence of bytes to be delivered, and a destination, which receives the sequence of bytes.

One of the advantages of the abstract concept of a stream is that one can author code to serve as a stream source without knowing the stream destination, and one can author code to serve as a stream destination without having to know the stream source. This improves the composability of software that uses the stream abstraction.

Abstract streams follow a few simple rules:

1. The data elements explicitly delivered via streams are *bytes*. Any other data type must be built on top of the byte stream.

2. The data bytes are delivered in order. That is, if the source sends byte A followed by byte B, the destination will receive byte A ahead of byte B.

3. Some streams guarantee reliable delivery. That is, if the source sends byte A, the destination will eventually receive byte A. This is not always the case, however, as some streams do not guarantee reliable delivery. In this case, a byte sent by the source might or might not eventually be received by the destination, or it might be delivered but have the wrong value (e.g, one or more bits within the byte might have been altered).

The `Serial` class is an example of a stream on the microcontroller. We have used `Serial.print()` and `Serial.println()` in a number of previous chapters as a mechanism for writing output to the serial port that connects the microcontroller to the IDE executing on a desktop machine. When running the `Serial Monitor` in the IDE, code executing on the microcontroller is serving as the stream source, and the `Serial Monitor` is serving as the stream destination.

158

12.2 Delivery of Streams

One of the benefits of the stream abstraction is that the endpoints of a stream do not need to know about one another (i.e, the source's implementation is independent of the destination and the destination's implementation is independent of the source). In a similar way, another benefit is that neither endpoint needs to be aware of the physical mechanism used to deliver the bytes from source to destination.

Possible delivery mechanisms include copying bytes in the memory of a processor (e.g., from one program to another), sending bytes over a local area network (LAN) or the Internet, delivering bytes wirelessly, using a serial port implemented on top of a USB link, or a host of other paths.

We will discuss a number of these delivery mechanisms in turn, next.

12.2.1 Internet

The Internet is composed of a vast number of links, switches, protocols, conventions, and mystery (especially to those who have not studied its design and implementation). For our purposes here, however, we don't need to know hardly anything about the mechanisms used to deliver data either across the room, across the continent, or around the world. By embracing the stream abstraction, we can take advantage of the infrastructure (both hardware and software) that the Internet provides without any obligation to comprehend how it was constructed.

When a stream is setup between two endpoints (and yes, this does require some setup, but like all of our other examples, the heavy lifting will be done by library code provided to us), bytes that are sent by one endpoint will be delivered, in order, to the other endpoint. This sending and receiving of bytes works in both directions.

12.2.2 Serial Ports

The most commonly used stream in the Arduino world is the serial connection between the Arduino and the host laptop or desktop computer used to develop sketches. It is a USB (or Universal Serial Bus) link that, with appropriate drivers on the host, appears as a serial port to the software on either end of the USB cable.

This physical link is used for a pair of purposes. First, it is used to download software from the IDE running on the host into program memory on the

159

Arduino. Second, it is used to communicate (under software control) between the Arduino microcontroller and the host.

12.2.3 Other Streams

One of the really nice things about the stream abstraction is that it can be used in any number of circumstances. For example, one can consider the file system to be a stream. The sender (writing to the file system) is delivering data to the stream at one time, and the receiver (reading from the file system) is retrieving data from the stream at a later time.

This abstraction isn't perfect, obviously, as it is possible in a file system to do random reads. However, for the most typical file system accesses, it works just fine.

12.3 Protocols

If Bob and Alice are near one another (within earshot) and Bob says, "Knock, knock!", we can be very confident in what Alice will reply. Throughout the English-speaking world, Alice will say, "Who's there?" After this initial interaction, Bob will continue telling one of an unknowable number of *knock knock jokes*.

OK, from the point of view of computer communications, what is happening here? Our contention is that both the joke teller and the joke recipient are conforming to a *protocol*, an agreement between the two parties that guides their interaction.

In the context of a stream that connects two computers, the protocol describes the agreement between the two parties to the communication as to how they are to interact with one another. Whose turn is it to transmit? Can they both transmit simultaneously? How does the recipient interpret the sequence of bytes it receives? Below, we will describe an example protocol that enables the communication of a number of different quantities between one computer and another.

12.3.1 Byte Delivery

The stream abstraction describe above essentially provides a sequence of individual bytes, reliably delivered, in order, from the source to the destination. In some circumstances, reliability of the byte delivery is not assured, and it is the responsibility of the higher-level protocol to deal with that issue.

160

In what follows, we will take a middle ground position on reliability, and assume that a byte sent by the source might or might not make it to the destination, but dropped bytes (as they are called) are relatively infrequent. We will also assume that any bytes that do get delivered are correct (i.e., not altered in transit).

