COMPUTING IN THE PHYSICAL WORLD

Roger D. Chamberlain and Ron K. Cytron

Version 0.05 - Draft Chapters 1 to 7 and 9
## Contents

1 Introduction 1
   1.1 Beginnings 1
       1.1.1 Why? 1
       1.1.2 The Arduino Platform 2
   1.2 Digital Systems 3
   1.3 Authoring Programs 7
   1.4 Interacting with the Physical World 8

2 Digital Output 9
   2.1 Why Digital Outputs? 9
   2.2 Software 10
   2.3 Example Digital Output Use Cases 11
       2.3.1 LED Indicator 11
       2.3.2 Buzzer 12
       2.3.3 Relay 12

3 Digital Input 15
   3.1 Why Digital Inputs? 15
   3.2 Hardware 16
   3.3 Software 16
   3.4 Example Digital Input Use Cases 17
       3.4.1 Switch 17
       3.4.2 Proximity Detector 18
   3.5 Debouncing Mechanical Contacts 18
   3.6 Hardware vs. Software 20

4 Analog Output 25
   4.1 Why Analog Outputs? 25
   4.2 Relating Analog Output Values to Physical Reality 26
# Contents

4.3 Software ........................................... 28

4.4 Example Analog Output Use Cases .............. 28
   4.4.1 Motor Speed .................................. 28
   4.4.2 Loudness ...................................... 29

5 Analog Input ......................................... 31
   5.1 Why Analog Inputs? .............................. 31
   5.2 Counts to Engineering Units ................. 32
      5.2.1 Input Range and Linear Transformation .... 32
   5.3 Software ........................................ 34
   5.4 Example Analog Input Use Cases ............ 35
      5.4.1 Temperature ................................ 35
      5.4.2 Level ........................................ 36
      5.4.3 Acceleration ................................. 38

6 Timing ............................................... 41
   6.1 Execution Time ................................. 42
   6.2 Controlling Time ............................... 43
   6.3 Delta Time ...................................... 45

7 Information Representation .......................... 47
   7.1 Numbers ......................................... 47
      7.1.1 Brief History of Number Systems .......... 47
      7.1.2 Positional Number Systems ............... 53
      7.1.3 Supporting Negative Numbers ............... 57
      7.1.4 Integer Data Types in Programming Languages ... 61
      7.1.5 Fractional Numbers ........................... 62
      7.1.6 Real Numbers ................................ 63
   7.2 Text: Characters and Strings ................ 66
      7.2.1 ASCII ....................................... 66
      7.2.2 Unicode .................................. 68
      7.2.3 String Representations ...................... 69
   7.3 Images ........................................... 70
      7.3.1 Monochrome Images .......................... 71
      7.3.2 Color Images ................................ 71

8 User Interaction ...................................... 73

9 Computer Architecture and Assembly Language ... 75
   9.1 Basic Computer Architecture ..... 75
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 come a long ways. 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
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 Atmel, 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, \texttt{setup()} and \texttt{loop()}. The code that is in \texttt{setup()} is executed once, at the beginning of the run, and the code that is in \texttt{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 9. 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).
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 [2]).

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

$$E = (A \text{ AND } B) \text{ OR } C \quad (1.1)$$
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>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(b)

<table>
<thead>
<tr>
<th>x</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

(c)

Table 1.1: (a) AND truth table. (b) OR truth table. (c) NOT truth table.

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

\[ E = (A \land B) \lor C \]  

(1.2)

where the \( \land \) symbol is used to represent the AND operation and the \( \lor \) 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,

\[ E = (A \cdot B) + C \]  

(1.3)

or this,

\[ E = AB + C \]  

(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>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>AB</th>
<th>AB + C</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

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.](image)

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.](image)

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).](image-url)
1.4 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 systems that control the flight surfaces (e.g., rudder, elevators, ailerons) in a fighter jet, and the Fitbit™ that you might be wearing on your wrist right now.

