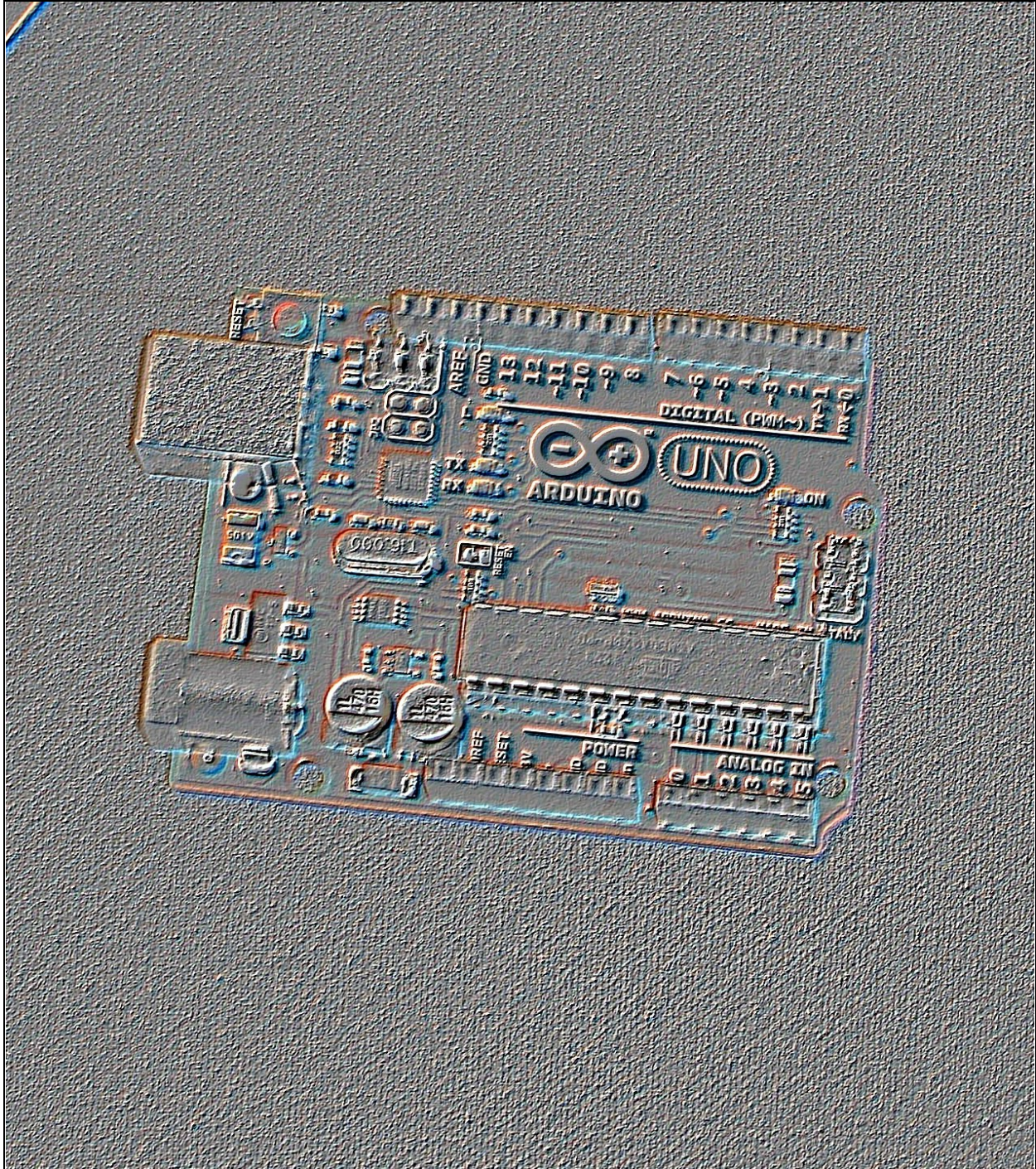# Embedded Controllers
Using C and Arduino / 2E

James M. Fiore

Embedded Controllers

Using C and Arduino

by

James M. Fiore

Version 2.1.10, 07 May 2021

For more information or feedback, contact:

James Fiore, Professor
Electrical Engineering Technology
Mohawk Valley Community College
1101 Sherman Drive
Utica, NY 13501
jfiore@mvcc.edu
oer@jimfiore.org

For the latest revisions, related titles, and links to low cost print versions, go to:

www.mvcc.edu/jfiore or my mirror sites www.dissidents.com and www.jimfiore.org

YouTube Channel: Electronics with Professor Fiore

Cover art by the author

Introduction

This text is designed to introduce and expand upon material related to the C programming language and embedded controllers, and specifically, the Arduino development system and associated Atmel ATmega microcontrollers. It is intended to fit the time constraints of a typical 3 to 4 credit hour course for electrical engineering technology and computer engineering technology programs, although it could also fit the needs of a hardware-oriented course in computer science. As such, the text does not attempt to cover *every* aspect of the C language, the Arduino system or Atmel AVR microcontrollers. The first section deals with the C language itself. It is assumed that the student is a relative newcomer to the C language but has some experience with another high level language, for example, Python. This means concepts such as conditionals and iteration are already familiar and the student can get up and running fairly quickly. From there, the Arduino development environment is examined.