In Sections 12.4 and 12.5 below, we discuss the mechanisms available to send and receive individual bytes.

12.3.2 Delivering Larger Data Items

Clearly, if we have the ability to send and/or receive individual bytes, to deliver larger data items it is necessary to use more than one byte for each larger data item. It is also important that both the source and the destination use the same convention for sending and receiving multi-byte data items.

Integers and Other Primitive Types

The convention for sending primitive types (integers, floats, and the like) is to send the bytes in order from most significant byte to least significant byte. This convention is sometimes referred to as "network order." Always sending in network order ensures that the endpoints (either sender or receiver) don't need to know the endianness of the other endpoint.

This convention, however, does not address the need for both endpoints to know the size of the primitive data type. For example, an integer data type on the Arduino platform is 2 bytes, while a Java integer is 4 bytes.

Arrays of primitive types are typically ordered from the lowest index to the highest index. As with the primitive data types, this doesn't help the endpoints know the length of the array, which must therefore be communicated via some other mechanism.

Strings

Strings are data types that are frequently represented in noticeably different ways on different machines. For example, a string in C is stored as an array of `char`s with a null termination (i.e., the string's length is represented by a `'\0'`, or null character, after the last valid character of the string. Each character in the string therefore occupies one byte of space (plus the additional byte to store the null termination).

In contrast, a string is Java is stored within a `String` object. In the standard implementation, the `String` class implements the underlying representation of the string as an array of characters, `char[]`, plus a separate

instance variable that retains the length of the array. Null termination is not used. Also, in Java, the `char` data type is 2 bytes long (using UTF-16 encoding of characters).

Given that different languages use different conventions for internal representations of strings, when one wishes to communicate a string from one computer to another it is insufficient to simply send the bytes using the sender's internal representation and expect the receiver to interpret them correctly. What is needed is an encoding of the string that is agnostic to the type of computer or language used by the sender and the receiver.

As described in Chapter 8, UTF-8 is a variable length character encoding mechanism that is widely used on the web. As such, it is well defined how to encode C strings and Java strings into UTF-8. A reasonable string communication protocol could then have the following form. First, the initial two bytes represent a 16-bit value (in network order, high-order byte first and low-order byte second) that describes the length of the string (in bytes). Next, this is followed by the UTF-8 encoded code units (bytes) that represent the individual characters of the string.

12.3.3 Messages

It is quite common to have a circumstance where more than one thing is to be communicated between two endpoints. For example, a microcontroller might be measuring temperature and pressure, and it wishes to send both values (possibly including a timestamp of when the measurements were taken) to a desktop computer. To accomplish this, we frequently will encapsulate the information to be delivered into one or more *messages*.

A message protocol that has been agreed to by both endpoints allows a range of capabilities that are, at the very least, more difficult without messaging.

1. Recovery from transmission errors.

2. Delivery of distinct data elements (e.g., temperature, pressure, time).

3. Delivery of distinct data types (e.g., integer, float).

In the discussion below, we will cover how to address each of the above capabilities in a message protocol.

Magic Numbers

Consider the following circumstance. A source is sending a series of 2-byte integer values, high byte first and low byte second, and at some point during the delivery process, the high byte of one value is lost. (Recall that our reliability assumption is that occasionally a byte that is sent isn't received.)

What happens in the above circumstance? All of the integers that follow will be erroneously received by the destination. Low bytes of one integer are paired with high bytes of the following integer, and none of it is correct.

In a message protocol, one of the things we would like to accomplish is to recover from errors like the one above, and while some data might be irrevocably lost, at the very least we get the source and destination back in sync with one another so that correct information delivery can resume.

One of the ways we do this is to encapsulate any information we send in a *message*. A message will have *header* information that facilitates the delivery of the entire message (the header is typically independent of the content of the message), and *payload* information that is the data to be communicated from sender to receiver.

An element that is included in the header of many messaging protocols is a *magic number*. A magic number is a fixed byte pattern that is always present at the beginning of a message and is used to signify to the receiver that this is the beginning of a message.

While often the magic number is multiple bytes long (4 bytes is a common size), let us consider the use of a single-byte magic number, for example, `0x21`. When the sender is preparing a message for delivery, it starts the first byte of the message with the byte `0x21`. When the receiver is reading individual bytes from the stream, it can expect that the first byte of any message has the value `0x21`. If not, it knows that the byte that it has just read is *not* the beginning of a message, and an appropriate response would be to discard all incoming bytes until it does see a `0x21`.

A good choice of a magic number is a byte pattern than is relatively unlikely to appear anywhere else in the message. While this is impossible to guarantee in general (e.g., encrypted data can have any value), making the magic number infrequent improves the odds that the receiver's actions in the above paragraph result in erroneously trying to start reading a message while actually still within the body of some other message. Clearly, the use of multiple-byte magic numbers can help this somewhat.

