Chapter 15: Embedded Computing

Overview

Prerequisites:
- Knowledge of binary and hexadecimal numerical representations and conversions
- Knowledge of basic digital logic circuits
- Knowledge of basic circuitry

Objectives of Section 15.1:
- Understand high-level architecture of a generic computer
- Define elemental parts of an embedded computers, i.e., CPU, memory, I/O peripherals, buses
- Understand architecture and function of the CPU, memory, and I/O

Objectives of Section 15.2:
- Understand the organization of memory
- Describe how data is stored in little- and big-endian microprocessors
- Describe and understand various categories and types of memory

Objectives of Section 15.3:
- Understand capabilities of Arduino Uno
- Install open-source Arduino IDE and driver software
- Learn how to write Arduino sketches and upload code to Arduino Uno

Objectives of Section 15.4:
- Understand basic data types available on Arduino Uno
- Perform basic operations on data types using arithmetic operations on Arduino
- Create functions to simplify code and reduce repeated instructions
- Understand libraries and their functions on Arduino
- Control a servomotor using Arduino Uno board

S.N. Makarov et al., *Practical Electrical Engineering*, DOI 10.1007/978-3-319-21173-2_15

Objectives of Section 15.5:

- Understand and create code with conditional statements
- Understand and implement code with switch statements
- Be able to control loops of a sketch and incite repetition
- Use strings and arrays in a program where appropriate
- Print out messages to the serial monitor for debugging
- Understand interrupts and their advantages and disadvantages
- Be able to generate a square wave with a controllable duty cycle on a pin

Application examples:

- Servomotor control
- Emergency motor stop

Keywords:

CPUs: Real-time, Von Neumann/Princeton architecture, Hierarchy, ALU (arithmetic logic unit), CPU functions (fetch, decode, execute, write-back), Opcode, Operands, Instruction set, RISC (Reduced Instruction Set Computing), Byte, Address, Kilobyte, Megabyte, Gigabyte, Kilobinary, Megabinaries, Gigabinaries, Bus, Bus width, Address bus, Data bus, Rule of shared buses; **Memory**: Address, Big endian, Little endian, Volatile, Nonvolatile, RAM (random access memory), ROM (read-only memory), EEPROM (electronically erasable programmable read-only memory), Flash memory, PROM (programmable read-only memory), EPROM (erasable programmable read-only memory), Address space; **Arduino**: Arduino, Arduino Uno, IDE (integrated development environment), File menu, Edit menu, Sketch menu, Tools menu, Functions, Return value, Void, Setup(), Loop(), Comment, Primitive data types, Int (integer), Float (floating point), Double (double precision), Boolean (true or false), Char (character), Variables, Type specifier, Declaration, Assignment statement, Typecasting, Function header, Body of the function, Argument, Arrays, Strings, Library, Encapsulation, Access functions, Header file, Function prototypes, Script file, Servo library, Servomotor, Servo object, Conditional statements, State programming, Switch statement, For loop, Pseudo-code, Infinite loop, Increment operator, Decrement operator, While loop, Element, Index, Serial communication, Baud rate, Polling, Interrupts, ISR (interrupt service routine), Interrupt queue, Debounced

Section 15.1 Architecture of Microcontrollers

Embedded computers are literally everywhere in modern life. Embedded computing is the incorporation of computing devices like microcontrollers into the design of a product or larger system that is not itself a computer. On any given day, we interact with and depend on dozens of small computers to make coffee, run cell phones, take pictures, run the dishwasher, control elevators, stop the car, and so on. For every PC-style computer made each year, there are approximately 50–100 embedded computer devices produced. Consider the contents of an average student's backpack. It is likely to contain a notebook-style personal computer (PC). However, it is just as likely to contain a digital music player, a cell phone, a handheld graphing scientific calculator, a keyless entry car key, and maybe an e-book reader! Each of these devices relies on the computing capability embedded within it.

A key feature of an embedded computer is that it typically only performs a single function or small set of very tightly coupled functions. It is not a general-purpose device like a PC. Another feature of an embedded system is that it can enable operation of a complex function by a nonexpert. For example, one can live a full and complete life without knowing the details of the Moving Pictures Expert Group Audio Layer-3 encoding standard (i.e., MPEG-3). To use a digital music player, we simply press the play button. Embedded computers are often resource constrained and must exhibit very high reliability. They must perform their assigned task in *real time* or very close to real time and must work flawlessly for years powered only by a small battery. To achieve these goals, embedded computers are usually a mix of tightly integrated hardware and software. An engineer must understand and appreciate both the hardware and the software aspects of embedded computers to use them effectively. This chapter presents a high-level overview of microcontrollers and how they can be interfaced with control hardware components in an electronic circuit.

15.1.1 A Generic Microcontroller

Figure 15.1 is a block diagram view of a generic computer. Any computer system, whether large or small, will contain three functional subblocks: the central processing unit (CPU), memory, and input/output (or I/O) devices. The general architecture in Fig. 15.1 is called the *Von Neumann* or *Princeton architecture*, and it dates from 1946. In earlier mainframe and PC-style computers, the CPU, memory, and I/O devices were each implemented as separate circuits often on separate circuit boards and were connected through wire buses. Modern microcontrollers are complete computer systems implemented within a single integrated circuit (IC).

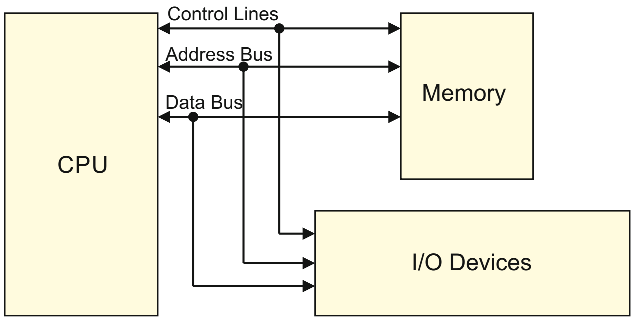


Fig. 15.1. Architecture of a generic computer.

The *hierarchy* of hardware and software for general-purpose computing (e.g., on a PC) is shown in Table 15.1.

Table 15.1. Hierarchy of hardware and software for general-purpose computing.

Applications (MultiSim, MATLAB, Google Earth, etc.)
Operating system (Windows 10, Linux, or Mac OS X)
Hardware Abstraction Layer (device drivers, basic I/O system—BIOS)
Hardware (CPU, memory, I/O devices)

This hierarchy is simplified in an embedded system where a single custom application interfaces directly with the microcontroller's hardware—see Table 15.2.

Table 15.2. Hierarchy of an embedded system.

Application (single, custom application)
Hardware (CPU, memory, I/O devices)

15.1.2 Central Processing Unit

The CPU is the "brain" of a computer. The CPU contains the control unit, the *arithmetic logic unit* (ALU), and bus interface circuitry as seen in Fig. 15.2. It controls the operation of memory and the I/O devices and executes program instructions. Conceptually, a CPU performs four functions: *fetch*, *decode*, *execute*, and *write-back*. First the CPU fetches an instruction from the part of memory where the program is stored. An instruction is simply a multi-byte binary code. The instruction is then decoded within the control unit to extract its *opcode* (operational code which specifies operations to be performed) and the *operands* (quantities on which operations are performed). The sample format of this process can be seen in Fig. 15.3. A CPU can only understand and execute a fixed number of operations which is called its *instruction set*.

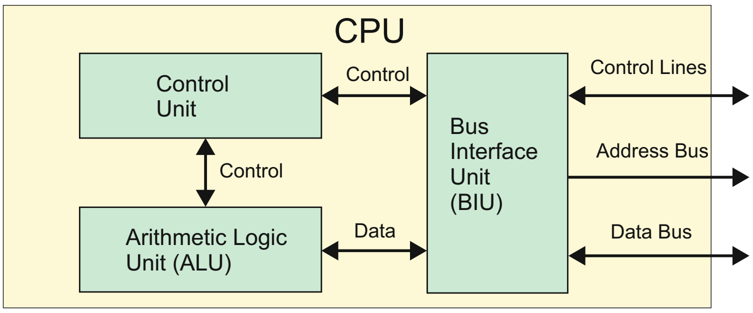


Fig. 15.2. Block diagram of a CPU.

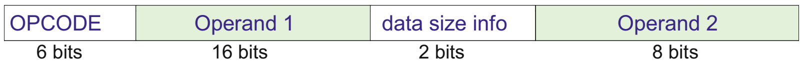


Fig. 15.3. Notional format of a 4-byte machine language instruction.

Most small microcontrollers used in embedded computing today are *RISC* (Reduced Instruction Set Computing) devices and only have about 20 or 30 instructions. Each of the instructions is represented by a unique binary opcode. Execution of the instruction generally involves the *arithmetic logic unit* (ALU). The ALU contains digital circuitry to perform binary arithmetic functions like addition and subtraction as well as bitwise logic functions like AND, OR, and NOT on operands supplied in the instruction. Many, if not most, small microcontrollers do not have multiply or divide instructions meaning they have no digital circuitry for multiplication or division in their ALU. When a program requires multiplication or division, those operations are implemented *in software* using repeated addition or subtraction instructions.

15.1.3 Memory

Memory is where both program and any data used or acquired during program execution is stored. Memory is an array of sequential storage locations. Each location typically holds 8 bits or 1 *byte*. Additionally, each memory location has an *address* (expressed to the CPU as a binary code) that uniquely identifies it. The amount of memory is specified in terms of bytes with the total number of bytes always being a power of 2. For example, $2^{10} = 1024$ bytes is commonly called a 1 *kilobyte* = 1 KB. Similarly, 1 *megabyte* (MB) is 2^{20} bytes and 1 *gigabyte* (GB) is 2^{30} bytes. Strictly speaking the quantity 2^{10} is defined as a *kilobinary* (Ki) and 2^{20} and 2^{30} are *megabinaries* and *gigabinaries* (Mi and Gi), respectively. However, manufacturers continue to regularly use the labels KB, MB, and GB for these quantities. The system designer need to consult the datasheets to determine how a given manufacture is defining memory capacity. Small microcontrollers may only have 2–16 KiB of memory, a more capable system may have 64 KiB to 1 MiB, and the most powerful embedded processors can have over 1 GiB of on-chip memory.

15.1.4 Input and Output Devices

The I/O devices, sometimes called peripherals, are how the CPU interfaces to the outside world. The complement of I/O devices varies between many different microcontrollers available. However, most microcontrollers will have at least one analog-to-digital converter to take measurements from analog devices like thermistors, pressure sensors, and accelerometers; one digital timer block to time events; and a one serial interface to communicate serial data to the outside world. Other common peripherals include digital-to-analog converters (DACs), comparators, and digital circuits for PWM generation.

15.1.5 Timers

Unlike PCs or other computers, microcontrollers have no access to a time reference. The only unit of time an embedded controller actually knows is the period of its clock signal:

$$T_{\text{clk}} = 1/f_{\text{clk}} \tag{15.1}$$

All other units of time like milliseconds or minutes must be generated by counting clock periods. Timers are digital circuit block that count the rising (or falling) edges of a clock signal. Virtually all microcontrollers have an on-chip timer and most have 2 or more.

Exercise 15.1: For an Arduino Uno running at a 16-MHz clock speed, how many rising or falling edges of the clock signal must be counted to measure the following times: (A) 37 ms, (B) 255 μs, and (C) 469 ns.

Answer: Simply multiply the clock frequency by the desired time.

A. Number of edges = 0.037 s * 16 MHz = 592,000 edges
B. Number of edges = 0.000255 s * 16 MHz = 4080 edges
C. Number of edges = 0.000000469 s * 16 MHz ~= 7 edges

An important note is that for 469 ns, a total of 7.5 edges must actually be counted, since half edges cannot be counted. This shows that there is a resolution of 62 ns for this clock in theory, but in practice this resolution will be much less precise.

15.1.6 Buses

A *bus* is the general name given to a collection of wires that make multiple connections *in parallel*. The number of wires bundled together is called the *bus width*. In digital circuits, the bus width also refers to the number of bits in a word. For example, transferring a byte from memory to the CPU requires eight parallel electrical connections, one for each bit. These eight parallel connections are an 8-bit bus. Instead of showing all eight connections in circuit drawings, buses are drawn as a single line with a slash—see Fig. 15.4. Inside a microcontroller, the buses that connect the CPU, memory, and peripherals are not wires but minute interconnections less than micrometer thick. Usually there will be a group of several control lines that the CPU uses to initiate operations by the memory or peripherals. There will also be an *address bus* which used by the CPU to specify where data should be read from or written to and a *data bus* which conveys binary data between the CPU and memory or between the CPU and the I/O devices. Notice in Fig. 15.1 that a single bidirectional data bus goes between memory, the CPU, and the I/O peripherals. It is the control lines from the CPU that establish who has control of the bus. Only one device is allowed to place data on the data bus at any given time. The fundamental *rule of shared buses* is that there can be only one bus driver.