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.
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 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.).

```cpp
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 2.1: Example digital output sketch.
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 2.2 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).

Figure 2.2: Schematic diagram of digital output controlled LED. The anode is the topmost terminal of the LED and the cathode is the bottom terminal.

If the sketch from Figure 2.1 is executed on the microcontroller that has the schematic from Figure 2.2 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 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. That is a completely arbitrary choice, however. Consider the schematic diagram shown in Figure 2.3. In this case, the digital output is connected to the cathode side of the LED, and the anode side goes to +5 V. In this case, an output HIGH, gives 5 V on both the
2. Digital Output

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. As a result of the alternative schematic connections, the operation of the light as a function of the digital output polarity has been reversed.

Figure 2.3: Schematic diagram of digital output controlled LED with altered 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.

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 electrical circuit(s) that connect them.

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 2.4. When the digital output is HIGH, the buzzer makes sound, and when the digital output is LOW, the buzzer is silent.

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 power from the wall socket. This might include motors, pumps, heating elements, etc.
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.

![Schematic diagram of circuit that interfaces a pushbutton input to a microcontroller digital input pin.](image)

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.

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.
3.4. Example Digital Input Use Cases

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 is a use case that is also 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.

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 `digitalRead()` function returns the constant value `LOW` (which is defined as a 0). If the voltage at the pin is near the midpoint ($\approx 2.5$ V), the return value is indeterminate, and either a `HIGH` or a `LOW` might result.

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`.

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.

\[1\] 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.
3. Digital Input

```cpp
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.

3.4.2 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.

3.5 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. 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.
3.5. Debouncing Mechanical Contacts

The horizontal scale is 200 $\mu$s/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 $\mu$s, while we are observing the signal over several hundred $\mu$s (almost 2 ms). Compare this to the time scale of the microcontroller, which executes multiple instructions every $\mu$s (approximately 16 instructions per $\mu$s 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
3. Digital Input

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 $\mu$s. 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.

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

$$ E = AB + C $$

which is the same as Equation (1.4).

While Figure 3.5 illustrates how to compute the economics requirement entirely in hardware, Figure 3.6 shows a sketch that computes the same economics requirement entirely in software. In the sketch, the inputs $A$, $B$, 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
3.6. Hardware vs. Software

```cpp
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.

Figure 3.5: Gate-level schematic diagram of circuit that implements economics course requirement check.

... a student's economics requirement) either in hardware or in software. It is worthwhile to consider some of the differences between these two implementations, 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.
3. Digital Input

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.
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.
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%.

Next, consider what happens when we provide the waveforms described above to an output pin wired to an LED as in Figure 2.2. 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 [6]. Instead, our eyes (and brain) respond to the average light intensity, 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,
4.2. Relating Analog Output Values to Physical Reality

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.
4. Analog Output

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).

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).

4.4 Example Analog Output Use Cases

4.4.1 Motor Speed

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 4.4.

![Figure 4.4: Schematic diagram of directly connected 5 V DC motor.](image)

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 4.5.
4.4. Example Analog Output Use Cases

Figure 4.5: 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.

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

```
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 4.6. 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.
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.

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

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

where the value of `volume` can range from 0 (minimum gain) to 255 (maximum gain).
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 $V_{REF}$. 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. Analog Input

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.](image)

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

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

$$s = 43 \, [\text{mV/\text{lb}}] \cdot w + 200 \, [\text{mV}]$$  (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
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

\[
c = \frac{(1023 - 0)}{(5000 - 0)} \cdot s + 0
\]

\[
c = 0.2046 \left( \frac{\text{cnt}}{\text{mV}} \right) \cdot s 
\] (5.2)

where \( s \) is again the analog signal (in mV) and \( c \) is the A/D counts (cnt).

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

\[
c = 0.2046 \cdot s
\]

\[
c = 0.2046 \cdot (43 \cdot w + 200)
\]

\[
c = 8.7978 \left[ \frac{\text{cnt}}{\text{lb}} \right] \cdot w + 40.92 \text{ [cnt]} 
\] (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).

\[
w = 0.1136648 \left[ \frac{\text{lb}}{\text{cnt}} \right] \cdot c - 4.6512 \text{ [lb]} 
\] (5.4)

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

Figure 5.3: Analog to digital converter response.

Table 5.1: Parameters for `analogReference()`

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Analog Reference Voltage</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFAULT</td>
<td>5 V</td>
</tr>
<tr>
<td>INTERNAL</td>
<td>1.1 V</td>
</tr>
<tr>
<td>EXTERNAL</td>
<td>AREF pin value</td>
</tr>
</tbody>
</table>

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

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.
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
for the voltage response of the probe as follows:

\[ s = 10 \left[ \frac{\text{mV}}{\text{C}} \right] \cdot t \]  

(5.5)

where \( s \) is the analog voltage signal into the A/D converter and \( t \) is the temperature in \( \text{C} \).