Unlike the myriad Arduino books now available, this text does not simply rely on the Arduino libraries. As convenient as the libraries may be, there are other, sometimes far more efficient, ways of programming the boards. Many of the chapters examine library source code to see "what's under the hood". This more generic approach means it will be easier for the student to use other processors and development systems instead of being tightly tied to one platform.

All Atmel schematics and data tables are derived from the published version of the Atmel 328P documentation which may be found at https://www.microchip.com/wwwproducts/en/ATMEGA328P This serves as the final word on the operation and performance of the 328P and all interested parties should become familiar with it.

There is a companion laboratory manual to accompany this text. Other OER (Open Educational Resource) laboratory manuals in this series include Computer Programming with Python, and Science of Sound. OER texts and laboratory manuals are available for Operational Amplifiers & Linear Integrated Circuits, Semiconductor Devices, DC Electrical Circuit Analysis and AC Electrical Circuit Analysis. Please check my web sites for the latest versions.

A Note from the Author

This text is used at Mohawk Valley Community College in Utica, NY, for our ABET accredited AAS program in Electrical Engineering Technology. Specifically, it is used in our second year embedded controllers course. I am indebted to my students, co-workers and the MVCC family for their support and encouragement of this project. While it would have been possible to seek a traditional publisher for this work, as a long-time supporter and contributor to freeware and shareware computer software, I have decided instead to release this using a Creative Commons non-commercial, share-alike license. I encourage others to make use of this manual for their own work and to build upon it. If you do add to this effort, I would appreciate a notification.

> *"When things get so big, I don't trust them at all*
> *You want some control-you gotta keep it small"*
>
> *- Peter Gabriel*

Table of Contents

* Included for language coverage but seldom used for small to medium scale embedded work.

** Including modest comic relief for *film noir* buffs.

1. Course Introduction

1.1 Overview

This course introduces the C programming language and specifically addresses the issue of embedded programming. It is assumed that you have worked with some other high level language before, such as Python, BASIC, FORTRAN or Pascal. Due to the complexities of embedded systems, we begin with a typical desktop system and examine the structure of the language along with basic examples. Once we have a decent grounding in syntax, structure, and the development cycle, we switch over to an embedded system, namely an [Arduino](#) based development system.

This course is designed so that you can do considerable work at home with minimal cost if you choose to (this is entirely optional, but programming these little beasties can be addicting so be forewarned). Along with this course text and the associated lab manual, you will need an Arduino Uno board (about $25) and a USB host cable. A small "wall wart" power adapter for it may also be useful. There's a lot of free C programming info on the Internet but if you prefer print books and want more detail, you may also wish to purchase one of the many C programming texts available. Two good titles are Kochan's book *Programming in C* and the one by Deitel & Deitel, *C-How to Program*. Whichever book you choose, make sure that its focus is C and not C++. You will also need a desktop C compiler. Just about any will do, including Visual C/C++, Borland, CodeWarrior, or even GCC. A couple of decent freeware compilers available on the Internet include [Pelles C](#) and [Miracle C](#).

1.2 Frequently Asked Questions

Why learn C language programming?

> C is perhaps the most widely used development language today. That alone is a good reason to consider it but there's more:

- It is a modern structured language that has been standardized (ANSI).
- It is modular, allowing reuse of code.
- It is widely supported, allowing source code to be used for several different platforms by just recompiling for the new target.
- Its popularity means that several third-party add-ons (libraries and modules) are available to "stretch" the language.
- It has type checking which helps catch errors.
- It is very powerful, allowing you to get "close to the metal".
- Generally, it creates very efficient code (small space and fast execution).

What's the difference between C and C++?

> C++ is a superset of C. First came C, then came C++. In fact, the name C++ is a programmer's joke because ++ is the increment operator in C. Thus, C++ literally means "increment C", or perhaps "give me the next C". C++ does everything C does plus a whole lot more. These extra features don't come free and embedded applications usually cannot afford the overhead. Consequently, although much

desktop work is done in C++ as well as C, most embedded work is done in C. Desktop development systems are usually referred to as C/C++ systems meaning that they'll do both. Embedded development systems may be strictly C or even a variant of it (as is ours).

Where can I buy an Arduino development board?

The [Arduino Uno](#) board is available from a variety of sources including Digi-Key, Mouser, Parts Express and others. Shop around!

What's the difference between desktop PC development and embedded programming?

Desktop development focuses on applications for desktop computers. These include things like word processors, graphing utilities, games, CAD programs, etc. These are the things most people think of when they hear the word "computer". Embedded programming focuses on the myriad nearly invisible applications that surround us every day. Examples include the code that runs your microwave oven, automobile engine management system, cell phone, and many others. In terms of total units, embedded applications far outnumber desktop applications. You may have one or even a few PCs in your house but you probably use dozens of embedded applications every day. Embedded microcontrollers tend to be much less powerful but also much less expensive than their PC counterparts. The differing programming techniques are an integral part of this course and we shall spend considerable time examining them.