Magic numbers are not constrained to messaging protocols at all. One common use is in files, helping to identify the type of data stored in a file. For example, the first four bytes of a PDF document are `0x25`, `0x50`, `0x44`, and

`0x46`, which are the ASCII encoding of the characters `%PDF`.

Fields

If we have defined the header of our messages (in the case we just described above, the magic number is the only thing in the header, other protocols might specify additional information) it is now time to specify how the data elements are to be delivered. We need to describe the *payload*.

A common approach to data delivery is to not just send the data element itself, but also include additional descriptive information so that the receiver can more readily understand what the data represent. For example, what is the data type: two-byte integer vs. four-byte floating point value vs. string? Or, what is the meaning of the data, temperature reading, pressure reading, or timestamp?

Here, we suggest using a payload convention that goes by several names: *key-value pair*, *name-value pair*, *tag-value pair*, or sometimes *attribute-value pair*. Just to keep things simple, we will use the *key-value pair* terminology, but don't be surprised if you see any of the above terms used elsewhere.

The basic idea is that each data element is communicated as an ordered pair, a *key* followed by a *value*. In our example protocol, the key indicates both the data type and the meaning of the value that follows it (e.g., the value is a four-byte integer that represents a timestamp, the number of milliseconds that has elapsed since the source program was started).

What is required then is that a list of keys (and what they represent) must be known both at the sending and receiving end of the communication. I.e., they must be listed explicitly as part of the protocol.

12.4 Sending Messages: Composition

Given that the source of a message knows the content prior to the actual composition and sending of the message, this task is actually fairly straightforward.

The low-level requirements include how does one send a single byte to the stream. In the Arduino libraries, the `Serial` class supports the delivery of an individual byte via the `Serial.write()` method.

In Java, the delivery of an individual byte depends upon which library is being used. If using the `JSSC` library, which is the one used by the Arduino IDE when communicating between the host computer and the microcontroller, the `SerialPort` class has a `writeByte()` method.

In the example that is included below, we will use the generic `sendByte()` syntax, with the understanding that it gets replaced in the actual code with one of the above options.

Once we have the ability to send individual bytes, there are two possible designs for message delivery. Design 1 is to formulate (compose) the message is memory (in an array of `byte`s) and then send the individual bytes out the data stream. Design 2 is to send bytes out the data stream as the message is being formulated (composed). Either design works just fine. In the example below, we'll use design 1.

Consider the task of sending a message that contains a timestamp value. Assume that the `key` for timestamp is `0x74` and the timestamp value is a 4-byte integer (e.g., the return value from `millis()`). If the byte array we are using to compose the message is named `msg`, the first two things to include in the message are the magic number and the key. This might look like the following:

```
msg[0] = 0x24;
msg[1] = 0x74;
```

or if the appropriate constants have been defined:

```
msg[0] = MAGIC_NUMBER;
msg[1] = TIMESTAMP_KEY;
```

The next four bytes of the message should contain the timestamp value. If that value is in the `unsigned long int time`, the logic to compose the message value is:

```
msg[2] = (time >> 24) & 0xff; // most significant byte
msg[3] = (time >> 16) & 0xff;
msg[4] = (time >> 8) & 0xff;
msg[5] = time & 0xff;          // least significant byte
```

What the above code does is to shift the appropriate 8 bits into the least significant byte and mask off any higher-order bits.

And the last task is to send the message to the stream:

```
int msgLength = 6;
for (i=0; i<msgLength; i++) {
    sendByte(msg[i]);
}
```

which is accomplished by looping over the `msg` array.

12.5   Receiving Messages: Parsing

The challenge in receiving messages is that the receiver doesn't know what message is coming next, and therefore the code to receive messages must recognize any legal messages. It is fairly straightforward to do this using a *finite-state machine* (FSM), and the diagram for such an FSM is illustrated in Figure 12.1.