If we use \texttt{INTERNAL} as the parameter to \texttt{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 \( \text{C} \)). The conversion from analog voltage to A/D counts is therefore

\[ c = \frac{(1023 - 0)}{(1100 - 0)} \cdot s \]  

(5.6)

\[ c = 0.93 \left[ \frac{\text{cnt}}{\text{mV}} \right] \cdot s \]  

(5.7)

which gives

\[ c = 0.93 \cdot (10 \cdot t) \]  

(5.8)

\[ c = 9.3 \left[ \frac{\text{cnt}}{\text{C}} \right] \cdot t \]  

(5.9)

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

\[ t = 0.1075 \left[ \frac{\text{C}}{\text{cm}} \right] \cdot c \]  

(5.10)

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

\texttt{temp = (float) (0.1075 * rawValue);} 

where \texttt{temp} is a \texttt{float} representing temperature and \texttt{rawValue} is an \texttt{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
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.

\[
\begin{align*}
  s &= \frac{(1000 - 5000)}{(4 - 0)} \cdot L + 5000 \\
  s &= -1000 \, \text{[mV/cm]} \cdot L + 5000 \, \text{[mV]}
\end{align*}
\]  

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

\[
\begin{align*}
  c &= 0.2046 \cdot (-1000L + 5000) \\
  c &= -204.6 \, \text{[cnt/cm]} \cdot L + 1023 \, \text{[cnt]}
\end{align*}
\]

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

\[
L = -0.0048876 \, \text{[cm/cnt]} \cdot c + 5 \, \text{[cm]}
\]

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

\[
\text{level} = (\text{float})(-0.004876 \times \text{rawValue} + 5);
\]

where \text{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.
5. Analog Input

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 I²C 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 5.6: Block diagram and directional reference for Freescale MMA8451Q accelerometer (from the data sheet).](image)

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 someone’s 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 9, 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 incrementing 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 \( T = 1/f = 1 \mu s \).) 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).

\[
\begin{align*}
\text{startTime} &= \text{readFreeRunningCounter()} \\
&\quad \text{// execute timed code} \\
\text{endTime} &= \text{readFreeRunningCounter()} \\
\text{runTime} &= \text{endTime} - \text{startTime}
\end{align*}
\]

Figure 6.1: Measuring elapsed time with a free-running counter. The variable \text{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, \text{millis()} returns the number of milliseconds since the last processor reset, and the second, \text{micros()} returns the number of microseconds since the last processor reset. Both functions return a \text{long int} type since an \text{int} will quickly run out of space to store sufficiently large values (see Chapter 7).

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 \text{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 \text{delay()} call, and those instructions take time to execute. Both calls to \text{digitalWrite()}
6. Timing

```c
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,

```c
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 \texttt{delay()} routine to control time has the issues described above, that all the code that is outside the \texttt{delay()} call isn’t accounted for in the elapsed time for the loop. We will next examine a more robust timing approach, called \textit{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.

\begin{verbatim}
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
    }
}
\end{verbatim}

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, \texttt{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 \texttt{delay()} based approach above in several ways. First, as long as the code that is executed once per loop (call it an \textit{iteration}) doesn’t take longer than 1 s to run, the code within the \texttt{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).
7 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.

7.1 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.

7.1.1 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 [7], although it might have happened even earlier than that.

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

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

\[ \text{This was not always the case, the Babylonians used base 60, which is how we ended up with 60 seconds in a minute and 60 minutes in an hour.} \]
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 \[3\]. 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
7.1. Numbers

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

$$ x + a = 0 $$

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. 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.

---

2Although 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

\[
ax + b = 0
\]

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 7.1 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

\[
a^2 + b^2 = x^2
\]

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

\[
x^2 - 2 = 0
\]

for which there are two solutions: \( x = \sqrt{2} \) and \( x = -\sqrt{2} \).

Figure 7.1: 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 \( (7.4) \), 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), $\pi$ (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:

$$x^2 + 2 = 0 \quad (7.5)$$

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 $a = c$ 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

---

*The notation iff is shorthand for *if and only if.*
7. Information Representation

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 (7.5). First we rewrite it as an equation in complex numbers (our two component vector number system).

\[ x^2 + (2, 0) = (0, 0) \] (7.6)

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

\[
\begin{align*}
x^2 + (2, 0) &= (0, \sqrt{2})^2 + (2, 0) \\
&= (0, \sqrt{2}) \times (0, \sqrt{2}) + (2, 0) \\
&= (-2, 0) + (2, 0) \\
&= (0, 0)
\end{align*}
\]

This shows that \( x = (0, \sqrt{2}) \) is a solution to Equation (7.5).