How does C compare with Python?

If, like many students taking this course, your background is with the [Python](#) language, you may find certain aspects of C a little odd at first. Some of it may seem overly complicated. Do not be alarmed though. The core of the language is actually simple. Python tends to hide things from the programmer while C doesn't. Initially, this seems to make things more complicated, and it does for the most simple of programs, but for more complicated tasks C tends to cut to the heart of the matter. Many kinds of data and hardware manipulation are much more direct and efficient in C than in other languages. One practical consideration is that C is a compiled language while most versions of Python are essentially interpreted. This means that there is an extra step in the development cycle, but the resulting compiled program is much more efficient. We will examine why this is so a little later.

How does C compare with assembly language?

Assembly has traditionally been used when code space and speed are of utmost importance. Years ago, virtually all embedded work was done in assembly. As microcontrollers have increased in power and the C compilers have improved, the tables have turned. The downside of assembly now weighs against it. Assembly is processor-specific, unstructured, not standardized, nor particularly easy to read or write. C now offers similar performance characteristics to assembly but with all the advantages of a modern structured language.

2. C Memory Organization

2.1 Introduction

When programming in C, it helps if you know at least a little about the internal workings of simple computer systems. As C tends to be "close to the metal", the way in which certain things are performed as well preferred coding techniques will be more apparent.

First off, let's narrow the field a bit by declaring that we will only investigate a fairly simple system, the sort of thing one might see in an embedded application. That means a basic processor and solid state memory. We won't worry about disk drives, monitors, and so forth. Specific details concerning controller architecture, memory hardware and internal IO circuitry are covered in later chapters.

2.2 Guts 101

A basic system consists of a control device called a CPU (central processing unit), microprocessor, or microcontroller. There are subtle distinctions between these but we have little need to go very deep at this point. Microcontrollers tend not to be as powerful as standard microprocessors in terms of processing speed but they usually have an array of input/output ports and hardware functions (such as analog-to-digital or digital-to-analog converters) on chip that typical microprocessors do not. To keep things simple we shall use the term "processor" as a generic.

Microprocessors normally are connected to external memory (RAM chips). Microcontrollers generally contain sufficient on-board memory to alleviate this requirement but it is worthwhile to note that we are not talking about large (megabyte) quantities. A microcontroller may only contain a few hundred bytes of memory but in simple applications that may be sufficient. Remember, a byte of memory consists of 8 bits, each bit being thought of as a 1/0, high/low, yes/no, or true/false pair.

In order for a processor to operate on data held in memory, the data must first be copied into a processor's register (it may have dozens of registers). Only in a register can mathematical or logical operations be carried out. For example, if you desire to add one to a variable, the value of the variable must first be copied into a register. The addition is performed on the register contents yielding the answer. This answer is then copied back to the original memory location of the variable. It seems a little roundabout at first, but don't worry, the C language compiler will take care of most of those details for you.

2.3 Memory Maps

Every byte of memory in a computer system has an address associated with it. This is a requirement. Without an address, the processor has no way of identifying a specific location in memory. Generally, memory addressing starts at 0 and works its way up, although some addresses may be special or "reserved" in some systems. That is, a specific address might not refer to normal memory, but instead might refer to a certain input/output port for external communication. Very often it is useful to draw a *memory map*. This is nothing more than a huge array of memory slots. Some people draw them with the lowest (starting) address at the top and other people draw them with the lowest address at the bottom.

Here's an example with just six bytes of memory:

address 0

address 1

address 2

address 3
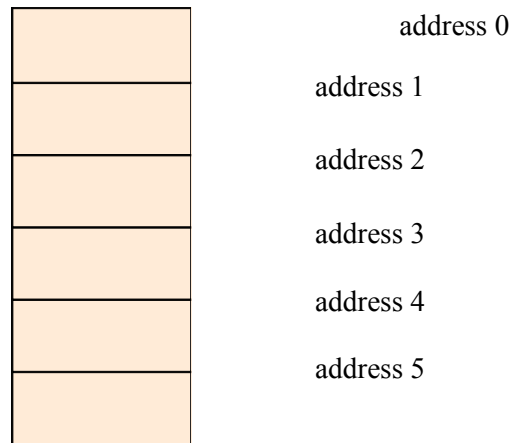
address 4

address 5

Figure 2.1, simple memory map

Each address or slot represents a place we can store one byte. If we had to remember specific addresses we would be doing a lot of work. Instead, the C compiler will keep track of this for us. For example, if we declare a `char` named X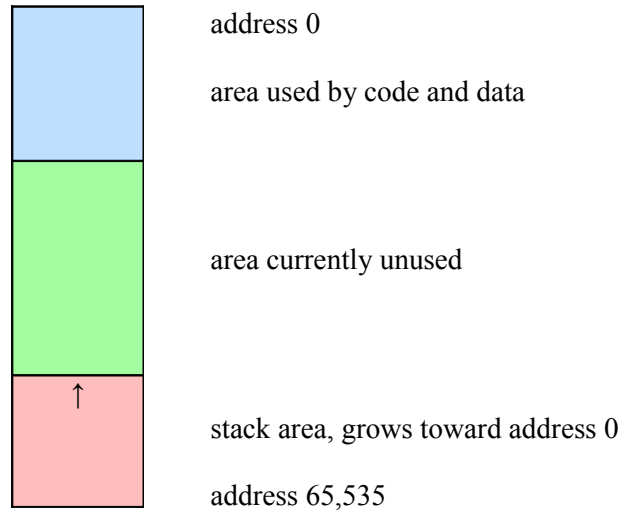, it might be at address 2. If we need to print that value, we don't have to say "fetch the value at address 2". Instead we say; "fetch the value of X" and the compiler generates code to make this work out to the proper address (2). This abstraction eases our mental burden considerably. As many variables require more than one byte, we may need to combine addresses to store a single value. For example, if we chose a `short int`, that needs two bytes. Suppose this variable starts at address 4. It will also require the use of address 5. When we access this variable the compiler automatically generates the code to utilize both addresses because it "knows" we're using a `short int`. Our little six byte memory map could hold 6 `char`, 3 `short int`, 1 `long int` with 1 `short int`, 1 `long int` with 2 `char`, or some other similar combination. It cannot hold a `double` as that requires 8 bytes. Similarly, it could not hold an array of 4 or more `short int` (see Chapter Three for details on numeric data types).

Arrays are of special interest as they must be contiguous in memory. For example, suppose a system has 1000 bytes of memory and a 200 element char array was declared. If this array starts at address 500 then all of the slots from 500 through 699 are allocated for the array. It cannot be created in "scattered" fashion with a few bytes here and a few bytes there. This requirement is due to the manner in which arrays are indexed (accessed), as we shall see later.

2.4 Stacks

Many programs need only temporary storage for certain variables. That is, a variable may only be used for a limited time and then "thrown away". It would be inefficient to allocate permanent space for this sort of variable. In its place, many systems use a *stack*. Ordinarily, an application is split into two parts, a code section and a data section. The data section contains the "permanent" (global) data. As these two will not consume the entire memory map, the remainder of the memory is often used for temporary storage via a stack. The stack starts at the opposite end of the memory map and grows toward the code and data sections. It is called a First-In-Last-Out stack or FILO stack. It works like a stack of trays in a cafeteria. The first tray placed on the stack will be the last one pulled off and vice versa. When temporary

variables are needed, this memory area is used. As more items are needed, more memory is taken up. As our code exits from a function, the temporary (`auto`) variables declared there are no longer needed, and the stack shrinks. If we make many, many function calls with many, many declared variables, it is possible for the stack to overrun the code and data sections of our program. The system is now corrupt, and proper execution and functioning of the program are unlikely.

address 0

area used by code and data

area currently unused

↑

stack area, grows toward address 0

address 65,535

Figure 2.2, basic memory layout

Above is a memory map example of a system with 64k bytes of memory (k=1024 or 2^{10}). Individual memory slots are not shown. Only the general areas are shown.

It is worthwhile to note that in some systems, code and data are in a common area as shown (Von Neumann architecture) while in others they are physically split (Harvard architecture). Whether split or not, the basic concepts remain. So, why would we want to split the two areas, each accessed via its own memory bus[1]? Simple, separating the code and data allows the processor to fetch the next instruction (code) using a memory bus that is physically separate from the data bus it is currently accessing. A shared code/data memory bus would require special timing to coordinate this process as only one thing can be on the bus at any given time. Having two separate memory buses will speed execution times.

---

[1] A *bus* typically refers to a collection of wires or connections upon which multiple data bits (or address bits) are sent as a group.

3. C Language Basics

3.1 Introduction

C is a terse language. It is designed for professional programmers who need to do a lot with a little code quickly. Unlike BASIC or Python, C is a compiled language. This means that once you have written a program, it needs to be fed into a compiler that turns your C language instructions into machine code that the microprocessor or microcontroller can execute. This is an extra step, but it results in a more efficient program than an interpreter. An interpreter turns your code into machine language while it's running, essentially a line at a time. This results in slower execution. Also, in order to run your program on another machine, that machine must also have an interpreter on it. You can think of a compiler as doing the translation all at once instead of a line at a time.

Unlike many languages, C is not line oriented, but is instead free-flow. A program can be thought of as consisting of three major components: Variables, statements and functions. Variables are just places to hold things, as they are in any other language. They might be integers, floating point (real) numbers, or some other type. Statements include things such as variable operations and assignments (i.e., set x to 5 times y), tests (i.e., is x more than 10?), and so forth. Functions contain statements and may also call other functions.