In the protocol that this FSM recognizes, there are two message keys: `0x74` (with a 4-byte value representing a timestamp) and `0x54` (with a 2-byte value representing an integer temperature in degrees C). In the diagram, an edge is traversed upon the receipt of a byte. If that edge is labeled, the value of the incoming byte must match the label, otherwise the FSM traverses the unlabeled edge. While actions (other than state transitions) are not shown on the diagram, assume that the edges departing states C to H store the received byte into the appropriately named variable: `first`, `second`, `third`, or `fourth` (each of type `byte`).
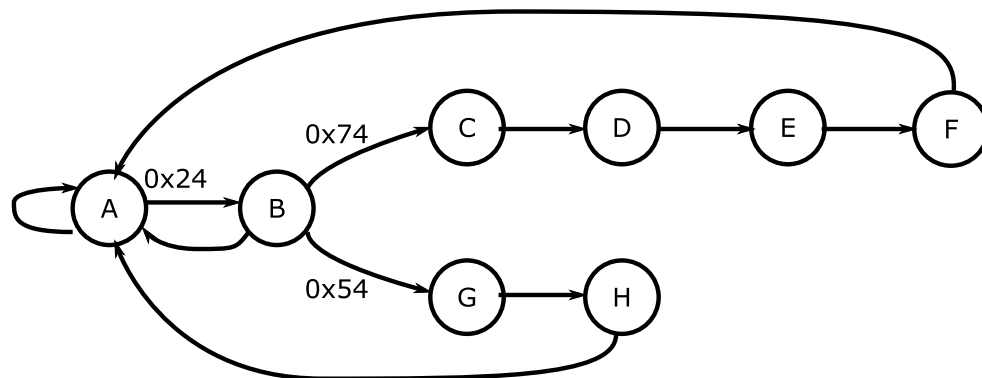
State	Meaning
A	Waiting for magic number (initial state)
B	Waiting for key
C	Waiting for first byte of 0x74 msg value
D	Waiting for second byte of 0x74 msg value
E	Waiting for third byte of 0x74 msg value
F	Waiting for fourth byte of 0x74 msg value
G	Waiting for first byte of 0x54 msg value
H	Waiting for second byte of 0x54 msg value

Figure 12.1: Receiver finite-state machine.

When traversing the outbound edge from state F, the FSM will store the received timestamp into a 4-byte long integer (if on the microcontroller):

```
unsigned long int timestamp = (first << 24) | (second << 16)
                              | (third << 8) | fourth;
```

and when traversing the outbound edge from state H, the FSM will store the received timestamp into a 2-byte integer:

```
unsigned int temperature = (first << 8) | second;
```

To implement the FSM described above, one still needs the capability of receiving individual bytes. Again, this will be different in C on the microcontroller versus in Java on a desktop machine. In Arduino C, the `Serial.available()` method returns the number of bytes that are available to be read from the serial port, and the `Serial.read()` method returns the first byte in the input buffer. As a result, a code structure like the following

```
void loop() {
    if (Serial.available() > 0) {
        byte inputByte = Serial.read();
        switch (FSMstate) {       // code to implement FSM
        case A:
            ...
        }
    }
}
```

will put each received byte into `inputByte` and then implement one transition (one step) of the finite-state machine (whose state is retained in `FSMstate`).

13 Conclusions

NOTE: This chapter has not yet been written.

A Languages

In this appendix, we will make targeted comparisons between what are essentially three languages. Under the assumption that many readers have experience with Java, we will compare Java with standard C (a language that predates Java by several decades, and is used quite frequently in embedded computer systems). The version of C used throughout the book, however, is a subset of the standard C language. We refer to it as Arduino C, and after the comparison between Java and C we will compare standard C with Arduino C.

In both cases, the intent of the comparison is not to fully describe the similarities and differences between the language definitions. Rather, we are interested in helping the reader navigate the distinctions they are likely to run into if they are initially familiar with the first language and are just being introduced to the second language. The examples given are just that, examples, not intended to elucidate every corner case or potential difference.

A.1 Java vs. C

Java is a language that is frequently used in introductory courses that teach computer programming. As such, we anticipate that many readers of this book will be familiar with it, and since Java derives many of its syntactic conventions from C, this makes the task of learning C substantially easier for those who are familiar with Java. In the paragraphs below, we will attempt to answer the question, "What does a programmer familiar with Java need to know to become reasonably functional in C?"

Probably the most important thing to know that differentiates Java and C is the fact that Java is an object-oriented language, while C is not. C was developed long before object-oriented programming was in vogue and does not support objects, per se. It is a procedural language. We will discuss this distinction a bit more below, however, we will start our comparison of the two languages with things for which they are more similar than different.

A.1.1 Basic Syntax

The look and feel of individual statements in Java and C are remarkably similar. This is not an accident, but derives from the fact that this was an intentional decision by the developers of Java (the newer of the two languages).

As such, the following things are essentially unchanged between the two languages:

- Identifiers use the same rules, e.g., they are case sensitive.

- Individual statements are terminated with a semi-colon, ';'.

- Scope is delineated using curly braces, '{' and '}'.

- Function calls look the same, e.g., `function(param1, param2)`.

- Comments are denoted in the same way, i.e., using '`/*`' with '`*/`' or '`//`'.

- Many unary and binary operators are the same, e.g., `+`, `-`, `*`, `/`, `&&`, `||`, `!`, `&`, `|`, `~`, `^`, `<=`, `>=`, and `==`.