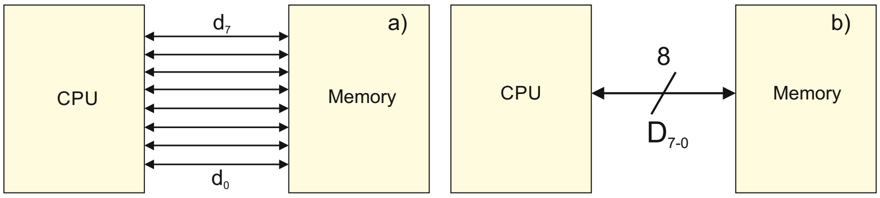


Fig. 15.4. (**a**) Explicitly showing all eight connections required to transfer 1 byte makes schematics cluttered; (**b**) multi-bit buses are drawn as a single, thicker line with bus width indicated above.

15.1.7 Universal Synchronous/Asynchronous Receiver/Transmitter (USART)

Parallel Data Transmission

Within a microcontroller, data is passed along parallel interfaces (buses) between the CPU and memory or between the CPU and I/O peripherals. However, parallel data transmission is not practical for the "long" haul (i.e., off chip) for several reasons. Chief among them are that the length of parallel links is usually limited and that there simply aren't enough pins available on a microcontroller's package to support parallel interfaces.

Serial Data Transmission

In serial communications, the bits that comprise a binary data word are sent sequentially along a single data line. Serial data transfer is typically slower than parallel data transfer but a minimal serial link requires only two data lines, transmit (TX) and receive (RX), regardless of word size. Also, there are serial devices available to communicate over a great range of distances. Serial data links can be made to communicate among integrated circuits on the same printed circuit board or made to communicate over many kilometers through an RF modem or satellite phone.

USART

A USART bridges the gap between the CPU and external serial devices by acting as a parallel-to-serial and serial-to-parallel translator. It converts parallel data from memory or CPU registers to serial format and then transmits the data to an external serial device. It also accepts serial data from external source(s) and converts it to parallel format so that the data may be read by the CPU or stored in memory.

Exercise 15.2: Outline asynchronous and synchronous serial communications.

Solution: In asynchronous communications, both the transmit device and receive device run using their own clock frequencies. In synchronous communications, both devices run at the same clock cycle. There is a data line that supplies the clock signal from the transmit device to any receiving devices to ensure the communication occurs smoothly and without errors.

Section 15.2 Memory

15.2.1 Organization of Memory

Memory is a group of sequential locations where binary data is stored. Typically, in a microcontroller, each memory location holds 8 bits or 1 byte of data. Each location in memory has a unique *address* which the CPU uses to read to and write from that location—see Fig. 15.5 as an example. You may think of each memory location as mailbox. The address number on the outside of the mailbox has nothing to do with what is actually in the mailbox; it just identifies where the mail is to be placed. Memory addresses serve the same purpose. The address identifies the location at which some data is stored but does not convey any information about what that data may be. Frequently, a single word of data is longer than 1 byte. For example, a 16-bit binary number would require 2 bytes of memory to store. Similarly, the output of a 12-bit ADC would require 2 bytes of storage even though the most significant 4 bits of the second byte are not used. Multi-byte data is stored in successive memory locations and the address associated with the whole word is the address where the first byte from the word is stored.

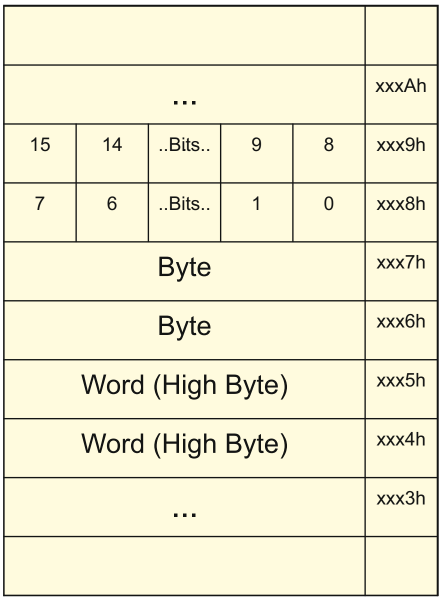


Fig. 15.5. Byte storage in memory.

The question is which byte is stored first in memory, the byte containing the most significant 8 bits of the word or the byte containing the least significant bits? For that matter, within a byte which bit is the most significant and which is the least? By convention, the bits within a byte are labeled left to right as bits 7 through 0 with bit 7 representing the most significant bit and bit 0 being the least significant.

This convention aligns with the powers of 2 that would be represented in an 8-bit binary number. As an example, we present the byte that represents the decimal number 149 in Fig. 15.6.

$$\underset{\substack{\Uparrow \\ \text{bit}_7\ \text{MSB}}}{1}001\ 010\underset{\substack{\Uparrow \\ \text{bit}_0\ \text{LSB}}}{1}\text{b} = 1\text{x}2^7 + 0\text{x}2^6 + 0\text{x}2^5 + 1\text{x}2^4 + 0\text{x}2^3 + 1\text{x}2^2 + 0\text{x}2^1 + 1\text{x}2^0$$

Fig. 15.6. The binary representation of the decimal number 149.

Determining the order in which the bytes of multi-byte data word are stored is more confusing. One would think that, with almost 70 years into the age of computing, the order of storage of multi-byte data words would be firmly established. Unfortunately, this is not the case. There are two possible storage conventions; the most significant byte can be stored first with the less significant bytes following in successive locations, or the least significant byte can be stored first with the more significant bytes following. The first method is called *big-endian* storage, and the second method is called *little-endian* storage. Neither endian convention is technically superior to the other and both are firmly established and used by various families of microcontrollers. The names big and little endian actually come from Jonathan Swift's *Gulliver's Travels* in which the neighboring kingdoms came to blows over which end of a hard-boiled egg to open first. Endian battles in computing are nearly as hotly contested! Typically, endian-ness is a basic design decision made early in the development of a family of CPUs and CPUs are either big endian or they are little endian (however, FreeScale does make endian-selectable processors). System designers need to know which convention is used to store their data in order to interpret the contents of memory and implement long word data types.

Example 15.1: How would the value 72468 be stored in memory starting at address 0200h by a little-endian microprocessor and by a big-endian microprocessor?

Solution: First convert 72468 to binary: 72468 = 10001101100010100b. This binary number is 17 bits long, but data representations available to computers are always multiple of 8 bits (i.e., of 1 byte). Common data word sizes for microcontrollers are 8 bits, 16 bits, and 32 bits. Since this number requires more than 16 bits to represent, we must use a 32-bit (4 bytes) data type with the most significant bits all equal to 0. It is easier to then write the number as its equivalent hexadecimal value:

$$72468 \text{ decimal} = 00000000\ 00000001\ 00011011\ 00010100\text{b} = 00\ 01\ 1\text{B}\ 14\text{h} \tag{15.2}$$

On a little-endian processor, the least significant byte (the little end) is placed in the assigned address and the more significant bytes are placed in successive locations.

However, on a big-endian processor, the most significant byte (the big end) is stored first and the more significant bytes are placed in successive slocations. Table 15.3 illustrates the corresponding data storage.

Example 15.1 (cont.):

Table 15.3. Data storage methods.

Little endian		Big endian	
Address	Byte value	Address	Byte value
0204h	. . .	0204h	. . .
0203h	00h	0203h	14h
0202h	01h	0202h	1Bh
0201h	1Bh	0201h	01h
0200h	14h	0200h	00h
01FFh	. . .	01FFh	. . .

15.2.2 Types of Memory

Volatile Memory

There are two main classifications of computer memory, *volatile* and *nonvolatile*. All computer systems, including microcontrollers, will have a mix of these two memory types. Volatile memory can be both read from and written to under program control but all of its contents are immediately lost when power is removed. In most computer systems, the volatile memory is called RAM, for *random access memory.* The term random access means the data at any address can be accessed in any order. This a throwback to the early days of computing where data storage devices like magnetic tape drives had to be accessed sequentially as the tape spun to a given address. Strictly speaking, all on-chip (or IC) memory, whether volatile or not, is random access but the name RAM has become specifically associated with volatile memory.

Nonvolatile Memory

Nonvolatile memory retains its contents when power is removed but it is generally *read-only memory* (ROM). This means that while nonvolatile memory can always be read from under program control, it cannot, in general, be written to. The exception is *electronically erasable programmable read-only memory* (EEPROM), such as commonly used *flash memory*, which can be written to during program execution. A valid question might be: why bother with volatile RAM? Why not just always use nonvolatile flash memory? The reason is that reading from and especially writing to EEPROM takes longer than accessing RAM. The erasure and writing process for flash is significantly more involved than writing to RAM. Flash must be erased in blocks, e.g., 512 bits at a time, before it can be written. Other types of ROM, like *programmable read-only memory* (PROM) and *erasable programmable read-only memory* (EPROM), cannot be written to at all during program execution. The contents of these types of memory are fixed and programmed in using a separate, off-line process (i.e., a PROM programmer).

The two types of memory serve different purposes within the computer. Nonvolatile memory is where the microcontroller's program is stored (code memory), while volatile memory is where data used or generated during program execution is kept (data memory). By convention, nonvolatile memory generally occupies the higher memory addresses, and RAM occupies the lower addresses. Often, the registers that control the I/O devices are also mapped to the lower memory addresses. Figure 15.7 illustrates a notional memory map for a small microcontroller with a total of 64 KB of memory. The mix of RAM and ROM present in a computer depends on the application of the processor. General-purpose PCs have much more volatile memory (i.e., several GB of RAM) than nonvolatile memory. Most small microcontrollers, on the other hand, may have only 512–4096 bits of RAM and 16K to 64 KB of flash or ROM.

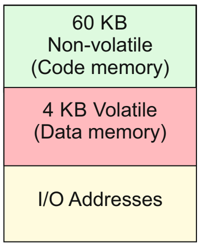


Fig. 15.7. A notional memory map for a small microcontroller with a total of 64 KB of memory.

Exercise 15.3: How wide must the address bus be to access 64 KB of memory?

Answer: Sixty-four kilobytes of memory contains $64 \times 2^{10} = 2^6 \times 2^{10} = 2^{16}$ bytes. Sixteen bits are required to label these 2^{16} locations. Usually memory addresses are expressed in hex. The labels for the *address space* of this microcontroller would run from 0000h to FFFFh.

15.2.3 Flash Memory in Embedded Devices

The advent of inexpensive flash memory has had a tremendous impact on the design and implementation of embedded systems. As mentioned above, many families of microcontrollers use flash as their nonvolatile memory. The primary purpose of these on-chip blocks of flash is to act as code memory. However, some embedded systems also collect measurements from I/O devices and must store this data permenantly. Data loggers are devices that record measurements from a sensor over time and store the data. Depending on the operational life of the system and the amount of data collected, part of the on-chip flash of the microcontroller could be used to save the data. More often, an interface to a removable flash device like a secure digital (SD) card is implemented using the serial commuications peripheral. Then, a small, inexpensive microcontroller with only a few KB of internal memory can save gigabytes of data to the external flash card. The SD device can then be removed and connected to a PC for data review and analysis.

Section 15.3 Arduino Uno: An Embedded Microcontroller

15.3.1 What Is Arduino?

The *Arduino* is an open-source platform designed to be a low-cost, flexible, and easy-to-program embedded microprocessor. Technically, the word Arduino refers to the name of the hardware, while the actual embedded microprocessors are named differently. The *Arduino Uno*, for example, is one of the lowest-cost and easily accessible variations of the Arduino boards. For the purpose of this textbook chapter, the examples and programming will be done on the Arduino Uno. The Arduino Uno's brain (the R3 Uno board) is derived from an ATmega328 microcontroller. While the board is powered by a supply voltage of 7–12 V, the actual board itself uses 5-V logic levels and features a 10-bit analog-to-digital convertor (ADC). The ATmega328 chip embedded into the board gives the Uno 14 programmable digital IO pins and 6 analog pins. The board also features 32 KB of flash storage (with a dedicated 0.5 KB for the bootloader to load the Arduino code), 1 KB of EEPROM, and 2 KB of SRAM. The Arduino Uno board also runs at a clock speed of 16 MHz.