Another interesting equation is shown below.

\[ x^2 + 1 = 0 \] (7.7)

A little bit of algebraic manipulation yields the following,

\[
\begin{align*}
x^2 + 1 &= 0 \\
x^2 &= -1 \\
x &= \sqrt{-1}
\end{align*}
\]

which is a number that has intrigued folks for years. Let’s now try out Equation (7.7) with \( x = (0, 1) \):

\[
\begin{align*}
x^2 + (1, 0) &= (0, 1)^2 + (1, 0) \\
&= (0, 1) \times (0, 1) + (1, 0) \\
&= (-1, 0) + (1, 0) \\
&= (0, 0)
\end{align*}
\]

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
7.1. Numbers

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:

\[
a_0x^n + a_1x^{n-1} + ... + a_{n-1}x + a_n = 0 \tag{7.8}
\]

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.

7.1.2 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

\[
uvw.xyz_{10} = u \cdot 10^2 + v \cdot 10^1 + w \cdot 10^0 + x \cdot 10^{-1} + y \cdot 10^{-2} + z \cdot 10^{-3}
\]

\[
= u \cdot 100 + v \cdot 10 + w + x \cdot 0.1 + y \cdot 0.01 + z \cdot 0.001.
\]
7. Information Representation

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

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

$$= u \cdot 4 + v \cdot 2 + w + x \cdot \frac{1}{2} + y \cdot \frac{1}{4} + z \cdot \frac{1}{8}.$$ 

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 011.100₂, then \( u = 0, v = 1, w = 1, x = 1, y = 0, \) and \( z = 0. \) The decimal value is therefore:

\[
011.100₂ = (0 \cdot 4) + (1 \cdot 2) + 1 + (1 \cdot \frac{1}{2}) + (0 \cdot \frac{1}{4}) + (0 \cdot \frac{1}{8})
\]

\[
= 0 + 2 + 1 + 0.5 + 0 + 0
\]

\[
= 3.5_{10}
\]

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 \( \text{a} \) through \( \text{f} \) as variables in algebraic notation, but instead think of
them as numerical digits.\footnote{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.} Table 7.1 gives the value (in decimal) of each of the digits we will use in hexadecimal notation.

### Table 7.1: Value (in decimal) of hexadecimal digits.

<table>
<thead>
<tr>
<th>Hex digit</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

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

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

$$= u \cdot 256 + v \cdot 16 + w + x \cdot \frac{1}{16} + y \cdot \frac{1}{256} + z \cdot \frac{1}{4096}.$$

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

$$1f.4 = (1 \cdot 16) + 15 + (4 \cdot \frac{1}{16})$$

$$= 16 + 15 + 0.25$$

$$= 31.25.$$

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
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

\[
\begin{align*}
1fc7.2d_{16} &= \\
0001 &1111 1100 0111 0101 0100_2
\end{align*}
\]

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

\[
\begin{align*}
86eb01_{16} &= \\
1000 &0110 1110 1011 0000 0001_2
\end{align*}
\]

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.

### 7.1.3 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 7.2: Sign-magnitude integers.

<table>
<thead>
<tr>
<th>Binary number</th>
<th>Decimal value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>+0</td>
</tr>
<tr>
<td>10</td>
<td>−0</td>
</tr>
<tr>
<td>01</td>
<td>+1</td>
</tr>
<tr>
<td>11</td>
<td>−1</td>
</tr>
<tr>
<td>0101</td>
<td>+5</td>
</tr>
<tr>
<td>1101</td>
<td>−5</td>
</tr>
<tr>
<td>01000000</td>
<td>+64</td>
</tr>
<tr>
<td>11000000</td>
<td>−64</td>
</tr>
</tbody>
</table>

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 7.2 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 \( 3 - 7 = -4 \).
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 7.3 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...
7. **Information Representation**

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

$$0100 = (0 \cdot -8) + (1 \cdot 4) + (0 \cdot 2) + 0$$
$$= 0 + 4 + 0 + 0$$
$$= 4$$

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

$$1101 = (1 \cdot -8) + (1 \cdot 4) + (0 \cdot 2) + 1$$
$$= -8 + 4 + 0 + 1$$
$$= -3$$

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.

### 7.1.4 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.

\[
\begin{align*}
  b_{15} & b_{14} b_{13} b_{12} b_{11} b_{10} b_9 b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
\end{align*}
\]

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.
7. Information Representation

7.1.5 Fractional Numbers

How positional number systems represent fractions was described in Section 7.1.2. 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 $Q_{m,n}$ 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 \( \pm 1 \).

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 \(+32767 (0.111111111111111)\).

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.