One minor distinction is that the right shift operator, '`>>`', is a signed operation in Java, while in C it follows the type (signed vs. unsigned) of the value to be shifted. An unsigned right shift in Java is denoted by '`>>>`'.

The code snippet below could legitimately be in either Java or C:

```
int a = 0;
int b = 100;
while (a < b) {
    for (int i=0; i<20; i++) {
        if (a > i) {
            a++;
        }
        else {
            b--;
        }
    }
}
```

While it doesn't really do anything interesting, one cannot tell simply by reading it which language it represents. It could be either.

A.1.2 Primitive Data Types

Java has the following primitive types: `byte`, `char`, `short`, `int`, `long`, `float`, `double`, and `boolean`. Other than `char` or `boolean`, all of the types are considered *signed*, in that they can store both positive and negative values. The storage requirements (size) for each of these types is specified by the language.

C has the following primitive types: `char`, `unsigned char`, `short int`, `unsigned short int`, `int`, `unsigned int`, `long int`, `unsigned long int`, `float`, `double`, and `long double`. Note, the type `unsigned` is a synonym for `unsigned int`, `short` is a synonym for `short int`, and `long` is a synonym for `long int`. In C, the storage requirements for each type is compiler-dependent, with the language specifying a minimum size.

Table A.1 shows sizes for a number of the primitive data types. The Java column has the actual size while the C column shows the size that is commonly used by modern compilers. Note, these typical sizes for C types are for desktop and server systems. We will revisit the size of data types in Arduino C below.

Table A.1: Sizes of some primitive data types in Java and C.

Data Type	Java Size	Typical C Size	Range (of integers)
`byte`	1 byte	N/A	
`char`	2 bytes	1 byte	
`unsigned char`	N/A	1 byte	
`short`	2 bytes	2 bytes	-2^{15} to $2^{15}-1$
`unsigned short`	N/A	2 bytes	0 to $2^{16}-1$
`int`	4 bytes	4 bytes	-2^{31} to $2^{31}-1$
`unsigned`	N/A	4 bytes	0 to $2^{32}-1$
`long`	8 bytes	8 bytes	-2^{63} to $2^{63}-1$
`unsigned long`	N/A	8 bytes	0 to $2^{64}-1$
`float`	4 bytes	4 bytes	
`double`	8 bytes	8 bytes	

The first observation we will make is that for the most part, the size of Java types matches the size of C types. This is true for the 2-byte integers (`short`s), 4-byte integers (`int`s), and 8-byte integers (`long`s). The second observation is that for a single-byte data type, Java uses the `byte` type and C uses the `unsigned char` type. It is common in C environments for a local definition (`typedef` in C) to declare a `byte` type that is the same as an `unsigned char`, further unifying the commonly used primitive types. Finally, our third observation is that the data type for characters is distinctly different

173

in the two languages. Java stores individual characters as 2-byte values using the UTF-16 character set in its `char` data type, while C stores characters as single-byte values using the ASCII subset of the UTF-8 character set in its `char` data type.

A.1.3 Strings

The two languages not only differ in how they store characters (i.e., the `char` data type), they also differ in how they store strings.

In Java, strings are managed by the `String` class, and objects of the `String` class are immutable (i.e., they cannot be altered once created). The `String` class supports a number of methods that are quite useful, such as querying the length of a string via the `String length()` method. While not required by the language definition, the common implementation is for each `String` object to retain the characters of the string in an array of `char`s and the length of the string in a separate instance variable. These details are, of course, hidden from the programmer by the interface to the `String` class.

In C, strings are stored in an array of C `char`s, with the array size required to be at least one element larger than the length of the string. The end of the string is denoted by the NULL character (0x00, `'\0'`), in the array position immediately after the final character of the string. As a result of this convention, strings in C are mutable, or alterable, simply by manipulating the individual elements of the array. In addition, this convention for storing strings, an array of `char`s that is NULL terminated, is totally exposed to the programmer and is not hidden behind an opaque interface.

A.1.4 Arrays

Arrays in Java and C are similar in some ways, but noticeably different in others. Pay attention to the following things with Java and C arrays:

- The syntax for array references is the same, e.g., `a[3]` refers to the fourth element of the array `a`, with indexing starting at 0 in both languages.

- The declarations and allocations differ some, e.g., a length 10 array of integers named `a` can be declared via

  ```
  int[] a = new int[10];
  ```
  in Java and

  ```
  int a[10];
  ```
  in C.

- An array in Java is an object, so there are a number of methods that can be invoked on it, e.g., `a.toString()` returns a `String` representation of the array `a`.

A.1.5   Heterogeneous Data Structures and Objects

Once one has some experience using arrays, which support a number of data elements that have the same type, the next natural thing one might like to declare is a data structure that can hold more than one data type. In Java, we can do this using objects, each of which can contain a number of instance variables, all of different types.