The Uno is surprisingly low cost for the actual board and the onboard microcontroller. Besides the Uno, there are many other variations of Arduino boards which are tailored to certain applications, such as the Arduino LilyPad for being sewn on clothing, the Arduino Robot board for robotics, and the Arduino Esplora for facilitating development of videogame controllers. One feature about all of the Arduino boards is that the Arduino software to program and compile code for the boards is completely free online. This allows for easy program creation. Also available online is a reference for all the syntax for programs and countless examples provided by the makers of the Arduino and the community as a whole; fostering a sense of creativity and creating a knowledge base for projects, tutorials, and code examples.

15.3.2 Arduino IDE

Now with a better understanding of what the Arduino is, the most logical question is how does one interface with the Arduino? This is accomplished through the use of the USB connection on the board. When the Uno is connected to the computer, the computer attempts to install the driver but will most likely fail in the process. This is normal as the driver required to run the Arduino must be installed with the *integrated development environment (IDE)* supplied by the makers of the Arduino board. An IDE is a program designed to aid in the programming of a certain language by providing a visual representation of a centralized collection of coding resources and files. The Arduino IDE provides a program editor, compiler, and uploading tool for the Arduino. The Arduino IDE is essentially an IDE for the C language, with a few prewritten libraries.

Example 15.2: Describe installation steps for the Arduino IDE.

Solution: This IDE can be downloaded directly from the Arduino home page. The builds are offered for Windows, OS X, and Linux. At the time of writing, the most recent stable version of the Arduino IDE is Arduino 1.0.5. For the purpose of this textbook, the installation process for Windows will be used. The Arduino distribution comes in two flavors: the Windows installer and the Windows ZIP file. The Windows installer is an .exe file which runs and installs the Arduino IDE on the computer. The Windows ZIP file is a compressed archive file which contains all the tools and packages necessary to run the Arduino IDE without installing it on the computer. This means that the Arduino environment could be "installed" on a flash drive by simply unzipping the contents of the ZIP file to the flash drive and then running the Arduino executable to start the IDE. Once the Windows installer has been downloaded and run, a splash screen appears which prompts the user to accept a license agreement. This license agreement basically outlines the terms and conditions of fair use of the Arduino software and how the code is open source and modifiable. After agreeing to this license, a screen showing the installation options appears. Select and install all the components listed in this screen. During the installation process, Windows may generate a warning complaining of the authenticity of the Arduino USB driver, but simply click "install this driver software anyways."

At this point, the Arduino can be plugged into the computer, and the computer should automatically recognize the device and install the drivers appropriately. If the installation was a success, a message will be shown that states the hardware has been recognized and installed. This message also shows the serial COM (communications port) that the Arduino will be using to send and receive data from the computer. This port must be selected within the Arduino IDE to ensure the Uno works properly.

15.3.3 Getting Started with Arduino IDE

The Arduino 1.0.5 IDE contains several notable features shown in Fig. 15.8. The first indicated item is the verify button. This checks the syntax and keyword usage and then compiles the code to a usable format for the Arduino. Item 2 is the upload button which compiles the code again and uploads it to the Arduino. Item 3 shows the name of the current "sketch" that the IDE is processing. A "sketch" is a code file (a .ino file) that the Arduino understands (essentially equivalent to a .c file in C programming). Item 4 is the button to create a new sketch file. Item 5 is the open sketch button. Item 6 is the save button. Item 7 is the serial monitor which is the console that shows the communication between the Arduino and the computer. Item 8 is the button that allows for the creation of tabs in the Arduino IDE to break up the code. Item 9 is the scripting area. Item 10 is the line number. Item 11 is the console output for showing the progress of compiling and uploading and also displays errors in the code caught by the compiler. Item 12 shows the current board being programmed and the serial port the board uses.

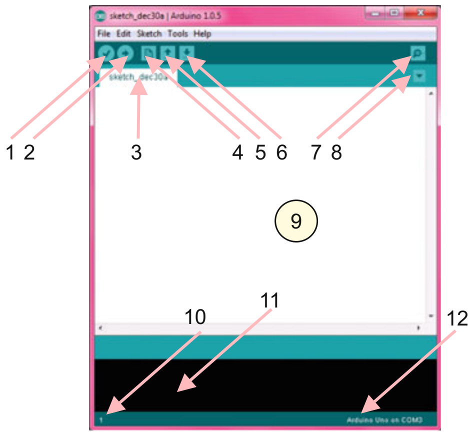


Fig. 15.8. The various parts of the Arduino IDE environment.

The IDE also features several menu options that aid the coding process. One of the first options is the ability to open examples that are preinstalled with the IDE. These can be accessed under the *File menu* → *Examples*. It is highly recommended to take a look at these examples as they provide how-to's on the various aspects of Arduino syntax and interfacing various external circuitry elements. The *File menu* also contains the various options to deal with file creation, saving, and printing. The *Edit menu* contains the basic tools to copy, paste, add indentation to the code, and find words and characters in the code. The *Sketch menu* contains options to add files to the current sketch as well as import a library. Libraries in the Arduino IDE will be discussed in greater detail later in this section. The *Tools menu* offers several resources for uploading and storing Arduino code. The most important options in the *Tools menu* are the Board and Serial Port options. The Board option lists all the various Arduino boards that have been created. From this menu, the correct board must be selected for programming purposes (simply select the Arduino Uno option for the purpose of this textbook). Two of the most useful features of the IDE help menu are the Reference and Find in Reference options under this menu. The Reference option brings up the Arduino reference page online that details the basic reserved words and functions in the Arduino language. Additionally, the Find in Reference option allows for searching of the online Reference page by selecting a word typed into the scripting area.

15.3.4 Arduino Language, Program Storage, and Basic Program Setup

In order for a microprocessor to understand instructions, it must have some general language and compiler to support those instructions. In the case of the Arduino, the language that is used to program the boards is essentially the C language which was developed by Brian W. Kernighan and Dennis M. Ritchie back in the 1970s. The actual boards execute machine code that is translated into this form by the compiler, but the scripting is all in the C language. The Arduino IDE introduced at the beginning of this section is essentially a specialized C programming environment with the specialized Arduino compiler (the AVR-GCC compiler for the ATmega328), which translates the code from the sketch files into the machine instructions which are then sent via USB to the Arduino. Once the code is uploaded to the board, the program is stored into flash memory on the Arduino board. This allows for quick access of the code with minimal space considerations. After the code is uploaded to the Arduino, the Arduino will start to run the code as written and will loop through the code while power is attached. When the Arduino is connected to the computer, the Arduino also behaves in the same fashion.

Most Basic Arduino Program

In order to get the Arduino Uno running with just the bare-bones minimum, only several lines of code need to be typed into the IDE. These lines of code can be seen in Fig. 15.9.

```
void setup(){
}
void loop() {
}
```

Fig. 15.9. The bare-bones minimum for an Arduino sketch.

Figure 15.9 shows the most basic Arduino program (sketch) that will run on any Arduino board. The actual executable code in Fig. 15.9 contains two *functions*. A function in the Arduino language is identical to a function in the C language and is a segment of code that is called and executed. This segment of code can have a *return value* or may not return a value (*void*). The first function in this basic code is the *setup*() function. This function is called once when the board receives power. Any initializations of variables or other functions that must be run before the code starts executing the loop should be placed here. After the setup function finishes, the *loop*() function is essentially called repeatedly while the board is powered. This loop function is where the Arduino program does the "thinking" and computing and is where the logic of the program is executed. Other function calls, variable declaration, variable assignment, and looping may be executed within this loop function.

Example 15.3: Relate Arduino code in Fig. 15.9 to the corresponding C code.

Solution: Those familiar with the C language should recognize the lack of a main() function in the Arduino code. The user of the Arduino does not have to worry about the main function but rather just the setup and loop functions for simplicity.

Example 15.3 (cont.):

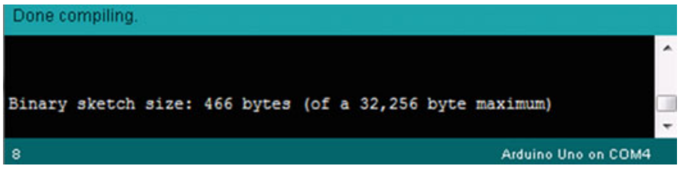
The general idea of how the Arduino code in Fig. 15.9 translates to C code can be seen in Fig. 15.10. The double slashes (//) indicate the start of a *comment*: a portion of the code that is not compiled and not executed but serves to document the code.

```
int main()
{
    //initializations of hardware
setup();  //call the setup function once
while(1) //loop forever
{
    loop(); //call the loop function forever
    //wait a little bit of time
}
return 0; //no errors
}
```

Fig. 15.10. The translation of the basic Arduino program code to C code.

15.3.5 Compiling and Uploading Code to Arduino Uno

Once the program is written in the IDE, the verify button can be clicked to check the syntax of the program. This will simply compile the code and check for errors in the code. If any errors are encountered in the compiling of the program, those errors will be shown below the scripting area. The user can then debug the program by using the information from the compiler to create code that can be compiled. For the basic code shown in Fig. 15.9, a screenshot of the output from the compiler can be seen in Fig. 15.11.



```
Done compiling.

Binary sketch size: 466 bytes (of a 32,256 byte maximum)

8                                         Arduino Uno on COM4
```

Fig. 15.11. The compiled basic code for the Arduino.

The output console in Fig. 15.11 shows a successful compiling of the basic .ino sketch and shows the sketch size (466 bytes). The maximum sketch size is 32,256 bytes which corresponds to most of the 32 Kb of flash memory on the Arduino Uno (there is some of this flash storage taken up by the boot loader which is used to load the software onto the Arduino). During the upload or compiling process, a small progress indication bar appears above the console output on the right side of the IDE. Once the upload finishes, several LEDs on the Arduino will flash and then the Arduino will begin executing the code that was uploaded to the board. In this case of the program in Fig. 15.9, the Arduino will not show any signs of activity on any of the pins. This is because no instructions have been added yet.

Section 15.4 Basic Arduino Syntax

15.4.1 Data Types

The most basic aspects of any programming language are the *primitive data types*. Primitive data types are the basic storage mechanisms for the language. These generally consist of integers (*int*), decimal or floating point numbers (*float*), double-precision floating point numbers (*double*), logic of true or false (*Boolean*), and single characters (*char*). These basic data types hold the information of the program and allow the programmer to create *variables* that change value. The complete listing of data types supported by the Arduino can be found on the Arduino reference page. The various data types listed above all have their own reserved word in the Arduino language called a *type specifier* that tells the compiler that a certain variable or constant is of a certain data type (such as an integer with the type specifier *int*).

In order for a data type to be implemented in Arduino as a variable or as a constant, the type specifier must be listed followed by the name of the data type. Figure 15.12 shows the creation or *declaration* of several variables and constants within the Arduino IDE.

```
Line 1. const int MOTOR_PIN = 9;
Line 2. const float KG_TO_LBS = 2.2;
Line 3. boolean is_stop_button_pressed = false;
Line 4. int button_presses = 0;
Line 5. int current_mode, num_switches = 2;
Line 6. float foo = 17.77;
Line 7. double conversion_factor = 4617.889;
Line 8. char check_letter = 'a';
```

Fig. 15.12. Declaring and initializing several variables and a constant.

All of the lines in Fig. 15.12 can be directly typed into the Arduino IDE and compiled without an issue. The important note is that all the declarations and initializations must be followed by a semicolon. Line 1 in Fig. 15.12 declares a fixed integer whose value cannot be changed at runtime of the code. The general naming convention of a constant is all capital letters with underscores between words.

Exercise 15.4: Declare several variables to store the following values:
true, 87, 3.14, ‘k’, ‘6’

Answer: Simply create the data types by declaring and naming the variables:

```
boolean is_robot_on = true;
int my_state = 87;
float pi_approximated = 3.14;
char my_letter = ‘k’, still_a_char = ‘6’;
```

15.4.2 Assignment Statements and Their Features

The code in Fig. 15.12 sets the integer constant `MOTOR_PIN` to have a value of 9 using the *assignment statement*. This value can be referenced later in the code by simply typing `MOTOR_PIN` instead of having to remember the value 9. This also aids in reusability of the code as the programmer can simply change the value of `MOTOR_PIN` at the declaration and this change will percolate through the code where the name was used (instead of having to change all the places where 9 was used in code). The constant identifier in front of the type identifier tells the compiler that the value of `MOTOR_PIN` should not change. The compiler will then generate a warning if future code segments try to alter the value of `MOTOR_PIN`. This allows for basic error avoidance.