### 7.1.6 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 \(0 \leq \text{mantissa} < 1\). 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 7.2 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 \(M = 1 + f\)).

\[
\begin{array}{cccccc}
31 & 30 & \cdots & 23 & 22 & \cdots & 0 \\
\text{sign} & \text{exponent bits} & \text{fraction bits}
\end{array}
\]

Figure 7.2: 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 7.3. 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.

<table>
<thead>
<tr>
<th>63</th>
<th>62</th>
<th>···</th>
<th>52</th>
<th>51</th>
<th>···</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign</td>
<td>exponent bits</td>
<td>fraction bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.3: 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

$$(-1)^s \times 2^{-1022} \times f.$$
7.2 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.

7.2.1 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 7.4. 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 \text{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 \text{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 \text{0x07} (BEL) would ring a bell on the old physical teletype machine. Table 7.5 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 \text{ê} or \text{á}, nor can one put the tildes over an \text{n} such as \text{ñ}. Neither can one represent many common currency symbols such as \text{£}, \text{€}, or \text{¥}.
Table 7.4: Table of ASCII codes.

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

#### 7.2.2 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

---

Table 7.5: Control codes and descriptions.

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

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.
7.2. Text: Characters and Strings

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.

7.2.3 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 are 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.
7. Information Representation

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.

7.3 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 7.4.

![Figure 7.4: Sequence of spots that result when 0 represents a white spot and 1 represents a black spot.](image)

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 7.5.

![Figure 7.5: Image that results when spots are arranged in rows.](image)

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.
7.3. Images

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 [8], and images compressed using this technique normally are stored with a .jpg file extension.

7.3.1 Monochrome Images

In the example image of Figure 7.5, 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 a 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 7.6. 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 7.5; however, it still leaves quite a bit to be desired.

7.3.2 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 7.6 is shown in Figure 7.7.
7. Information Representation

Figure 7.6: Monochrome image that is 512 pixels tall and 512 pixels wide. (Photo courtesy Tracy L. Chamberlain, © 2013.)

Figure 7.7: Color image that is 512 pixels tall and 512 pixels wide. (Photo courtesy Tracy L. Chamberlain, © 2013.)
8 User Interaction
9 Computer Architecture and Assembly Language

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 examine the human-readable and -writable variation of the machine instructions, commonly called assembly language.

9.1 Basic Computer Architecture

A high-level view of the AVR microcontroller family computer architecture is shown in Figure 9.1. 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.

9.1.1 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, 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*.

### 9.1.2 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.

### 9.2 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 Atmel [1].

### 9.2.1 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”? 

---

1. [1] Instruction Set Manual from Atmel
9.2. Instruction Set Architecture (ISA)

More precisely, it comprises 8 individual status bits, each of which has its own dedicated function. Table 9.1 gives the meaning and shorthand label for each bit of the status register.

Table 9.1: Bits of status register SREG.

<table>
<thead>
<tr>
<th>Bit</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label</td>
<td>I</td>
<td>T</td>
<td>H</td>
<td>S</td>
<td>V</td>
<td>N</td>
<td>Z</td>
<td>C</td>
</tr>
<tr>
<td>Meaning</td>
<td>interrupt enable</td>
<td>transfer bit</td>
<td>half carry bit</td>
<td>sign overflow</td>
<td>signed flag</td>
<td>negative flag</td>
<td>zero flag</td>
<td>carry flag</td>
</tr>
</tbody>
</table>

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.

9.2.2 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.

**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 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 9.2.1 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 9.2 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 9.2.1. 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 timer 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
## Table 9.2: Peripherals accessible through the memory interface.