3.2 Variable Naming, Types and Declaration

Variable naming is fairly simple. Variable names are a combination of letters, numerals, and the underscore. Upper and lower case can be mixed and the length is typically 31 characters max, but the actual limit depends on the C compiler in use. Further, the variable name cannot be a reserved (key) word nor can it contain special characters such as `.  ;  ,  *  -` and so on. So, legal names include things like `x`, `volts`, `resistor7`, or even `I_Wanna_Go_Home_Now`.

C supports a handful of variable types. These include floating point or real numbers in two basic flavors: `float`, which is a 32 bit number, and `double`, which is a higher precision version using 64 bits. There are also a few integer types including `char`, which is 8 bits, `short int`, which is 16 bits, and `long int`, which is 32 bits. As `char` is 8 bits, it can hold 2 to the 8th combinations, or 256 different values. This is sufficient for a single ASCII character, hence the name. Similarly, a `short int` (or `short`, for short!) can hold 2 to the 16th combinations, or 65,536 values. `chars` and `ints` may be `signed` or `unsigned` (`signed`, allowing negative values, is the default). There is also a plain old `int`, which might be either 16 or 32 bits, depending on which is most efficient for the compiler (to be on the safe side, never use plain old `int` if the value might require more than 16 bits).

Sometimes you might also come across special double long integers (also called long longs) that take up 8 bytes as well as 80 bit extended precision floats (as defined by the IEEE).

Here is a table to summarize the sizes and ranges of variables:

Variable Type	Bytes Used	Minimum	Maximum
char	1	−128	127
unsigned char	1	0	255
short int	2	−32768	32767
unsigned short int	2	0	65535
long int	4	≈ −2 billion	≈ 2 billion
unsigned long int	4	0	≈ 4 billion
float (6 significant digits)	4	± 1.2 E −38	± 3.4 E +38
double (15 significant digits)	8	± 2.3 E −308	± 1.7 E +308

Figure 3.1, numeric types and ranges

C also supports arrays and compound data types. We shall examine these in a later segment.

Variables must be declared before they are used. They cannot be created on a whim, so to speak, as they are in Python. A declaration consists of the variable type followed by the variable name, and optionally, an initial value. Multiple declarations are allowed. Here are some examples:

```
char x;               declares a signed 8 bit integer called x
unsigned char y;      declares an unsigned 8 bit integer called y
short z, a;           declares two signed 16 bit integers named z and a
float b =1.0;         declares a real number named b and sets its initial value to 1.0
```

Note that each of these declarations is followed with a semi-colon. The semi-colon is the C language way of saying "This statement ends here". This means that you can be a little sloppy (or unique) in your way of dealing with spaces. The following are all equivalent and legal:

```
float b = 1.0;
float b=1.0;
float    b   =   1.0 ;
```

3.3 Functions

Functions use the same naming rules as variables. All functions use the same template that looks something like this:

```
return_value function_name( function argument list )
{
       statement(s)
}
```

Figure 3.1, basic function template

You might think of the function in the mathematical sense. That is, you give it some value(s) and it gives you back a value. For example, your calculator has a sine function. You send it an angle and it gives you back a value. In C, functions may have several arguments, not just one. They might not even have an argument. Also, C functions may return a value, but they don't have to. The "guts" of the function are defined within the opening and closing brace pair `{}`. So, a function which takes two integers, x and y, as arguments, and returns a floating point value will look something like this:

```
float my_function( int x, int y )
{
       //...appropriate statements here...
}
```

If the function doesn't take or return values, the word `void` is used. If a function neither requires values nor returns a value, it would look like:

```
void other_function( void )
{
       //...appropriate statements here...
}
```

This may appear to be extra fussy work at first, but the listing of data types makes a lot of sense because C has something called *type checking*. This means that if you try to send a function the wrong kind of variable, or even the wrong number of variables, the compiler will warn you that you've made a mistake! Thus if you try to send `my_function()` above two floats or three integers, the compiler will complain and save you a big headache during testing.

All programs must have a place to start, and in C, program execution begins with a function called `main`. This does not have to be the first function written or listed, but all programs must have a function called `main`. Here's our first program, found in Figure 3.2, following:

```
    /* Our first program */

    void main( void )
    {
          float x = 2.0;
          float y = 3.0;
          float z;

          z = x*y/(x+y);
    }
```

Figure 3.2, a simple program

There is only one function here, `main()`. It takes no variables and returns nothing. What's the other stuff? First, the `/* */` pair denotes a comment[2]. Anything inside of the comment pair is ignored by the compiler. A C comment can stretch for many lines. Once inside the function, three variables are declared with two of them given initial values. Next, the variables `x` and `y` are multiplied together, divided by their sum, and assigned to `z`. As C is free-flow, an equivalent (but ugly) version is:

```
    /* Our first program */ void main(void){
    float x=2.0;float y=3.0;float z;z=x*y/(x+y);}
```

Figure 3.3, alternate format (to be avoided)

This is the complete opposite of Python which has very rigid spacing and formatting rules.