Line 2 creates a constant floating point value of 2.2 that is called `KG_TO_LBS`. This name could be used in the code by simply writing `KG_TO_LBS`. Line 3 declares a Boolean variable called `is_stop_button_pressed` that can only take on the values true or false. This type of variable is only stored in 1 byte of data (8 bits). Line 4 declares a new integer variable called `button_presses` and sets its value to 0. This is an example of a single declaration. The declaration of an integer sets aside 2 bytes of memory on the Arduino Uno. The next line shows a double declaration of integer variables. Line 5 declares two new integer variables `current_mode` and `num_switches`. A note about these two variables is that while `num_switches` contains the value 2, `current_mode` has not been initialized. The value will most likely be 0, but this is not guaranteed. Thus, the safest course of action is to initialize the variable to a known value as otherwise the variable uses whatever was last in the memory location where the variable was stored. For example, as this code stands now, `num_switches` will only take on the value of 2 when the code is first uploaded to the Arduino or when power is applied to the Arduino (either externally or with the USB).

Line 6 declares and initializes a floating point variable called `foo` to have the value of 17.77. This value could be changed by typing `foo=9.83`; or by a more detailed assignment statement. Floating points on the Arduino are stored in 4 bytes as are doubles. Thus on the Arduino, floats and doubles can be considered the same. In particular, line 7 is equivalent to `float conversion_factor = 4617.88`;. An interesting note about floating point values and integer values is that an integer value may be converted to a floating point automatically if for instance the following line of code is written:

```
float needed_value = 9; //stores the value 9.0 in needed value        (15.3)
```

If the opposite is attempted:

```
int not_a_float = 7.77; //try to store 7.77 in an int                 (15.4)
```

then a problem exists because a floating point value (with numbers to the right of the decimal point) is trying to be stored an integer with a lesser precision. How does the Arduino handle this conundrum? One could say rounding would solve this, but the

Arduino simply truncates the value *without* rounding. The compiler will show a warning if this occurs by in the code during compile time. If this occurs at runtime, however, this can lead to a difficult bug to track. In order to avoid the warning from the compiler and to show that the operation of storing a floating point value into an integer value is correct, the code `(int)` must be added to the operation:

$$\texttt{int not\_a\_float} = (\texttt{int})\,\texttt{7.77;} \tag{15.5}$$

The line avoids the warning potentially issued by the compiler and the value of 7 will be stored into the variable `not_a_float`. This operation is referred to as *typecasting* and can be applied to most data types. Obviously, this typecasting only makes sense where the two data types are compatible. The type that a certain operation is being typecast into is included in the parentheses.

Exercise 15.5: A variable `distance` holds the value of the distance from a sensor connected to the Arduino. The distance is originally stored as an integer value, but later it becomes necessary that the distance has greater precision. How can this be achieved? Write code that converts the distance to the various data types.

Answer: If a greater precision is required, then the distance variable should be declared as a floating point or double initially instead of an int. In order to convert the distance to a floating point, double, or int, the following code can be used:

```
double d_distance = (double) distance; //assuming distance was a float
float f_distance = (float) distance; //assuming distance was a double
int i_distance = (int) distance; //a loss of precision in this
conversion
```

15.4.3 Arithmetic Operations

Typecasting is done automatically for operations involving an integer and a floating point in addition, subtraction, multiplication, and division. The code in Fig. 15.13 shows several floating point and integer assignment operations.

The comments show the various values stored in the variables. Some interesting notes are on line 5 and line 7. On line 5, the compiler may complain of a loss in precision

```
Line 1.  float result = 9 + 9.1;   // = 18.1
Line 2.  float divided_result = 9.77/ 4;  // = 2.44
Line 3.  float equivalent_divided_result = 9.77 / ((float) 4);  // = 2.44
Line 4.  int int_result = 5 / 9;   // = 0
Line 5.  int still_int_result = 9.7 / 2.8;  // = 3
Line 6.  float float_result = 5 / ((float) 9);   // = 0.556
Line 7.  float ensure_float_result = 5 / 9;    // = 0.00
```

Fig. 15.13. Declaring and initializing several variables and a constant.

converting to an int without a cast. Line 7 returns a value of 0.00 because the 5 and 9 are both integer values and thus integer division is performed. The result of the integer division of 5 and 9 is 0 and then this value is cast as a floating point to 0.00. This can become a major issue of confusion for code operation, so care must be taken when performing calculations.

Example 15.4: Write a line of code that converts a mass (in grams) stored in a variable called `mass` to a weight (in newtons). Then store the result in double, floating point, and integer variables.

Solution:
First, we find the result in the greatest precision available. Then, we convert the result to the other data types using casting:

```
double f_result = mass / ((double) 1000) * 9.8;
float result = (float) f_result;
int i_result = (int) f_result; // there is a loss of precision
```

One important note about the naming of variables in the Arduino IDE is that the names cannot start with a number or a punctuation mark. Additionally, the variable names cannot contain punctuation marks besides underscores.

Operations with Characters and Relation to C Language

Going back to Fig. 15.12, however, line 8 declares a character. On the Arduino, a character variable is only stored using a single byte of data and can only hold a single character, number, or punctuation mark. These characters can be added together using arithmetic just like integers (e.g., to change a lowercase letter to an uppercase letter), but this makes no sense unless the ASCII character set is used. Strings which are a collection of characters will be discussed later.

As noted before, the Arduino runs a slightly modified version of the C language. As the C language was developed, it is not intended as a highly sophisticated object-oriented language like the other higher object-oriented languages Java and C++. This is not to say that C does not support any objects, but rather it is intended as a more low-level language as a step above the hardware and assembly languages.

15.4.4 Functions

Before moving on to objects, an important piece of the Arduino language is the creation of *functions*. Functions allow the user to specify a segment of code that can be used or *called* over and over again. This helps us to minimize writing the same segment of code over and over by having one simple function to call that does the action of the segment of code. The two basic functions that define an Arduino sketch have already been introduced with `setup` and `loop`. In Arduino, a function can be created by using the generic structure shown in Fig. 15.14.

A specific function that returns a floating point representing the number of pounds from an input of kilograms is shown in Fig. 15.15.

```
return_type functionName(parameter 1, parameter 2 … parameter n)
{
// Body of function
Return data type //if needed
}
```

Fig. 15.14. The generic structure of a function in Arduino.

The function defined in Fig. 15.15 has one float output parameter and functions in Arduino only may have 1 output parameter. The two lines before the definition of the

```
//float convertKgToLbs(float kgs)
//takes in a float that is the mass in kg and returns the value in lbs
float convertKgToLbs(float kgs)
{
    const float KG_TO_LBS = 2.2;    //conversion factor
    return (kgs * KG_TO_LBS);   //return the value after conversion
 }
```

Fig. 15.15. A function for converting kilograms to pounds.

function are called the *function header* and exist to document the functions' inputs and outputs and what the function does. Any parameters to the function must be separated by a comma. The return data type, specified as float here, could be any data type that has been defined previously. Only the data type identifier is included though. The actual *body of the function* in Fig. 15.15 only consists of the declaration of a constant integer called `KG_TO_LBS` which is the conversion factor in this case (2.2 lbs in 1 kg). The actual work of the function is done in the line `return (kgs * KG_TO_LBS);`. This line converts the parameter `kgs` supplied to the function to pounds by multiplying by `KG_TO_LBS` and then returns the value. The function can be either placed at the top of the file or in a separate file that is included with the Arduino sketch that uses the function. A function cannot be declared within another function. In order to call the function `convertKgToLbs` (in Arduino), the following line is used:

```
float totalPounds = convertKgToLbs(4.7); //convert 4.7 kg to lbs    (15.6)
```

The above line converts 4.7 kg to pounds and then stores the result in the float variable called `totalPounds`. The value 4.7 is the *argument* to the function. If the function did not have a return value, the function call would simply be: `convertKgToLbs(4.7);`. In this case, the function definition must use the reserved word "void" instead of any data type before the function name.

Example 15.5: Write a function that calculates the moment of inertia of a thin rod about a perpendicular axis passing through the rod's center. The function should return the result in the highest precision and take in the mass m and length L of the rod as the input arguments.

Solution: The moment of inertia about the rod's center is given by

$$I_{\text{rod}} = mL^2/12 \tag{15.7}$$

Therefore, the required function can be written in the form:

```
//double calculateRotInertiaOfRod(double mass, double length)
//takes in mass and length and returns the moment of inertia
double calculateRotInertiaOfRod(double mass, double length)
{
double rotational_inertia;
rotational_inertia = (1.0 / 12) * mass * length * length;
return rotational_inertia;
}
```

15.4.5 Libraries

Two of the standard objects C and Arduino both support are called *arrays* and *strings*. In Arduino, objects can also be created using a *library*. A library is a collection of code files that define variables and functions which create a new data type or add functionality to a preexisting data type. In the case of a new data type, the data type designed to be accessed only using the defined methods and is supposed to keep its data inside the data type without revealing the data. This is called *encapsulation* where the data is kept hidden from the user and is only accessed through specialized *access functions* when needed. A library in the Arduino environment consists of at minimum two code files. The first code file is a *.h or *header file* which defines the *function prototypes* (for the functions that the library allows the user to call). The actual definitions of the function will be implemented in the *.cpp *script file*. The script file and header file must have the same name to be understood as a library. One particularly important library is the *servo library* included in the Arduino installation. This servo library must be imported into the current sketch by either going to *Sketch* → *Import Library* → *Servo* from the IDE or by using writing the following line at the top of the sketch file:

`#include < Servo.h >` (15.8)

Notice that the above line does not terminate in a semicolon. This is because this type of line is processed by the compiler before processing of the code begins. Upon seeing this line of code, the compiler basically copies the content from the `Servo.h` header file over to the top of the sketch when the sketch is being compiled.

15.4.6 Objects. Application Example: A Servomotor

A *servomotor* is basically a motor with a potentiometer attached to the motor shaft. A control loop is implemented using specialized circuitry to actuate and finely control the position of the motor shaft. In Arduino, a *servo object* is created in order to better control a servomotor attached to the Arduino board. Figure 15.16 shows the basic setup for attaching a servomotor to the actual Arduino Uno board. In order for the servomotor to work properly, the motor must be attached to a pin that supports pulse-width modulation (PWM). These pins on the Arduino Uno are pins 3, 5, 6, 9, 10, and 11. They are distinguished by a tilde next to the pin on the board itself. Figure 15.16 shows the corresponding example.

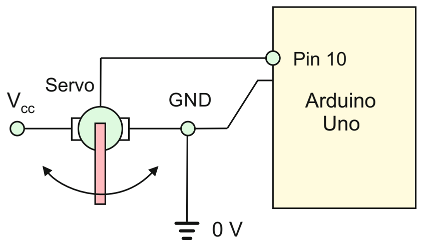


Fig. 15.16. Attaching a servomotor to the Arduino Uno.

Example 15.6: Write a program to drive a servomotor shown in Fig. 15.16 to one position.

Solution: The code is actually quite simple and is given by

```
#include <Servo.h> //import Servo library
Servo left_arm_servo; //declare Servo object
const int SERVO_PIN = 10; //pin Servo is on
void setup() {
pinMode(SERVO_PIN, OUTPUT); //set Servo pin to output mode
left_arm_servo.attach(SERVO_PIN); //attach servomotor
}
void loop() {
left_arm_servo.write(37); //write the position
delay(5); //wait 5 ms between each iteration of loop
}
```

Example 15.6 (cont.):
The second argument to the `pinMode` function is either `INPUT` or `OUTPUT` and specifies the type of the pin. The `attach` function can also take an optional second argument and third argument which specify the timing specifics of the PWM signal. For most small-power servomotors, the default settings of this example are sufficient. Line 9 is actually what causes the servomotor to move. This line generates a PWM signal on pin 9 that the servomotor interprets to drive to the set position of 37. This position is usually decently close to the degree rotation of the servomotor as most servomotors drive from 0° to about 180°. The value supplied to the write function on line 9 must be between 0 and 180 inclusive otherwise the behavior of the servomotor is undefined. Information on the servo library can be found in the Arduino tutorial page.