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>I/O Address</th>
<th>Label</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>–</td>
<td>r0</td>
<td>general purpose register r0</td>
</tr>
<tr>
<td>0x0001</td>
<td>–</td>
<td>r1</td>
<td>general purpose register r1</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>0x001E</td>
<td>–</td>
<td>r30</td>
<td>general purpose register r30</td>
</tr>
<tr>
<td>0x001F</td>
<td>–</td>
<td>r31</td>
<td>general purpose register r31</td>
</tr>
<tr>
<td>0x0020</td>
<td>0x00</td>
<td>–</td>
<td>reserved</td>
</tr>
<tr>
<td>0x0021</td>
<td>0x01</td>
<td>–</td>
<td>reserved</td>
</tr>
<tr>
<td>0x0022</td>
<td>–</td>
<td>–</td>
<td>reserved</td>
</tr>
<tr>
<td>0x0023</td>
<td>0x03</td>
<td>PINB</td>
<td>input pins port B</td>
</tr>
<tr>
<td>0x0024</td>
<td>0x04</td>
<td>DDRB</td>
<td>data direction register port B</td>
</tr>
<tr>
<td>0x0025</td>
<td>0x05</td>
<td>PORTB</td>
<td>data register port B</td>
</tr>
<tr>
<td>0x0026</td>
<td>0x06</td>
<td>PINC</td>
<td>input pins port C</td>
</tr>
<tr>
<td>0x0027</td>
<td>0x07</td>
<td>DDRC</td>
<td>data direction register port C</td>
</tr>
<tr>
<td>0x0028</td>
<td>0x08</td>
<td>PORTC</td>
<td>data register port C</td>
</tr>
<tr>
<td>0x0029</td>
<td>0x09</td>
<td>PIND</td>
<td>input pins port D</td>
</tr>
<tr>
<td>0x002A</td>
<td>0x0A</td>
<td>DDRD</td>
<td>data direction register port D</td>
</tr>
<tr>
<td>0x002B</td>
<td>0x0B</td>
<td>PORTD</td>
<td>data register port D</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>0x005D</td>
<td>0x3D</td>
<td>SPL</td>
<td>stack pointer (low byte)</td>
</tr>
<tr>
<td>0x005E</td>
<td>0x3E</td>
<td>SPH</td>
<td>stack pointer (high byte)</td>
</tr>
<tr>
<td>0x005F</td>
<td>0x3F</td>
<td>SREG</td>
<td>status register (see Table 9.1)</td>
</tr>
<tr>
<td>0x0060</td>
<td>–</td>
<td>WDTCSR</td>
<td>watchdog timer control register</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>0x0078</td>
<td>–</td>
<td>ADCL</td>
<td>analog-to-digital conv. register (low byte)</td>
</tr>
<tr>
<td>0x0079</td>
<td>–</td>
<td>ADCH</td>
<td>analog-to-digital conv. register (high byte)</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>0x00FF</td>
<td>–</td>
<td>–</td>
<td>reserved</td>
</tr>
</tbody>
</table>
access to the free-running counter used for timing that was described in Chapter 6 and control other aspects of the processor’s operation.

Memory Map

The memory model can be shown visually in a diagram known as a memory map. Figure 9.2 shows the memory map for the AVR microcontroller used in the Arduino Uno.

![Figure 9.2: AVR microcontroller memory map.](image)

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.

9.2.3 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:

\[
\text{add Rd, Rs}
\]

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 9.3 shows
Table 9.3: SREG bits effected by add instruction.

<table>
<thead>
<tr>
<th>Label</th>
<th>Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>half carry</td>
<td>set if carry from bit 3</td>
</tr>
<tr>
<td>S</td>
<td>sign bit</td>
<td>( N \oplus V ), for signed tests</td>
</tr>
<tr>
<td>V</td>
<td>signed overflow</td>
<td>set if two’s complement overflow</td>
</tr>
<tr>
<td>N</td>
<td>negative flag</td>
<td>equal to most significant bit of the result</td>
</tr>
<tr>
<td>Z</td>
<td>zero flag</td>
<td>set if result is 0x00</td>
</tr>
<tr>
<td>C</td>
<td>carry flag</td>
<td>set if carry from bit 7 (msb)</td>
</tr>
</tbody>
</table>

how they are effected. The remaining two bits in SREG are not changed by the instruction.

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

\[
\text{adc } Rd, Rs
\]

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:

\[
\begin{align*}
\text{add } r6,r8 \\
\text{adc } r7,r9
\end{align*}
\]

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,

\[
\text{inc } r17
\]

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

\[ Rd \leftarrow Rd \land Rs \]

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 \texttt{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

\begin{verbatim}
  cp   r3,r22
  brne loop
\end{verbatim}

first compares register \( r3 \) with \( r22 \), which will set the \( Z \) bit of SREG if \( r3 - r22 \) is equal to zero (i.e., if \( r3 \) and \( r22 \) are equal). The conditional branch instruction then branches to the program location labeled \( \texttt{loop} \) 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.,

\begin{verbatim}
  mov r23,r22
\end{verbatim}

copies the contents of \( r22 \) into \( r23 \). Note, \texttt{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:

\begin{verbatim}
  ldi Rd,k
\end{verbatim}
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:

\[
\text{lds } Rd, k
\]

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 \textit{addressing modes} and are described below.