Now, suppose that this add, multiply, divide operation is something that you need to do a lot. We could split this off into a separate function. Our program now looks like Figure 3.4 on the following page:

---

[2] C also allows // to denote a single line comment without the "backend pairing".

```
    /* Our second program */

    float add_mult_div( float a, float b )
    {
        float answer;

        answer = a*b/(a+b);
        return( answer );
    }

    void main( void )
    {
        float x = 2.0;
        float y = 3.0;
        float z;

        z = add_mult_div( x, y );
    }
```

Figure 3.4, program with separate function

The new math function takes two `floats` as arguments and returns a `float` to the caller. The compiler sees the new function before it is used in `main()`, thus, it already "knows" that it should be sent two `floats` and that the return value must be assigned to a `float`. It is very important to note that the new math function uses different variable names (`a` and `b`) from the caller (`x` and `y`). The variables in the new math function are really just place-holders. The values from the original call (`x` and `y`) are copied to these new variables (`a` and `b`) and used within the new function. As they are copies, they can be altered without changing the original values of `x` and `y`. In this case, `x` and `y` are said to be *local* to the `main()` function while `a` and `b` are *local* to the `add_mult_div()` function. In other words, `a` isn't visible from `main()` so you can't accidentally alter it! Similarly, `x` isn't visible from `add_mult_div()`, so you can't accidentally alter it either. This is a positive boon when dealing with large programs using many variable names. While it's not usually preferred, there are times when you want a variable to be known "everywhere". These are called *global* items. You can make variables *global* by simply declaring them at the beginning of the program outside of the functions (i.e., right after that initial comment in our example).

3.4 Libraries

The examples above are rather limited because, although they perform a calculation, we have no way of seeing the result! We need some way to print the answer to the computer screen. To do this, we rely on system functions and libraries. There are a series of libraries included with most C development systems to cover a variety of needs. Essentially, someone has already coded, tested and compiled a bunch of functions for you. You add these functions to your program through a process called *linking*. Linking simply combines your compiled code along with any required library code into a final executable program. For basic printouts, data input, and the like, we use the *standard IO (Input/Output) library*, or *stdio* for short. There is a function in this library named `printf()` for "print formatted". So that the

compiler can do type checking, it must know something about this new function. We tell the compiler to look into a special file called a *header file* to find this information. Every library will have an associated header file (usually of the same name) and it will normally end with a `.h` file extension[3]. The compiler directive is called an `include` statement.

```
// Our third program, this is an example of a single line comment

#include <stdio.h>

void main( void )
{
      printf("Hello world.\n");
}
```

Figure 3.5, program with library function call

This program simply prints the message *Hello world.* to the screen. The backslash-n combo is a special formatting token that means *add a new line* (i.e., bring the cursor to the line below). If we did not add the `#include` directive, the compiler wouldn't know anything about `printf()`, and would complain when we tried to use it. So, what's in a header file? Well, among other things they contain *function prototypes*. The prototypes are nothing more than a template. You can create your own by cutting and pasting your function name with argument list and adding a semicolon to it. Here is the function prototype for our earlier math function:

```
float add_mult_div( float a, float b );
```

You could make your own library of functions if you want. To use them, all you'd need is an appropriate `include` statement in your code, and remember to add in your library code with the linker. This will allow you to reuse code and save time. We will look at multiple file projects and more on libraries in a later segment.

Consequently, if we want to print out the answer to the first program, we'd wind up with something like Figure 3.6 on the following page:

---

[3] It is worth noting that a large number of subgroups of code are collected together into what is referred to as the *C standard library*. The precise implementation of this is dependent on the operating system, however, the header files (e.g., stdio.h, string.h, math.h, etc.) remain distinct.

```
      /* Our fourth program */

      #include <stdio.h>

      void main( void )
      {
            float x = 2.0;
            float y = 3.0;
            float z;

            z = x*y/(x+y);

            printf("The answer is %f\n", z);
      }
```

Figure 3.6, a more complete program

The `%f` in the `printf()` function serves as a place holder for the variable `z`. If you need to print several values you can do something like this:

```
      printf("The answer from %f and %f is %f\n", x, y, z);
```

In this case, the first `%f` is used for `x`, the second `%f` for `y`, and the final one for `z`. The result will look like:

```
      The answer from 2.0 and 3.0 is 1.2
```

3.5 Some Simple Math

C uses the same basic math operators as many other languages. These include +, −, /(divide), and *(multiply). Parentheses are used to group elements and force hierarchy of operations. C also includes `%` for modulo. Modulo is an integer operation that leaves the remainder of a division, thus 5 modulo 7 is 2.