While Java object are much more than just a heterogeneous data structure (e.g., they also can contain methods), our interest here is just in their data retention capabilities.

C, on the other hand, does not support objects at all. It does, however, use a different mechanism for declaring data structures that are heterogeneous in nature, called a *structure*. An example structure definition is shown below, using the keyword `struct` to define a structure with the name `alpha`.

```
struct alpha {
     int   k;
     float x;
};
```

Subsequently, one can then declare a variable `alf` whose type is the structure `alpha`:

```
struct alpha alf;
```

and access fields within the structure as follows.

```
alf.k = 2;
alf.x = 3.14159;
```

If it helps your understanding, it is quite reasonable to think of a `struct` as just an object with instance variables but no methods.

A.1.6   Memory Management

A clear difference between Java and C is the way that dynamic memory management is handled. In Java, the bulk of the memory management responsibilities are with the language infrastructure. Programmers can request new

objects, which are allocated in memory, they are not responsible for managing pointers to those objects, and the system handles the reclaiming of memory once the program is finished with it.

In C, almost all of the above is different. Programmers are explicitly responsible for allocated memory, which must be freed by the programmer when no longer needed. Pointer data types are explicitly supported in the language to facilitate direct manipulation of the memory subsystem.

In most small embedded systems like the Arduino, the use of dynamic memory is either disallowed or discouraged, so while this is a subject of major differences between Java and C as a language, it is not an issue that will be substantial for readers of this book.

A.1.7   Other Minutiae

There are, of course, a host of other differences between Java and C, some in philosophy, others just in minor semantic variation. One example of this is the fact that, by default, Java will initialize all variables at their declaration, while C will not. As a result, the declarations below:

```
int abc = 0;
int xyz;
```

will not have the same impact in Java vs. C. In Java, both `abc` and `xyz` will start out with the value 0. In C, however, `abc` will have the value 0, but `xyz` will be indeterminate.

Another important distinction between Java and C is that the logistics of printing is different in each language. Since Arduino C actually differs fairly significantly from standard C in this respect, we omit a discussion of a comparison between Java and C and below discuss the logistics of printing in standard C and Arduino C in the section below.

A.2   C vs. Arduino C

As stated earlier, the C language supported by the Arduino platform has some distinctions that separate it from standard C. Here, we will highlight those differences, so that the reader that is familiar with the C language won't be tripped up by Arduino C.

176

A.2.1 Primitive Data Types

As mentioned above, the size of primitive data types in C is compiler-dependent. Table A.2 gives the size of commonly used data types that are supported by the Arduino compiler.

Table A.2: Sizes of primitive data types in Arduino C.

Data Type	Arduino C Size
bool	1 byte
byte	1 byte
char	1 byte
unsigned char	1 byte
int	2 bytes
unsigned	2 bytes
long	4 bytes
unsigned long	4 bytes
float	4 bytes
double	4 bytes

There are a number of things to note here:

- The `bool` data type has an alias, `boolean`, and supports the constants `true` and `false`. When evaluating a logical test, the value 0 is interpreted as FALSE and any non-zero value is interpreted as TRUE. (This last point is valid for standard C as well.)

- The `byte` data type is actually the same as `unsigned char`.

- The `int` and `unsigned` data types are smaller than is typical for desktop or server machines. This limits the range of `int` to -2^{15} to $2^{15} - 1$, or $-32,768$ to $32,767$, and the range of `unsigned` to 0 to $2^{16} - 1$, or 0 to $65,535$.

- The `float` and `double` data types are equivalent to one another (i.e., they are the same size). This is never the case in desktop or server machines, but is not unusual in small microcontrollers.

A.2.2 Objects

We've stated several times that C is not an object-oriented language. However, the compiler used with the Arduino platform is, technically, not just a C

compiler but is a C/C++ compiler. As such, it supports a number of C++ features, including classes and objects.

The declaration and implementation of classes is beyond the scope of this appendix. However, there are a number of commonly used classes that are available as libraries and can be invoked from Arduino C sketches. The two most commonly used classes are the following:

- `String` – The `String` class is an alternative approach to storing strings. Objects of the `String` class support methods such as `concat()`, which enables concatenation of two strings, and `length()`, which returns the number of characters in an object of type `String`.

- `Serial` – The `Serial` class is used to support communication between the Arduino microcontroller and other computers. It is the primary vehicle to support printing on the Arduino and gets used extensively.

A.2.3 Printing

Likely the largest difference that the typically programmer will see is the printing support provided. In Arduino C, the printing of strings is supported via the `Serial.print()` function within the `Serial` class. In standard C, the formatting of output is handled primarily by `printf()` (with similar functionality provided by `sprintf()` and `fprintf()` for printing to strings and files, respectively).