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

\[
\text{sts } k, Rs
\]

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 9.2), there are also dedicated instructions for accessing the I/O registers in less time (and smaller instruction size) than load and store instructions. The \texttt{in} instruction supports reading from an I/O peripheral:

\[
\text{in } Rd, k
\]

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

\[
\text{out } k, Rs
\]

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 \texttt{nop} is “no operation” (it is an instruction that has no effect other than to take time to execute) and the \texttt{wdr} instruction is the watchdog timer reset instruction.

\textbf{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 ← 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:

```plaintext
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

```plaintext
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

```plaintext
ld Rd,Y+
```

for a load, and the syntax for a pre-decrement indirect access is

```plaintext
st -Z,Rs
```

for a store.

There are a few specialized addressing modes for program memory access; however, we will not discuss them here.
9.2.4 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.

9.3 Assembly Language and Assemblers

In the examples of instructions provided in Section 9.2.3 above, we actually cheated a bit. In the real machine, instructions are binary values (mostly 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 section, 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).

9.3.1 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 9.4. 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 in the table 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
follows in program memory).

Table 9.4: 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.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>wdr</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>inc Rd</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>inc r12</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ld Rd,X</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ld r6,X</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mov Rd,Rs</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>s</td>
<td>4</td>
<td>d</td>
<td>4</td>
<td>d</td>
<td>3</td>
<td>d</td>
<td>2</td>
<td>d</td>
<td>1</td>
</tr>
<tr>
<td>mov r3,r1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>add Rd,Rs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>s</td>
<td>4</td>
<td>d</td>
<td>4</td>
<td>d</td>
<td>3</td>
<td>d</td>
<td>2</td>
<td>d</td>
<td>1</td>
</tr>
<tr>
<td>add r7,r4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>subi Rd,k</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>k</td>
<td>7</td>
<td>k</td>
<td>6</td>
<td>k</td>
<td>5</td>
<td>k</td>
<td>4</td>
<td>d</td>
<td>3</td>
<td>d</td>
<td>2</td>
</tr>
<tr>
<td>subi r21,2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>lds Rd,k</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lds r8,0x100</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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 1d3d2d1d0. Restrictions like the one above are not uncommon 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 $+127$).

### 9.3.2 Assembly Language Instructions

In the code examples of Section 9.2.3 and in the left-most column of Table 9.4, we have been using the assembly language expressions for the instructions of interest. The general form for assembly language instructions is:
9. Computer Architecture and Assembly Language

**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.

### 9.3.3 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:

```assembly
ldi r3, 2
loop: add r7, r8
dec r3
brne loop
```

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.
9.3.4 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,

```assembly
.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

```assembly
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`.

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.

Data Section Pseudo-ops

The data section starts with the following pseudo-op:

```assembly
.data
```

which declares that the assembly language statements to follow are associated with the data memory (SRAM).