15.4.7 Interfacing with IO Pins

Here, we intend to show several techniques of interfacing with the IO pins on the Arduino board. Two of the IO pins on the Arduino board are digital and analog IO pins, respectively. Digital IO pins can be either read or written to by the Arduino. The same is true of the analog pins. Both digital pins and analog pins also support PWM. For any of the IO pins to work properly, they must be declared as either an `INPUT` or an `OUTPUT` pin using the `pinMode` function as follows:

$$\texttt{pinMode(pin\_number, TYPE); //TYPE is either INPUT or OUTPUT} \tag{15.9}$$

Once the pin is enabled, it can be written to or read from depending on which type it was declared as. If the pin is a digital pin, then it can be read using the following:

$$\texttt{int digital\_value = digitalRead(pin\_number); //returns a 1 or 0} \tag{15.10}$$

Since the pin is a digital pin, it can only return a value of 1 or 0. An analog pin is read using the analogRead function:

$$\texttt{int anal\_value = analogRead(pin\_number); //returns a value (0 - 1023)} \tag{15.11}$$

The analogRead function returns a value from 0 to 1023 (because of the 10 bits ADC). In order to write to a digital pin, the command is simply

$$\texttt{digitalWrite(pin\_number, STATE); //STATE is either HIGH or LOW} \tag{15.12}$$

The value of the pin then becomes either high (5 V) or low (0 V). On analog pins, the writing command creates a PWM signal with a controllable duty cycle from about 5 % to around 95 %. This is achieved by the command

```
analogWrite(value); //where value is from 0 to 255 (inclusive)   (15.13)
```

A value of 0 corresponds to the lowest duty cycle and a value of 255 corresponds to the highest duty cycle. With this information, more complex Arduino programs can be written that interface external circuitry, LEDs, and other sensors. There are numerous examples within the Arduino IDE for all types of analog writing to pins and reading of sensors, which can be found under the *File → Examples menu*. Reading through the code under this menu will help with understanding the Arduino syntax.

Section 15.5 More Advanced Arduino Programming

15.5.1 Conditional Statements

The previous section is enough to get simple programs working on the Arduino. However, these programs are limited in functionality and in complexity as they lack the real control statements and functional blocks that afford greater detail in programs. Some of these control statements are called *conditional statements*. These statements create some logical expressions with the output of true or false. The basic structure of conditional statements in Arduino is shown in Fig. 15.17.

```
Line 1. if(logical expression 1) {
Line 2.      //Some code to run if logical expression 1 is True
Line 3. }
Line 4. else if (logical expression 2)  {
Line 5.      //Different code to run if logical expression 2 is True
Line 6. }
Line 7. //more else if statements
Line 8. else if (logical expression n) {
Line 9.      //Even more code to run if logical expression n is True
Line 10.    }
Line 11.    else {
Line 12.        //Code to run if all above are false
Line 13.    }
```

Fig. 15.17. The basic structure of conditional statements.

In Fig. 15.17, there can be an arbitrary number of conditional statements. However, there must be at least one `if` statement before any `else if` or `else` statements. Also there can only be one `else` statement for any block of conditional statements. This makes intuitive sense after thinking that one does not say "or else" without first stating the "if" clause of a demand. The Arduino starts at line 1 and check logical expression 1 first. If logical expression 1 is true, then the Arduino runs the code between the curly braces (brackets) on lines 1 and 3. After executing that code, the Arduino skips execution of the code until after line 13. Using this technique, it is possible to explicitly lay out the control of the program through conditional statements. This can lead to *state programming* using variables as flags, but state machine programming is outside the scope of this text. If logical expression 1 is false, then the Arduino jumps to line 4 and checks logical expression 2 and processes the code in curly brackets on lines 4 and 6 if logical expression 2 were true, etc. If the Arduino processes all of the conditional statements in Fig. 15.17 and none of them are true, then control would jump to the curly brackets following the else statement on line 13. The curly brackets are only needed if the conditional code is more than a single line.

Example 15.7: Define a function which takes in a student GPA as a double and whose output is the grade range of a student: A (return 1) = 4.0–3.3; B (return 2) = 3.3–2.7; C (return 3) = 2.7–2.0; F (return 4) = 2.0–0.0.

Solution:
The function is rather simple to define and uses conditionals to check the various grade ranges of the student.

```
//int returnGradeRange (double GPA)
//takes in a double representing GPA of a student. Returns
//1, 2, 3, or 4 meaning the student is an A, B, C or F student
int returnGradeRange(double gpa)
{
if(gpa > 3.3 && gpa <= 4.0)
return 1; //An 'A' student
else if(gpa > 2.7 && gpa <= 3.3)
return 2; //A 'B' student
else if(gps > 2.0 && gpa <= 2.7)
return 3; //A 'C' student
else
return 4; //An 'F' student
}
```

The `if`, `else if`, and `else` structure is used here to check the conditions. The logical expression is a compounded expression joined together using two ampersands indicating logical AND: gpa > 3.3 && gpa <= 4.0. The other relational operators available for variables are less than (<), greater than or equal to (>=), and equal to (==). Additionally, logical OR (||) and logical NOT (!) are available to string together logical expressions in conditionals. Bitwise logical operators are also available for dealing with individual bits.

The function `returnGradeRange` could have been defined using only if statements that checked the GPA range. While this is syntactically valid, this forces the Arduino to check every conditional statement. Thus, the code above ensures the Arduino only checks as many conditions are necessary before exiting the function. This improves the execution speed of the code.

Example 15.8: Implement a logic circuit from Fig. 15.18 in Arduino code.

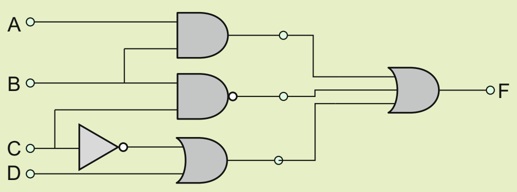


Fig. 15.18. A logic circuit.

Example 15.8 (cont.):

Solution: The shortest implementation may be cast in the form:

```
Line 1. if((A && B) || !(B && C) || (!C || D))
Line 2. digitalWrite(F_PIN, HIGH);
Line 3. else //the output of the digital circuit is False
Line 4. digitalWrite(F_PIN, LOW);
```

Note that the logical NOT operator does not work on integers but rather only Booleans. Thus A, B, C, and D would have to be declared as Booleans. Technically in Arduino however, true is defined as anything nonzero and false is defined only as 0. The code to implement the circuit in Fig. 15.18 could also be achieved using nested `if` statements and an `if`, `else if`, and else configuration.

15.5.2 Switch Statements

Another type of control statement is called the *switch statement*. In terms of control of the program, it is equivalent to conditional statements except that it follows a case-by-case setup. This arrangement allows the program to quickly jump to the correct case and execute the required code. Its generic layout is shown in Fig. 15.19.

```
Line 1. switch(variable)
Line 2. {
Line 3.      case (expression 1):
Line 4.           //Code to run if variable == expression 1
Line 5.      break;
Line 6.      case (expression 2):
Line 7.           //Code to run if variable == expression 2
Line 8.      break;
Line 9.      //More cases to check
Line 10.          default:
Line 11.          //Code to run if variable does not equal a defined case
Line 12.  }
```

Fig. 15.19. The basic setup of a switch statement with multiple cases.

The main elements of the switch statement are the beginning setup of the switch on line 1 of Fig. 15.19. This line lists the variable that will be checked against the various cases. Due to rounding errors, this variable should be an integer. After line 1 in Fig. 15.19 sets up the switch statement, different cases are listed. The order of the cases is not important. Each case must start with the word “case” and then be followed by some expression. If the variable and the expression are the same, the code between the case declaration and the break statement is executed. If the value stored in the variable does not equal any of the results of any of the particular expressions, then the code following the default case is executed. The break statement with semicolon is optional. The switch statements tend to execute slightly *faster* than conditional statements. This reduces the overall *latency* or delay of the code due to large numbers of instructions. Some instructions can be time intensive. For example, analogRead requires ~100 μs to complete!

Example 15.9: Write a function that returns an integer value indicating the following: 1 = drive forward, 2 = turn right, 3 = turn left, and 4 = stop, if a certain number of lines have been counted. The number of lines counted corresponds to: 1, 2, 3 = forward; 4 = turn right; 5 = forward; 6 = turn left; 7, 8 = forward; 9 = turn left; 10 = forward; and 11 = stop.

Solution: This problem is solved using either conditionals or a switch statement. The solution using conditionals is left as a homework problem. The solution using the switch statement is as follows:

```
Line 1. //int determineDriveStatus(int number_of_lines)
Line 2. //takes in the number of lines a robot has seen and returns
Line 3. //what the robot should do to continue driving around
Line 4. int determineDriveStatus(int number_of_lines)
Line 5. {
Line 6. switch(number_of_lines)
Line 7. {
Line 8. case 7: case 1: case 2: case 3: case 5: case 8: case 10:
Line 9. return 1; //drive forward
Line 10. break;
Line 11. case 4:
Line 12. return 2; //turn right
Line 13. break;
Line 14. case 6: case 9:
Line 15. return 3; //turn left
Line 16. break;
Line 17. default:
Line 18. return 4; //stop the robot
Line 19.}
Line 20.}
```

A switch statement can also be translated over to an `if`, `else if`, and `else` configuration. Line 8 shows how cases can be cascaded together to run the same piece of code. If `number_of_lines` equals 7, 1, 2, 3, 5, 8, or 10, the code execution is said to return 1. A break statement is only needed if there is more than 1 line of code to execute after a case or set of cases. Line 17, the default statement, is only executed if `number_of_lines` does is not in the range of 1–10. This minimizes the amount of debugging.

15.5.3 Loops

For Loop

Loops in Arduino are very similar to those in C although the Arduino has a few minor differences. A *for loop* is basically a segment of code that runs as many times as specified by the loop conditions. The basic setup of a for loop can be seen in Fig. 15.20.

```
Line 1. for(variable; logical expression; counting expression)  {
Line 2.      //Code to execute - the body of the loop
Line 3. }
```

Fig. 15.20. Basic construction of a for loop in Arduino.

The *pseudo-code* in Fig. 15.20 shows major items required in a for loop. Line 1 contains the important information about how the loop operates. Curly brackets are only needed if the code to execute in the loop is more than 1 line. The first part after the parenthesis is the variable being used to loop through the code. This variable can be defined outside of the loop or it can be defined in the loop itself. The next part is a logical expression. This logical expression includes $>$, $>=$, $==$, $<$, or $<=$ operators. The last part is a counting expression which describes how the loop changes the value of the variable. This must be an expression that alters the value of the variable; otherwise, the loop could be executed indefinitely. A typical mistake is a counting expression which never allows the logical expression of the loop to evaluate to true. An example is a loop that starts counting from 0, is counting by 2 each time and the terminating condition is when the counter equals 7. A similar problem can occur when using floating point numbers. Note that a for loop should never be followed by a semicolon. This could cause an *infinite loop*.

Example 15.10: Write a function that will print the message “Please press the reset button” 15 times on the serial monitor.

Solution: The required loop uses the integer variable `counter` to count from 0 to 14. The line `counter++` is equivalent to `counter = counter + 1`. The double plus is called the *increment operator*. Two other methods replace lines 6–7.

Method 1:

```
Line 1. //void printErrorMessage()
Line 2. //takes nothing into the function and returns nothing although
Line 3. //a message has been printed 15 times
Line 4. void printErrorMessage() {
Line 5.   int counter = 0;
Line 6. for(counter; counter < 15; counter++)
Line 7. Serial.println("Please press the reset button");
Line 8.}
```

Method 2:

```
Line 1. for(int i = 15; i > 0; i--)
Line 2. Serial.println("Please press the reset button");
```

The double minus in method 2 is called the *decrement operator*.

Example 15.10 (cont.):

Method 3:

```
int count;
for(count = 0; count != 15; count++)
Serial.println("Please press the reset button");
```

Method 3 uses a counter called "count" that starts at 0 and counts upward by 1 each time. The terminating condition is when value is equal to 15. This sort of terminating condition can be risky.

While Loop

Another type of loop similar to the for loop is called the *while loop*. The while loop acts exactly as the name implies: it executes the code in the body of the loop as long as a certain logical expression is true. The structure of a while loop can be seen in Fig. 15.21.

```
while(logical expression) {
//Code for the body of the loop
//increment the variable in the logical expression
}
```

Fig. 15.21. Basic structure of a while loop in Arduino.

The while loop requires the counter that is a part of the logical expression to be changed in the body of the loop. The while loop will not automatically update the variable each time through the loop unlike the for loop. The while loop can suffer from the infinite looping if the counter update is omitted or the logical expression is never reached. The while loop requires the curly brackets since it will have at least one line where the counter is updated. One note about the loops in general is that they can be terminated at any time by either a return, a return (value), or a break statement. The break statement will cause the program to jump to the end of the loop and begin code execution from there onward. The return or return (value) statement will cause execution of the function currently being called to cease and control will pass back to the calling function.

Example 15.11: Rewrite the function from the previous example using while loops.