The divide behaves a little differently with integers than with floats as there can be no remainder. Thus 9 integer divide 4 is 2, not 2.25 as it would be if you were using floats. C also has a series of bit manipulators that we will look at a little later. For higher math operations, you will want to look at the math library (`math.h` header file). Some examples are `sin()`, `cos()`, `tan()`, `log10()` (common log) and `pow()` for powers and roots. Do **not** try to use ^ as you do on many calculators. `x` raised to the `y` power is **not** x^y but rather `pow(x, y)`. The ^ operator has an entirely different meaning in C! Recalling what we said earlier about libraries, if you wanted to use a function like `sin()` in your code, you'd have to tell the compiler where to find the prototype and similar info. At the top of your program you'd add the line:

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

A final caution: The examples above are meant to be clear, but not necessarily the most efficient way of doing things. As we shall see, sometimes the way you code something can have a huge impact on its performance. Given the power of C, expert programmers can sometimes create code that is nearly indecipherable for ordinary people. There is a method behind the apparent madness.

3.6 The program creation/development cycle

To create a C program:

1. Do the requisite mental work. This is the most important part.
2. Create the C source code. This can be done using a text editor, but is normally done within the IDE (Integrated Development Environment). C source files are plain text and saved with a ".c" extension.
3. Compile the source code. This creates an assembly output file. Normally, compiling automatically fires up the assembler, which turns the assembly file into a machine language output file.
4. Link the output file with any required libraries using the linker. This creates an executable file. For desktop development, this is ready to test.
5. For embedded development, download the resulting executable to the target hardware (in our case, the Arduino development board). For the Arduino, steps 3, 4, and 5 can be combined by selecting "Build" from the IDE menu.
6. Test the executable. If it doesn't behave properly, go back to step one.

3.7 Summary

Here are some things to keep in the back of your mind when learning C:

- C is terse. You can do a lot with a little code.
- As it allows you to do almost anything, a novice can get into trouble very quickly.
- It is a relatively thin language, meaning that most "system functions" are not part of the language per se, but come from link-time libraries.
- Function calls, function calls, and more function calls!
- Source code is free flow, not line oriented. A "line" of code is usually terminated with a semicolon.
- Shortcuts allow experts to create code that is almost indecipherable by normal programmers.
- All variables must be declared before use (not free flow as in Python).
- Variables can be global or local in scope. That is, a local variable can be "known" in one place of the program and not in another.

3.8 Exercises

1. Write a C code comment that includes your name and the date. Use both the single line and the multi-line styles.

2. Write a function that will take three floating point values as arguments. The function should return the average value of the three arguments.

3. Write a program that will print out your name.

4. C Basics II

4.1 Input and Output

We've seen the use of `printf()` to send information to the computer screen. `printf()` is a very large and complicated function with many possible variants of format specifiers. Format specifiers are the "% things" used as placeholders for values. Some examples are:

`%f`	float
`%lf`	double (long float)
`%e`	float using exponent notation
`%g`	float using shorter of `e` or `f` style
`%d`	decimal integer
`%ld`	decimal long integer
`%x`	hexadecimal (hex or base 16) integer
`%o`	octal (base 8) integer
`%u`	unsigned integer
`%c`	single character
`%s`	character string

Figure 4.1, print format types

Suppose that you wanted to print out the value of the variable `ans` in decimal, hex, and octal. The following instruction would do it all:

```
printf("The answer is %d, or hex %x, or octal %o.\n", ans, ans, ans );
```

Note how the three variables are labeled. This is important. If you printed something in hex without some form of label, you might not know if it was hex or decimal. For example, if you just saw the number "23", how would you know it's 23 decimal or 23 hex (35 decimal)? For that matter, how would you set a hex constant in your C code? The compiler would have no way of "knowing" either. To get around this, hex values are prefixed with `0x`. Thus, we have `0x23` for hex 23. The `printf()` function does not automatically add the `0x` on output. The reason is because it may prove distracting if you have a table filled only with hex values. It's easy enough to use `0x%d` instead of just `%d` for the output format.

You can also add a field width specifier. For example, `%5d` means print the integer in decimal with 5 spaces minimum. Similarly, `%6.2f` means print the floating point value using 6 spaces minimum. The ".2" portion is a precision specifier and in this case indicates 2 digits after the decimal point are to be used. As you can see, this is a very powerful and flexible function!

The mirror input function is `scanf()`. This is similar to Python's `input` statement. Although you can ask for several values at once, it is generally best to ask for a single value when using this function. It uses the same sort of format specifiers as `printf()`. There is one important point to note. The `scanf()` function needs to know where to place the entered value in computer memory. Simply informing it of the name of the variable is insufficient. You must tell it where in memory the variable is, in other words, you

must specify the address of the variable. C uses the `&` operator to signify "address of". For example, if you wish to obtain an integer from the user and place it in a variable called `voltage`, you might see a program fragment like so…

```
printf("Please enter the voltage:");
scanf("%d", &voltage);
```

It is very common for new programmers to forget the `&`. Be forewarned!