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.,

```assembly
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.

**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.

**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 to macros that are useful in this process, i.e.,

\[
\text{lo8(value)}
\]

takes a 16-bit value and returns the least significant 8 bits, while

\[
\text{hi8(value)}
\]

takes a 16-bit value and returns the most significant 8 bits. These macro can be very useful in address manipulation.

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:

\[
\text{ldi r26,lo8(array1)}
\text{ldi r27,hi8(array1)}
\text{ld r2,X}
\]

which loads the value stored at address `array1` into register `r2`. 
10 Computer to Computer Communication
11 Conclusions
A Languages
B Simple Introduction to Electricity
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)
C. Base Conversions

using decimal arithmetic then proceeds as follows:

\[
3e_{16} = 3 \cdot 256 + 14 \cdot 16 + 1 \\
= 768 + 224 + 1 \\
= 993_{10}.
\]

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
C.2. Convert Base A to Base B using Base A Math

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. \( \text{temp} \leftarrow \text{value to be converted} \)
   \( \text{result} \leftarrow \text{empty} \)

2. perform integer division \( \text{temp}/b \) resulting in quotient \( q \) and remainder \( r \)

3. prepend \( r \) (as an individual digit in base B) to the front of \( \text{result} \) (i.e., \( r \) is the new most significant digit of \( \text{result} \))

4. \( \text{temp} \leftarrow q \)

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) \( \text{temp} = 993 \) and \( \text{result} \) is empty

(2) divide \( 993/16 \), which gives quotient \( q = 62 \) and remainder \( r = 1 = 1_{16} \)

(3) \( \text{result} = 1_{16} \)

(4) \( \text{temp} = 62 \)

(5) \( q = 62 \), return to step 2

(2) divide \( 62/16 \), which gives \( q = 3 \) and \( r = 14 = e_{16} \)

(3) \( \text{result} = e_{16} \)
C. Base Conversions

(4) \( temp = 3 \)

(5) \( q = 3, \) return to step 2

(2) divide \( 3/16, \) which gives \( q = 0 \) and \( r = 3 = 3_{16} \)

(3) \( result = 3e_{16} \)

(4) \( temp = 0 \)

(5) \( q = 0, \) finished

At the end of the above procedure, \( result \) is \( 3e_{16}, \) 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) \( result = 02 \)

(4) \( temp = 67 \)

(5) \( q = 67, \) return to step 2

(2) divide \( 67/2, \) which gives \( q = 33 \) and \( r = 1 \)

(3) \( result = 102 \)

(4) \( temp = 33 \)

(5) \( q = 33, \) return to step 2

(2) divide \( 33/2, \) which gives \( q = 16 \) and \( r = 1 \)

(3) \( result = 1102 \)

(4) \( temp = 16 \)

(5) \( q = 16, \) return to step 2

(2) divide \( 16/2, \) which gives \( q = 8 \) and \( r = 0 \)

(3) \( result = 01102 \)

(4) \( temp = 8 \)
C.2. Convert Base A to Base B using Base A Math

(5) \( q = 8 \), return to step 2
(2) divide 8/2, which gives \( q = 4 \) and \( r = 0 \)
(3) \( \text{result} = 00110_2 \)
(4) \( \text{temp} = 4 \)
(5) \( q = 4 \), return to step 2
(2) divide 4/2, which gives \( q = 2 \) and \( r = 0 \)
(3) \( \text{result} = 000110_2 \)
(4) \( \text{temp} = 2 \)
(5) \( q = 2 \), return to step 2
(2) divide 2/2, which gives \( q = 1 \) and \( r = 0 \)
(3) \( \text{result} = 0000110_2 \)
(4) \( \text{temp} = 1 \)
(5) \( q = 1 \), return to step 2
(2) divide 1/2, which gives \( q = 0 \) and \( r = 1 \)
(3) \( \text{result} = 10000110_2 \)
(4) \( \text{temp} = 0 \)
(5) \( q = 0 \), finished

which gives \( \text{result} = 10000110_2 \). We can check this by converting from binary back into decimal:

\[
10000110_2 = 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \\
= 1 \cdot 128 + 0 \cdot 64 + 0 \cdot 32 + 0 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 1 \cdot 2 + 0 \\
= 128 + 4 + 2 \\
= 134_{10}.
\]

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


Index

A/D converter, 31
A/D counts, 31
addressing modes, 87
analog-to-digital converter, 31
analogReference(), 34–36
analogWrite(), 28
assembler, 90
assembly language, 7, 75, 90

Boolean, 3
Boolean algebra, 4

complex numbers, 51
counting numbers, 47
data section, 93
delay(), 43–45
delta time, 45
denormalized, 65
digitalRead(), 17, 81
digitalWrite(), 10, 43, 81
direct addressing, 88
directives, 93
double precision, 64
duty cycle, 26

excess notation, 58
exponent, 64

fetch-decode-execute cycle, 77
fixed point numbers, 62
floating point numbers, 63

hard real-time, 41
Harvard architecture, 76, 79
hex, 55
hexadecimal, 55
high-level language, 7

I/O bus, 80
I/O port, 81
I/O registers, 80, 87
IDE, 2
imaginary numbers, 51
immediate addressing, 88
index registers, 89
indirect addressing, 89
integers, 49
integrated development environment, 2
irrational numbers, 51
loop(), 2, 34

machine instructions, 75
machine language, 7, 75, 90
mantissa, 64
memory map, 83
microcontroller, 2, 75
microprocessor, 75
micros(), 43
millis(), 43
Index

NaN, 65
natural numbers, 48
normalized, 64

object code, 90
offset notation, 58
opcode, 77, 92
operands, 77, 92
operating modes, 90

peripheral, 80

pinMode(), 10, 17, 28, 81

pixel, 70

positional numbers, 47, 53
post-increment, 89
pre-decrement, 89
pseudo-operations, 93
pulse-width modulation, 26
PWM, 26

Q notation, 62

radix, 54
radix complement, 59
radix point, 54
rational numbers, 49
real numbers, 51
real-time, 41
register addressing, 88

setup(), 2

sign bit, 60
sign-magnitude, 58
single precision, 64

sketch, 2
soft real-time, 41

text section, 93

truth table, 4
two’s complement, 59

watchdog timer, 81, 87
whole numbers, 48