Solution:
Method 1:

```
//void printErrorMessage()
//takes nothing into the function and returns nothing although
//a message has been printed 15 times
void printErrorMessage()
{
int counter = 0;
```

Example 15.11 (cont.):

```
Line 7. while(counter < 15)
Line 8. {
Line 9. Serial.println("Please press the reset button");
Line 10. counter++;
Line 11.}
Line 12.}
```

Method 2:

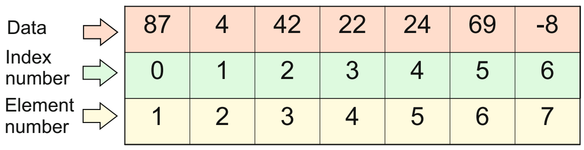
```
Line 1. int i = 15;
Line 2. while(i > 0)
Line 3. {
Line 4. Serial.println("Please press the reset button");
Line 5. i--;
Line 6.}
```

Method 3:

```
Line 1. int count = 0;
Line 2. while(count != 15)
Line 3. {
Line 4. Serial.println("Please press the reset button");
Line 5. count++;
Line 6.}
```

15.5.4 Arrays and Strings

One of the C objects the Arduino does support is a linear *array*. An array can be thought of simply as a long rectangular box that has been partitioned to hold various items. The particular storage location of an item is called the *element* and the number of the location is called the *index* of that location. The indexing of the array starts at 0 in C and Arduino and goes to the length of the array minus 1. A visual representation of an array of size 7 can be seen in Fig. 15.22. When an array is stored in memory, a block of memory is set aside and then the elements in the array are stored sequentially one after another in this block of memory.



Data	87	4	42	22	24	69	-8
Index number	0	1	2	3	4	5	6
Element number	1	2	3	4	5	6	7

Fig. 15.22. Values stored in the `usedPins` array after initialization.

In order to declare the array in the Arduino language, the type of data the array stores must be defined. Once created, the array should be initialized to some values before accessing the elements. Both tasks can be accomplished by adding the values in the corresponding assignment statement:

```
int usedPins[7] = {87, 4, 42, 22, 24, 69, -8};
```
(15.14)

This initializes the values of the array in Fig. 15.22. After the assignment, array elements can be individually accessed and individually changed:

```
int retrievedValue = usedPins[4]; //access array element
usedPins[4] = 7; //store 7 into 5th elem in usedPins
```
(15.15)

In the second expression (15.15), the programmer must ensure that the requested index value is always within the size of the array.

Exercise 15.6: What is the value of integer variable `retrievedValue` in expression (15.15)?

Answer: 24

Exercise 15.7: Write an Arduino function that adds 17 to the array in Fig. 15.22.

Answer:

```
int numbers_array = {6, 8, 7, 99, 100, 2, −3};
void add17ToArray(int size_of_array){
for(int index = 0; index < size_of_array; index++)
numbers_array[index] += 17; //access element and add 17
}
void setup() {
add17ToArray(7); //the array is size seven
}
void loop() {
//execute any code needed here
}
```

Another object that C and Arduino support is called a *string*. A string is simply a collection of characters that forms a longer word or sentence. Obviously, the size of a string is variable and can be modified by adding or removing characters. Strings in C and the Arduino are simply declared by using the same sort of notation as an array:

```
char name[6] = “Frank”;
```
(15.16)

Upon inspection of the declaration (15.16), an observant student would notice the assigned name is actually five letters long. The 6th character is actually the null character “\o” which is treated as a single character and is needed for displaying a string properly as the null character signals the end of the string. In order to extend the usefulness of the string objects, the *string library* defines a collection of variables, functions, and files that define how a string is processed and accessed. Strings are accessed in the same way arrays are accessed: by using the square brackets.

$$\begin{aligned}&\texttt{char letter} = \texttt{name}[2];\\&\texttt{char new\_line} = \texttt{name}[5];\end{aligned}\tag{15.17}$$

This would allow the new line character to be printed to see the exact value of the character (if cast as an integer) or would simply print a new line to the console.

Exercise 15.8: What is the value of variables `letter` and `new_line`, respectively, in expressions (15.17)?

Answer: ‘a’ and ‘\o’.

15.5.5 Serial Communication

The Arduino does not have a way to visually represent a string on the board itself. The way to get around this is to use the *serial communication* capabilities of the USB connection of the Uno to the computer. The Arduino IDE contains the *serial monitor* (console) button. If the Arduino is connected to the computer, then the serial monitor can be opened. This establishes a serial communications link with the Arduino at a set *baud rate*. The baud rate is a measure of how many bits per second are being transferred over the communications line. The default value is 9600 baud, which refers to 9600 bits/s of data transfer. The Arduino has the capability to communicate with the computer at baud rates from 300 all the way of to 115,200 bits/s. In order to have the Arduino communicate with the computer, both devices must be sending data at the same rate. In order to accomplish this, the serial communications must be started on the Arduino. The serial communication is illustrated in the following example.

Example 15.12: Write Arduino code that will read in a button sensor and an analog value and will print both values to the serial monitor.

Solution:

```
Line 1. const int BUTTON_PIN = 4;
Line 2. const int ANALOG_PIN = A0;
Line 3. int button_state, analog_value;
Line 4. void setup() {
Line 5. pinMode(BUTTON_PIN, INPUT);
Line 6. pinMode(ANALOG_PIN, INPUT);//both pins must be input pins
Line 7. digitalWrite(BUTTON_PIN, HIGH); //enable pull up resistor
Line 8. Serial.begin(9600); //begin the serial communication
```

Example 15.12 (cont.):

```
Line 9. button_state = 0;
Line 10. analog_value = 0;
Line 11.}
Line 12.void loop() {
Line 13. button_state=digitalRead(BUTTON_PIN);//read button state
Line 14. analog_value=analogRead(ANALOG_PIN);//read analog pin val.
Line 15. Serial.print("Button state: ");
Line 16. Serial.print(button_state);
Line 17. Serial.print(" Analog value: ");
Line 18. Serial.println(analog_value);
Line 19. delay(1); //wait 1 ms
Line 20.}
```

The two input pins have been declared and initialized in the setup function. Note line 7 where the code enables the internal pull-up resistor on the pin 4. This guarantees that the button is always in a known state. The pull-up resistors can be disabled by writing a LOW value instead of a HIGH one. The two input pins are simply read using the standard reading functions for analog and digital pins. The results are then printed in lines 15–18. Note that line 18 has a `println` function, which additionally skips a line to make the output more readable. Otherwise, using the print statements relies on the user to specify any needed spacing for printing as on lines 15–17.

15.5.6 Interrupts. Application Example: Emergency Motor Stop

Previously on the Arduino, buttons, switches, and other digital IO devices have been read using a method called *polling*. Polling is where each time through a loop the input pin is checked and read. This usually works well while the latency of the code is low and the loop executes quickly. Otherwise, the reading of the input pin gets delayed and information can be missed. This problem is illustrated in Fig. 15.23 where the input signal changes between sample 2 and sample 3, but the change is missed.

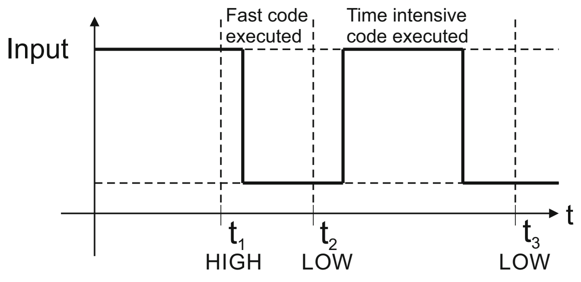


Fig. 15.23. The problem with polling an input pin.

In order to still handle looping and make sure the signal is processed in “almost” real time, an input pin is configured to handle *interrupts*. An interrupt allows the microprocessor to handle code execution while still checking a pin in almost real time. Once an interrupt is generated (a certain programmed state is achieved on the pin), the execution of the code finishes the last machine instruction and then jumps to an *interrupt service routine (ISR)*. An ISR is a special segment of code written to be called when an interrupt is generated. This code is executed and then the control of the program resumes from the last executed machine code instruction. On the Arduino Uno, there exist two pins capable of handling interrupts: pins 2 and 3. In order to configure a pin as an interrupt, the following line of code is needed in the setup function (the pin should already be declared as an input):

`attachInterrupt(interrupt number, ISR, trigger state);` (15.18)

Command (15.18) attaches the interrupt to the specific pin as denoted by the interrupt number. An interrupt number of 0 refers to pin 2 and an interrupt number of 1 refers to pin 3 being used. The ISR is the code to be called when the interrupt is triggered. The *trigger state* can be LOW (for triggering an interrupt when the pin is LOW), HIGH (for triggering when the pin is HIGH), RISING (for triggering an interrupt when the pin sees a rising edge of the signal), FALLING (when the pin transitions from HIGH to LOW), or CHANGE (for triggering an interrupt when the pin changes from HIGH to LOW or LOW to HIGH). The Arduino has only 1 priority level for interrupts and thus sees every interrupt as the same (despite how important the user may feel one interrupt to be). Thus in order to handle an interrupt coming in when already in an ISR, the Arduino has an *interrupt queue*. The interrupt queue is capable of storing 1 additional interrupt while processing another. Note that the use of switches with interrupts is tricky. When a switch closes, the mechanical contacts tend to quickly make and break contact several times before settling. This can result in many different rising and falling edges being generated. In order to avoid this problem, the switch must be *debounced*.

Example 15.13: Write an Arduino code that acts as an “emergency stop” and stops a motor attached to pin 10 using an interrupt when a button is pressed.

Solution:

```
Line 1. include <Servo.h>
Line 2. Servo motor;
Line 3. const int MOTOR_PIN = 10;
Line 4. const int BUTTON_PIN = 4;
Line 5. volatile boolean stop_motor;//variables modified in ISR are
volatile
Line 6. //void stopMotorISR()
Line 7. //Takes no parameters. Returns nothing.
Line 8. //Handles the interrupt code to stop the motor on the button
press
```

Example 15.13 (cont.):

```
Line 9. void stopMotorISR() {
Line 10. stop_motor = True;
Line 11.}
Line 12.void setup() {
Line 13. pinMode(BUTTON_PIN, INPUT);
Line 14. pinMode(MOTOR_PIN, OUTPUT);
Line 15. digitalWrite(BUTTON_PIN, HIGH); //pull up resistors on
Line 16. motor.attach(MOTOR_PIN);
Line 17. stop_motor = False;
Line 18. attachInterrupt(0,stopMotorISR,FALLING); //interrupt on
pin 2
Line 19.}
Line 20.void loop(){
Line 21. if(!stop_motor)
Line 22. motor.write(180); //full speed
Line 23. else
Line 24. motor.write(90); //stopped motor
Line 25.}
```

The first notable part of the code is line 5 where a Boolean data type is declared using the identifier of "volatile." This indicates that the value of `stop_motor` can be changed by code outside of where the variable is currently executing. The next important part is the ISR with the header. In the loop on lines 21–25, the motor will stop being driven when the logical expression on line 21 evaluates to false. Line 18 attaches the ISR function `stopMotorISR` to pin 2 and sets the pin to wait for a falling edge to hit the pin. However, since pull-up resistors are enabled, the button must be pulling the pin LOW to ground. The loop drives the motor at full speed in one direction using the servo object as the servo object drives the pin with a PWM pulse. The code will only drive the motor until the switch is pressed and then the motor will not move afterward no matter how many times the switch is pressed. In order to start the motor again, the reset button must be pressed on the Arduino.

15.5.7 Square Wave and PWM Generation with Arduino

The classic digital-to-analog conversion is not available on the Arduino. Therefore, a sinusoidal signal cannot be produced directly from the Arduino without using external circuitry. But what about creating a square wave with a certain period or frequency? The answer is yes, but the correct timing requires work. Let us consider a 50-kHz square wave with the period of 20 μs. This means that the signal is high for 0.01 ms and low for 0.01 ms. The standard `delay` function can only take in whole numbers of milliseconds; thus a function that deals with smaller portions of seconds is needed. In the Arduino language, the required function is `delayMicroseconds`, which takes in an argument that is an integer number of microseconds. In the present case, `delayMicroseconds` would be called with an argument of 10 after the pin was switched HIGH and then after the pin was switched LOW again. The code to create the 50-kHz square wave is given in Fig. 15.24.

```
const int SQUARE_WAVE_PIN = 4;
void setup() {
     pinMode(SQUARE_WAVE_PIN, OUTPUT);
}
void loop() {
     digitalWrite(SQUARE_WAVE_PIN, HIGH);
     delayMicroseconds(10); //wait 10 microseconds
     digitalWrite(SQUARE_WAVE_PIN, LOW);
     delayMicroseconds(10);
}
```

Fig. 15.24. Program that generates a 50-kHz square wave on pin 4.