`Serial.print()` supports the printing of all of the data types shown in Table A.2 in addition to strings (both `String` objects and NULL-terminated arrays of type `char`). An optional second parameter specifies the base to be used for formatting the value. Valid bases include `BIN` for binary, `OCT` for octal, `DEC` for decimal, or `HEX` for hexadecimal. If the first argument is a floating-point value, the optional second parameter specifies the number of fractional digits to print after the decimal point.

B   Simple Introduction to Electricity

NOTE: This appendix has not yet been written.

C   Base Conversions

When converting from one number base to another number base, the primary consideration is which base does one wish to use for the mathematical operations. In what follows, we will convert numbers from base A to base B using base B math, and then will convert from base A to base B using base A math. In both cases the math will be in decimal (i.e., in the first section, base B is decimal and in the second section base A is decimal).

C.1   Convert Base A to Base B using Base B Math

When the destination base of the conversion is the same as the base used to perform arithmetic, the conversion is essentially an application of the basic definitions of the positional number system. Given the 3-digit number denoted uvw_a in base a, where u is the 1st digit, v is the 2nd digit, and w is the 3rd digit, the conversion to base 10 (using base 10 arithmetic) is as follows:

$$uvw_a = u \cdot a^2 + v \cdot a^1 + w \cdot a^0$$
$$= u \cdot a^2 + v \cdot a + w.$$

It is important when performing the operations above that the individual digits u, v, and w, as well as the base a, are all represented on the right-hand side of the equation using their decimal equivalents. This enables the decimal math to function. Notationally, the base is a on the left-hand side of the equation and the base is 10 on the right-hand side.

As a first example, we will convert the hexadecimal value 0x3e1 into decimal. Putting this in terms of the symbols above, $u = 3$, $v = e_{16} = 14_{10}$, and $w = 1$. Converting 0x3e1 from hexadecimal to decimal (base 16 to base 10)

181

using decimal arithmetic then proceeds as follows:

$$3\mathrm{e}1_{16} = 3 \cdot 256 + 14 \cdot 16 + 1$$
$$= 768 + 224 + 1$$
$$= 993_{10}.$$

In a second example, we will generalize beyond the 3-digit numbers above, and convert an 8-bit binary value (base 2) into decimal. As the number of digits in the initial number increases, we simply generalize the equation above to use increasing powers of the input base a. Take the value 10010101_2, which has 8 binary digits (or bits). It is converted into decimal as shown below:

$$10010101_2 = 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$
$$= 1 \cdot 128 + 0 \cdot 64 + 0 \cdot 32 + 1 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1$$
$$= 128 + 16 + 4 + 1$$
$$= 149_{10}.$$

There is one wrinkle that must be considered when converting from binary numbers into decimal. In the above example, we treated the binary value as an unsigned number. If, instead, the 8-bit binary number is to be interpreted as a two's complement signed value, the weight associated with the most significant bit position is no longer 2^7, but is instead -2^7. In this case, the conversion proceeds as shown below:

$$10010101_2 = 1 \cdot -2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$
$$= 1 \cdot -128 + 0 \cdot 64 + 0 \cdot 32 + 1 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1$$
$$= -128 + 16 + 4 + 1$$
$$= -107_{10}.$$

Note that there is no information present in the original binary number, 10010101, that indicates whether it is to be interpreted as an unsigned or a signed value. The interpretation must be communicated separately from the number itself. Indeed, the same bit pattern can be interpreted either way, and as we see above, each interpretation yields a different decimal value.

C.2 Convert Base A to Base B using Base A Math

When the mathematical manipulations are being performed in the destination base, the required operations are fairly straightforward, requiring just

the insertions of the appropriate values into the conversion formula and then execution of some multiplication and addition. The situation is not quite as straightforward when the mathematical operations are to be performed in the origin base.

The following technique is characterized by repeated division operations. In the description that follows, we will assume that base A is 10 (the base in which we are performing our mathematical manipulations) and base B is represented by b. The procedure repeatedly performs integer division, retaining both the quotient, q, and the remainder, r. The resulting value (in base B) is constructed one digit at a time, starting with the least significant digit and proceeding towards the most significant digit.

1. $temp \leftarrow$ value to be converted

   $result \leftarrow$ empty

2. perform integer division $temp/b$ resulting in quotient q and remainder r

3. prepend r (as an individual digit in base B) to the front of $result$ (i.e., r is the new most significant digit of $result$)

4. $temp \leftarrow q$

5. if $q \neq 0$ then return to step 2

As an initial example, we will reverse the initial base conversion we did at the beginning, converting 993_{10} into hexadecimal (base 16). The labels on each line below refer to the specific step being performed in the algorithm above. Values in base 10 will not have the base explicitly shown, and values in base 16 will be denoted via a subscript 16.