The program in Fig. 15.24 produces a square wave with a frequency of around 33 kHz. Why is there a disparity between theory and application? The reason is in hardware switching times. The transistors that regulate on and off of digital pins require some finite time to change state and process the instructions from the code. What happens when there is not a delay? In theory, the pin should be able to change state infinitely fast. This obviously cannot happen as things cannot change instantaneously due to the hardware constraints and software latency. In the Arduino, the *fastest frequency* achievable is around 100 kHz. This shows that the hardware switching time is about 5 μs as the 100-kHz square wave has a period of around 10 μs. An astute observation would be that this switching time is on the same order of magnitude as the delay times of the 50-kHz square wave. Thus, in order to compensate for hardware switching times and produce the 50-kHz signal, the delays could be modified as shown in Fig. 15.25. The numbers there are the closest integers since `delayMicroseconds` does not accept floating point values.

```
const int SQUARE_WAVE_PIN = 4;
void setup() {
     pinMode(SQUARE_WAVE_PIN, OUTPUT);
}
void loop() {
     digitalWrite(SQUARE_WAVE_PIN, HIGH);
     delayMicroseconds(4); //wait 4 microseconds
     digitalWrite(SQUARE_WAVE_PIN, LOW);
     delayMicroseconds(6);
}
```

Fig. 15.25. Program that generates ~50-kHz square wave.

In order to change the duty cycle of a square signal in the servo library, the `analogWrite` function can be employed. The `analogWrite` function can be called by supplying the pin number and the value to write. This value is in the range of 0–255 which represent duty cycles from about 5 % to about 100 %. If the `analogWrite` function is used, the servomotor must be connected to an analog pin.

Summary

Topic	Arduino sample program

Data types and assignments	Define several constants and assigns conversions to variables after using type casting
	`#define CONVERSION 2.54`
	`#define NUM_INCHES 10`
	`int intResult = 0;`
	`float floatResult = 0.0;`
	`double doubleResult = 0.0;`
	`void setup(){`
	`intResult = (int)(NUM_INCHES*CONVERSION);`
	`floatResult = NUM_INCHES * CONVERSION;`
	`doubleResult = NUM_INCHES * CONVERSION;`
	`}`
	`void loop(){ }`

Arithmetic operations and functions	Read analog voltage on pin A1 and output a square wave on pin 3. The duty cycle is proportional to the analog voltage
	`#define ANALOG_PIN A1`
	`#define SQR_W_PIN 3`
	`void setup() {`
	`pinMode(ANALOG_PIN, INPUT);`
	`pinMode(SQR_W_PIN,OUTPUT);`
	`}`
	`void loop() {`
	`int an_read = analogRead(ANALOG_PIN);`
	`int dutycycle = map(an_read,0,1023,0,255);`
	`analogWrite(SQR_W_PIN, dutycycle);`
	`delay(1);`
	`}`

Conditional statements	Function that uses conditionals to determine the return value
	`char returnMovieRating(int rating){`
	`if(rating <= 50)`
	`return ‘G’;`
	`else if(rating > 50 && rating <60)`
	`return ‘F’;`
	`else if(rating >= 60 && rating <70)`
	`return ‘D’;`
	`else if(rating >= 70 && rating <80)`
	`return ‘C’;`
	`else if(rating>=80&&rating<90)`
	`return ‘B’;`
	`else if(rating>=90&&rating<=100)`
	`return ‘A’;`
	`else`
	`return ‘U’;`
	`}`

Switch statements	Function that uses a switch statement to return a drive value. When called from the loop, it returns the robot state
	`int returnMotorDriveVal(int state){`
	`switch(state){`
	`case 1: case 4: case 5: case 7:`
	`return 180; //Full power forward`
	`break;`
	`case 2: case 6:`
	`return 90; //Stop motor`
	`break;`
	`case 3:`
	`return 0; //Full power backward`
	`break;`
	`case 8: case 9: case 10:`
	`return 45; //Half power backward`
	`break;`
	`} }`

For loop	Program that will retrieve the drive values using the function from above and store them in a new array to be used
	`#define SIZE 8`
	`int state;`
	`int robotMot[SIZE]={8,1,5,2,7,4,6,1};`
	`int driveValues[SIZE];`
	`void setup(){`
	`state = 0;`
	`}`
	`void loop(){`
	`for(int i = 0; i < SIZE; i++)`
	`driveValues[i] = returnMotorDriveVal(robotMot[i]);`
	`}`

While loop	Program that will blink an LED a set number of times when a button is pressed
	`#define NUM_BLINKS 10`
	`#define WAIT_TIME 250`
	`#define BUTTON_PIN 5`
	`#define LED_PIN 7`
	`void setup(){`
	`pinMode(LED_PIN, OUTPUT);`
	`pinMode(BTN_PIN, INPUT);`
	`digitalWrite(BTN_PIN,HIGH);`
	`}`
	`void loop(){`
	`int buttonInput=digitalRead(BTN_PIN);`
	`if(buttonInput == 0){`
	`int num = 1;`
	`while(num <= NUM_BLINKS){`
	`digitalWrite(LED_PIN, HIGH);`
	`delay(WAIT_TIME);`
	`digitalWrite(LED_PIN, LOW);`
	`delay(WAIT_TIME);`
	`num++;`
	`}  }  }`

Numerical array	Program that sorts an array into an even and odd arrays in the setup function
	`#define SIZE 5`
	`int dataArray[SIZE]={74,-1,12,68,47};`
	`int evensArray[SIZE];`
	`int oddsArray[SIZE];`
	`int evenIndex;`
	`int oddIndex;`
	`void setup(){`
	`evenIndex = 0;`
	`oddIndex = 0;`
	`for(int i=0;i<10;i++){`
	`if(dataArray[i]%2==0){//number is even`
	`evensArray[evenIndex] = dataArray[i];`
	`evenIndex++;`
	`}`
	`else {`
	`oddsArray[oddIndex] = dataArray[i];`
	`oddIndex++;`
	`} } }`

(continued)

String array	Program that continually swaps the letters in a string array
	`#define SIZE 8`
	`char name[SIZE] = "Charles";`
	`void setup(){ }`
	`void loop() {`
	`int j = SIZE - 2;`
	`for(int i = 0; i < j; i++) {`
	`char temp = name[i];`
	`name[i] = name[j];`
	`name[j] = temp;`
	`j--;`
	`}`
	`delay(1);`
	`}`

Serial monitor	Program that reads analog/digital inputs and prints values to the Serial Monitor
	`#define BTN_PIN 7`
	`void setup() {`
	`pinMode(BTN_PIN, INPUT);`
	`digitalWrite(BTN_PIN,HIGH);`
	`pinMode(A0, INPUT);`
	`Serial.begin(115200);`
	`}`
	`void loop() {`
	`int analogVal = analogRead(A0);`
	`int digitalVal=digitalRead(BTN_PIN);`
	`Serial.print("Analog value: ");`
	`Serial.print(analogVal);`
	`Serial.print(" Button state: ");`
	`Serial.println(digitalVal);`
	`delay(1);`
	`}`

Interrupt	Pushbutton on pin 4 that will increment the state of the robot depending on the number of times the button has been pressed. The code uses an ISR on pin 2
	`#include <Servo.h>`
	`#define MAX_STATE 5`
	`Servo motor;`
	`const int MOT_PIN=10;`
	`const int BTN_PIN = 4;`
	`volatile int BTNPress;`
	`void countBTNISR() {`
	`buttonPresses++;`
	`if(BTNPress>MAX_STATE)`
	`buttonPresses = 0;`
	`}`
	`void setup() {`
	`pinMode(BTN_PIN,INPUT);`
	`pinMode(MOT_PIN, OUTPUT);`
	`digitalWrite(BTN_PIN, HIGH);`
	`motor.attach(MOT_PIN);`
	`BTNPress = 0;`
	`attachInterrupt(0,countBTNISR, RISING);`
	`}`
	`void loop(){`
	`switch(BTNPress){`
	`case 0:`
	`motor.write(0);`
	`break;`
	`case 2: case 5:`
	`motor.write(90);`
	`break;`
	`case 3:`
	`motor.write(135);`
	`break;`
	`case 4:`
	`motor.write(180);`
	`break;}  }`

Square wave/PWM	Program that generates robust PWM signal on pin 7. This code should only be used to generate frequencies between 61Hz and 50KHz. For more accurate timing use the Servo Library
	`#define PWM_PIN 7`
	`#define TIME_HL 4//constant time 4 musec`
	`#define TIME_LH 6//constant time 6 musec`
	`#define DUTY_CYCLE 0.5//between 0 and 1`
	`#define PERIOD 20 //in microseconds`
	`int timeHigh;`
	`int timeLow;`
	`void setup() {`
	`pinMode(PWM_PIN, OUTPUT);`
	`timeHigh=(int)(((float)PERIOD)* DUTY_CYCLE-TIME_LH);`
	`timeLow=(int)(((float)PERIOD)* DUTY_CYCLE-TIME_HL);`
	`}`
	`void loop() {`
	`digitalWrite(PWM_PIN, HIGH);`
	`delayMicroseconds(timeHigh);`
	`digitalWrite(PWM_PIN, LOW);`
	`delayMicroseconds(timeLow);`
	`}`

Problems

15.1 Architecture of Microcontrollers

Problem 15.1.

A. List some common IO devices.
B. Draw the schematic of a CPU.
C. Draw the schematic of a basic computer.

Problem 15.2. Convert the following numbers to bits:
A. 14 GB
B. 27 MB
C. 42 KB
D. 0.97 TB

Problem 15.3.

A. Draw a diagram showing a CPU communicating with memory over a 16-bit bus and label each of the data connections.
B. Repeat the previous task but use the abbreviated bus notation.

15.2 Memory

Problem 15.4. Store the value 57984 (decimal) in memory starting at address 0100h using little-endian notation. Repeat with big-endian notation. Present the corresponding tables.

Problem 15.5. Store the value 372110 in memory starting at address 0400h using little-endian notation. Repeat with big-endian notation. Present the corresponding tables.

Problem 15.6. What is the decimal value stored in memory in the following table starting from address 0200h using little-endian notation? Repeat with big-endian notation.

Address	Byte value
0204h	...
0203h	01h
0202h	22h
0201h	FFh
0200h	A0h
01FFh	....

(continued)

Problem 15.7. What is the decimal value stored in memory in the following table starting from address 0200h using little-endian notation? Repeat with big-endian notation.

Address	Byte value
0204h	...
0203h	77h
0202h	04h
0201h	BBh
0200h	08h
01FFh	....

Problem 15.8. Describe each of the following types of memory and their application:
A. RAM
B. ROM
C. EEPROM
D. PROM
E. EPROM
F. Flash

Problem 15.9.

A. Describe the difference between volatile and nonvolatile memory.
B. Describe the advantages and disadvantages of flash memory.

Problem 15.10. How wide must a data bus be to access:
A. 1 GB of memory?
B. 1.44 MB of memory?
C. 1 TB of memory?
D. 22 KB of memory?

15.3 Arduino Uno: An Embedded Microcontroller

Problem 15.11.

A. Write the bare-bones minimum that is required to create an Arduino sketch.
B. Write the corresponding C code.

Problem 15.12. What type of memory and how much of it is available for programs on the Arduino Uno?

Problem 15.13. Describe the process of writing and uploading a program for the Arduino.

15.4 Basic Arduino Syntax

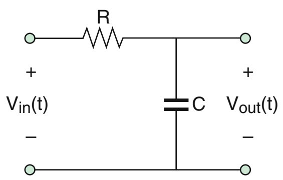
Problem 15.14. Describe the rationale behind a function header and write an example for a function that accepts several integers and returns the maximum. Do not worry about implementation of the function.

Problem 15.15. Design a function that will return the integer value of a character that is passed to the function (*Hint:* use typecasting).

Problem 15.16. Write a function for the Arduino that takes in the resistance and current in a circuit branch and finds and returns the voltage drop across the resistance.

Problem 15.17. Write a function for the Arduino that takes in the value of two resistors and finds the equivalent parallel combination of the two (returned as a double).

Problem 15.18. Write a function for the Arduino that returns the value of the transfer function for a given frequency for the following low-pass filter with $R = 100$ kΩ and $C = 1.59$ nF. Present the corresponding code.



Problem 15.19. Write a function that will configure the pins in the following configuration. Also if needed enable the pull up resistors on any input pins:
Pin 2 = INPUT
Pin 3 = OUTPUT
Pin 4 = OUTPUT
Pin 5 = INPUT
Pin 6 = INPUT
Pin 7 = OUTPUT
Pin A0 = INPUT
Pin A2 = OUTPUT

Problem 15.20. Design and implement a code that reads in an analog voltage value across a variable resistor (potentiometer) and converts the resulting value into the duty cycle value of a square wave on pin 3. The potentiometer should be on pin A4. Discuss the corresponding normalization procedure.

Problem 15.21. Write a program that reads in an analog voltage value from a light sensor (phototransistor) on analog pin A0 and converts this value to the corresponding binary number to be displayed on 8 LEDs. Discuss the corresponding normalization procedure.

Problem 15.22. If the output of `analogWrite` is a duty cycle ranging from 5 to 95 % corresponding to 0–255, convert the following values to their complement:

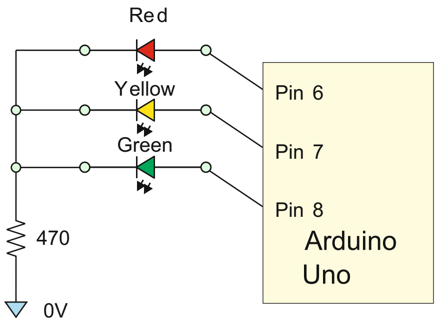
A. 67 %
B. 23
C. 44 %
D. 233

Problem 15.23. The ADC on the Arduino takes in an input signal in the range of 0–5 V and converts it to a decimal value in the range of 0–1023. Convert the following values to their complements:

A. 3.7 V
B. 898
C. 1.2 V
D. 469

Problem 15.24. Write a program that counts in binary from 0 to 255 on eight external LEDs on pins 4–11 (the binary counter). Use appropriate resistors to limit the current.

Problem 15.25. Create a program and external circuitry that mimics a traffic light. The "green" stays active for 8 s. The "yellow" is active for 3 s and then "red" is active for 10 s. The pattern repeats indefinitely.



Problem 15.26. Create a program that turns the Arduino into a handheld flashlight using a push button to turn an LED on when the button is held down.

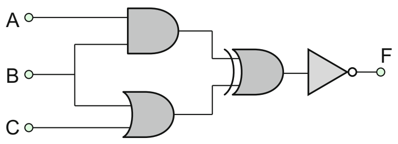
Problem 15.27.

A. Write a program to control the speed of a motor using external circuitry (a single switching transistor or an H-bridge), a potentiometer, and the servo library.
B. Realize the corresponding project in hardware.

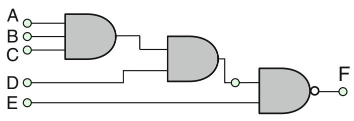
Problem 15.28. Write a program to control the position of a servo using ten switches attached to digital pins 2 and 4–12. The servo should be attached to pin 4. Each switch is acting as one bit of the 10-bit ADC of the Arduino.

15.5 More Advanced Arduino Programming

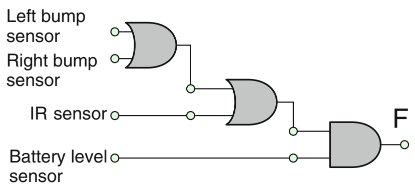
Problem 15.29. Implement the following digital logic circuit in a function that sets the output pin (pin 10 of the Arduino) HIGH or LOW.



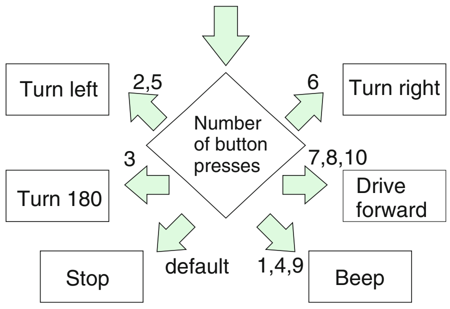
Problem 15.30. Repeat the previous problem for the circuit shown in the following figure.



Problem 15.31. Repeat problem 15.29 for the circuit shown in the following figure.



Problem 15.32. Implement a function that sets an output pin (pin 10) HIGH or LOW as shown in the following figure (use a switch statement).



Problem 15.33. Translate the code from Example 15.9 in Section 15.5.2 to use conditional statements instead of the switch statement.

Problem 15.34. What is an infinite loop? Discuss and document several ways that an infinite loop can occur and how to ensure this does not happen.

Problem 15.35. Translate the following while loops into for loops:

a)
```
int counter = 7;
while(counter <= 20)
{
 counter++;
}
```

b)
```
int index = 99;
while(index > -7)
{
 index -= 9;
}
```

Problem 15.36. Translate the following for loops into while loops:

a)
```
for(int row = 0; row < 6; row++)
{
     delay(9);
}
```

b)
```
for(int count=278;count!=0;count-=2)
{
     delay(17);
}
```

Problem 15.37. Find and fix the errors in the following loops:

a)
```
for(int count=278;count!=0;count-=3)
{
     delay(2);
}
```

b)
```
int number_of_times = 11;
while(number_of_times > 0)
{
     delay(11);
}
```

Problem 15.38. Write a function that will cycle through the string "Elizabeth" and will copy the string into a blank string that has ten entries (remember the terminating null character at the end of the string).

Problem 15.39. Write a sketch that will print "An Arduino says Hello world!" to the serial monitor the following number of times:

A. 7
B. 12
C. 190

Problem 15.40. Write a sketch that can print out the binary representation of the numbers 0–64 to the serial monitor. *Hint:* Use a function to convert the number to binary then print it out.

Problem 15.41. Write a function that will cycle through an array and print the values with the corresponding indices to the serial monitor. The array should be {7, 29, 444, 42, 69, 8, −10020}.

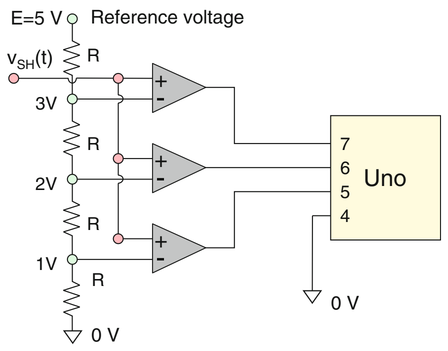
Problem 15.42. Design a program that uses a potentiometer to cycle through letters in the following character array (string) and print the result to the serial monitor.
char name[8] = "Stephan";

Problem 15.43. Compare and contrast the use of interrupts versus polling of inputs.

Problem 15.44. Discuss the issue of latency in code: what it is, what causes latency, and how to reduce it?

Problem 15.45. Create a program that turns the Arduino into a voltmeter (0–5 V only!) using two analog pins A0 and A1. The Arduino should print out the voltage at the two pins and the difference between the two pins in both polarities. The voltage should be expressed in volts, not in the decimal numbers from the ADC. *Hint*: Write a function that performs the function of a DAC in software to print out values.

Problem 15.46. Implement a code that uses an external 2-bit flash ADC to read values on four digital pins and convert the result to a 2-bit binary.



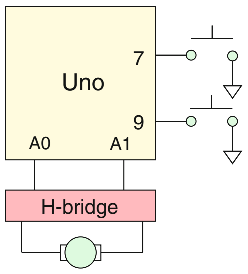
Problem 15.47. Create a program that flashes an LED every second without using the built-in delay function.

Problem 15.48.

A. Create and present a program that will increment a counter on a button push. The counter is then used to blink an LED that many times per minute.
B. Design the corresponding circuitry and present the circuit outline.

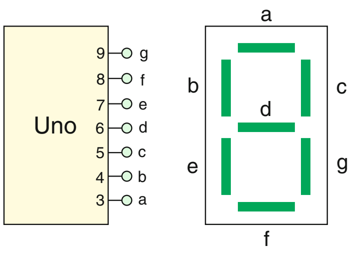
Problem 15.49. Update the code from the previous problem and the corresponding circuitry to add a decrementing counter. Make sure the result of the LED blinks per minute does not go negative or cause a runtime error.

Problem 15.50. Design and implement the code and the corresponding circuitry that will actuate a motor in two directions without using the servo library.

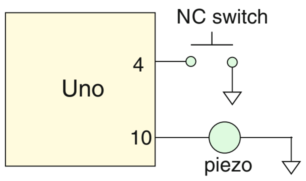


The circuitry should use two push button switches to choose the direction.

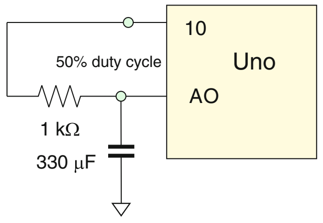
Problem 15.51. Write a program that counts hexadecimal numbers from 0 to F every second and displays the count on a seven segment display as shown in the following figure. The program that drives the 7-segment display must manually set the digital pins high corresponding to the value to be shown.



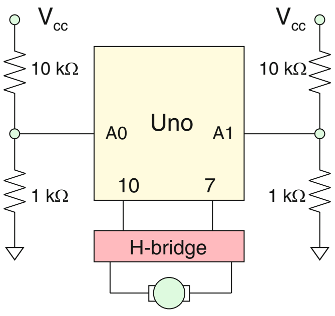
Problem 15.52. Write a program that acts as a burglar alarm. The corresponding schematic follows. The alarm waits for the switch to be released and then turns on a piezo-buzzer as the alarm. The alarm should not reset if the switch is reset (only after resetting the Arduino). The piezo can be sounded by simply writing a PWM signal to it.



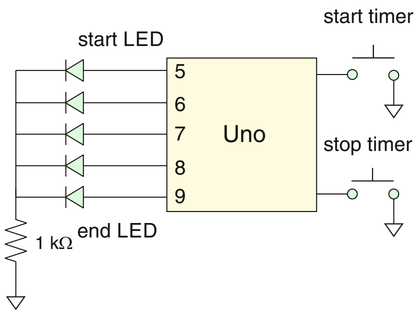
Problem 15.53. Create a program that will read the voltage across the capacitor and print the value to the screen (one voltage reading per line) for the circuit in the following figure. Also print out the time measurement along with the voltage. *Hint:* In order to observe voltage variation across the capacitor, a square wave must be applied to pin 10.



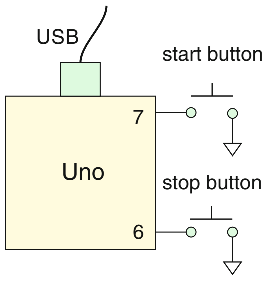
Problem 15.54. Create a program that uses two light sensors (LDRs) to actuate a motor in two directions. The sensors are modeled by voltage dividers. In order to run the motor in two directions, two digital pins are used on the Uno.



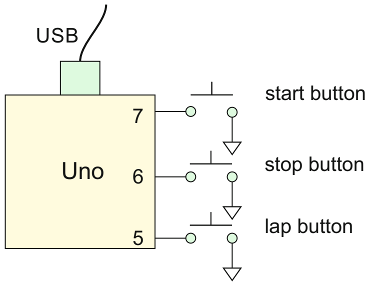
Problem 15.55. Create a program that turns the Arduino into a reaction time calculator. A sketch is shown in the following figure. The program starts a countdown of LEDs when the start button is pressed. The program then counts the time it takes the user to press the end button once the final button has been pressed.



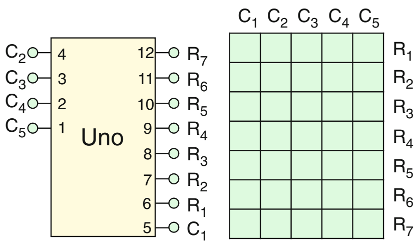
Problem 15.56. Write a program that turns the Arduino into a rudimentary stopwatch. The Arduino should begin "timing" after the push of a push button and then "stop" timing after a push of a different button. A sketch is shown in the following figure. *Hint:* Use some of the built-in timing features.



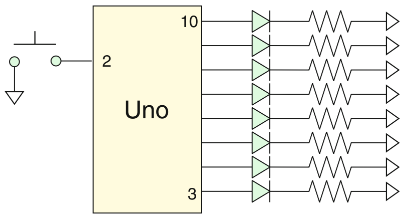
Problem 15.57. Implement a "lap" feature which spits out the time elapsed instead of stopping the count. A sketch is shown in the following figure.



Problem 15.58. Write a program that will control a 7 × 5 LED matrix as shown in the following figure. In order to make an LED the cell {R1, C1} on C1 must be LOW and R1 must be HIGH. *Hint:* This problem involves the use of arrays and several functions.



Problem 15.59. Write a program and create external circuitry that mimics the ball drop at New Years. The "drop" should start when a button is pressed and counts down 10 LEDs. The final LED stays on. *Hint:* Use a function to create the ball drop.



Problem 15.60. Create a program that will generate a square wave on pin 10 of the Arduino with the following frequencies:

A. 25 kHz
B. 7 kHz
C. 300 Hz
D. 2 Hz

Problem 15.61. Write Arduino code that generates a 20-Hz square wave on pin 7 without using `delay` or `delayMicroseconds` function.