(1) $temp = 993$ and $result$ is empty

(2) divide 993/16, which gives quotient $q = 62$ and remainder $r = 1 = 1_{16}$

(3) $result = 1_{16}$

(4) $temp = 62$

(5) $q = 62$, return to step 2

(2) divide 62/16, which gives $q = 3$ and $r = 14 = e_{16}$

(3) $result = e1_{16}$

(4) *temp* $= 3$

(5) $q = 3$, return to step 2

(2) divide 3/16, which gives $q = 0$ and $r = 3 = 3_{16}$

(3) *result* $= 3\mathrm{e}1_{16}$

(4) *temp* $= 0$

(5) $q = 0$, finished

At the end of the above procedure, *result* is $3\mathrm{e}1_{16}$, which is what we expect.

As a second example, we will convert 134_{10} into binary. The sequence of steps is as follows:

(1) *temp* $= 134$ and *result* is empty

(2) divide 134/2, which gives quotient $q = 67$ and remainder $r = 0$

(3) *result* $= 0_2$

(4) *temp* $= 67$

(5) $q = 67$, return to step 2

(2) divide 67/2, which gives $q = 33$ and $r = 1$

(3) *result* $= 10_2$

(4) *temp* $= 33$

(5) $q = 33$, return to step 2

(2) divide 33/2, which gives $q = 16$ and $r = 1$

(3) *result* $= 110_2$

(4) *temp* $= 16$

(5) $q = 16$, return to step 2

(2) divide 16/2, which gives $q = 8$ and $r = 0$

(3) *result* $= 0110_2$

(4) *temp* $= 8$

(5) $q = 8$, return to step 2

(2) divide $8/2$, which gives $q = 4$ and $r = 0$

(3) $result = 00110_2$

(4) $temp = 4$

(5) $q = 4$, return to step 2

(2) divide $4/2$, which gives $q = 2$ and $r = 0$

(3) $result = 000110_2$

(4) $temp = 2$

(5) $q = 2$, return to step 2

(2) divide $2/2$, which gives $q = 1$ and $r = 0$

(3) $result = 0000110_2$

(4) $temp = 1$

(5) $q = 1$, return to step 2

(2) divide $1/2$, which gives $q = 0$ and $r = 1$

(3) $result = 10000110_2$

(4) $temp = 0$

(5) $q = 0$, finished

which gives $result = 10000110_2$. We can check this by converting from binary back into decimal:

$$
\begin{aligned}
10000110_2 &= 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \\
&= 1 \cdot 128 + 0 \cdot 64 + 0 \cdot 32 + 0 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 1 \cdot 2 + 0 \\
&= 128 + 4 + 2 \\
&= 134_{10}.
\end{aligned}
$$

The above procedure does not readily admit to two's complement representations. To convert a negative decimal number into its binary two's complement equivalent, first convert the decimal magnitude into unsigned binary, extend with 0s to the left so that the correct number of digits (bits) are included, and then negate the result.

Bibliography

[1] George Boole. *An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities.* Macmillan, Ltd., Cambridge, UK, 1854.

[2] Brahmagupta. *Algebra, with Arithmetic and Mensuration.* John Murray Press, London, UK, 1817. Translated from *Brāhmasphuṭasidhānta* (c. 628) by Henry Thomas Colebrooke.

[3] Roger D. Chamberlain, Ron K. Cytron, Doug Shook, and Bill Siever. Computers interacting with the physical world: A first-year course. In *Proc. of Workshop on Embedded and Cyber-Physical Systems Education,* October 2018. DOI: 10.1007/978-3-030-23703-5_11.

[4] Leonhard Euler. Recherches sur les racines imaginaires des équations. *Histoire de l'Académie Royale des Sciences et des Belles-Letters de Berlin,* 5:222–288, 1751.

[5] IEEE. Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Standard No. 754-1985, Institute of Electrical and Electronics Engineers, Inc., New York, NY, USA, 1985.

[6] Microchip Technology, Inc. AVR® Instruction Set Manual, 2021. http://ww1.microchip.com/downloads/en/DeviceDoc/AVR-InstructionSet-Manual-DS40002198.pdf.

[7] Mary C. Potter, Brad Wyble, Carl Erick Hagmann, and Emily S. McCourt. Detecting meaning in RSVP at 13 ms per picture. *Attention, Perception, & Psychophysics,* 76(2):270–279, 2014.

[8] Denise Schmandt-Besserat. *Before Writing, Vol. I: From Counting to Cuneiform.* University of Texas Press, Austin, TX, USA, 1992.

[9] Gregory K. Wallace. The JPEG still picture compression standard. *Communications of the ACM*, 34(4):30–44, April 1991.

Index