4.2 Variable Sizes

A common question among new programers is "Why are there so many sizes of variables available?" We have two different sizes of reals; `float` at 32 bits, and `double` at 64 bits. We also have three different sizes of intgers at 8, 16, and 32 bits each[4]. In many languages, there's just real and integer with no size variation, so why does C offer so many choices? The reason is that "one size **doesn't** fit all". You have options in order to optimize your code. If you have a variable that ranges from say, 0 to 1000, there's no need to use more than a short (16 bit) integer. Using a 32 bit integer simply uses more memory. Now, you might consider 2 extra bytes to be no big deal, but remember that we are talking about embedded controllers in some cases, not desktop systems. Some small controllers may have only a few hundred bytes of memory available for data. Even on desktop systems with gigabytes of memory, choosing the wrong size can be disastrous. For example, suppose you have a system with an analog to digital converter for audio. The CD standard sampling rate is 44,100 samples per second. Each sample is a 16 bit value (2 bytes), producing a data rate of 88,100 bytes per second. Now imagine that you need enough memory for a five minute song in stereo. That works out to nearly 53 megabytes of memory. If you had chosen long (32 bit) integers to hold these data, you'd need about 106 megabytes instead. As the values placed on an audio CD never exceed 16 bits, it would be foolish to allocate more than 16 bits each for the values. Data sizes are power-of-2 multiples of a byte though, so you can't choose to have an integer of say, 22 bits length. It's 8, 16, or 32 for the most part (some controllers have an upper limit of 16 bits).

In the case of `float` versus `double`, `float` is used where space is at a premium. It has a smaller range (size of exponent) and a lower precision (number of significant digits) than `double`. `double` is generally preferred and is the norm for most math functions. Plain floats are sometimes referred to as *singles* (that is, single precision versus double precision).

If you don't know the size of a particular data item (for example an `int` might be either 16 or 32 bits depending on the hardware and compiler), you can use the `sizeof()` command. This looks like a function but it's really built into the language. The argument is the item or expression you're interested in. It returns the size required in bytes.

```
size = sizeof( int );
```

`size` will be either 2 or 4 depending on the system.

---

[4] In some systems 80 bit doubles and/or 64 bit integers are also available.

4.3 More Math

OK, so what happens if you add or multiply two `short int` together and the result is more than 16 bits long? You wind up with an overrange condition. Note that the compiler cannot warn you of this because whether or not this happens will depend entirely on values entered by the user and subsequently computed within the program. Hopefully, you will always consider maximum value cases and choose appropriate data sizes and this won't be a problem. But what actually happens? To put it simply, the top most bits will be ignored. Consider an 8 bit unsigned integer. It goes from 0 to 255. 255 is represented as eight 1s. What happens if you add the value 1 to this? You get a 9 bit number: a 1 followed by eight 0s. That ninth bit is thrown away as the variable only has eight bits. Thus, 255 plus 1 equals 0! This can create some serious problems! For example, suppose you wanted to use this variable as a loop counter. You want to go through a loop 500 times. The loop will never terminate because an 8 bit integer can't go up that high. You keep adding one to it, but it keeps flipping back to 0 after it hits 255. This behavior is not all bad; it can, in fact, be put to good use with things like interrupts and timers, as we shall see.

What happens if you mix different types of variables? For example, what happens if you divide a `double` by an `int` or a `float` by `double`? C will *promote* the lesser size/precision types to the larger type and then do the operation. This can sometimes present a problem if you try to assign the result back to something smaller, even if you know it will always "fit". The compiler will complain if you divide a `long int` by another `long int` and try to assign the result to a `short int`. You can get around this by using a *cast*. This is your way of telling the compiler that you know there is a potential problem, but to go ahead anyway (hopefully, because you know it will always work, not just because you want to defeat the compiler warning). Casting in C is similar to type conversion in Python (e.g., the `int()` function). Here's an example.

```
short int x, y=20;
long int z=3;

x=(short int)(y/z);
```

Note how you are directing the compiler to turn the division into a `short int`. Otherwise, the result is in fact a `long int` due to the promotion of `y` to the level of `z`. What's the value of `x`? Why it's 6 of course! Remember, the fractional part is meaningless, and thus lost, on integer divides.

Casting is also useful when using math functions. If you have to use `float`, you can cast them to/from `double` to make use of functions defined with `double`. For example, suppose a, b, and c are declared as `float` but you wish to use the `pow()` function to raise a to the b power. `pow()` is defined as taking two `double` arguments and returning a `double` answer.

```
c = (float)pow( (double)a, (double)b );
```

This is a very explicit example. Many times you can rely on a "silent cast" promotion to do your work for you as in the integer example above. Sometimes being explicit is a good practice just as a form of documentation.

4.4 Bitwise Operations

Sometimes you'd like to perform bitwise operations rather than ordinary math. For example, what if you want to logically AND two variables, bit by bit? Bitwise operations are very common when programming microcontrollers as a means of setting, clearing and testing specific bits in control registers (for example, setting a specific pin on a digital port to read mode instead of write mode). C has a series of bitwise operators. They are:

```
&       AND
|       OR
^       XOR
~       One's Complement
>>      Shift Right
<<      Shift Left
```

Figure 4.2, bitwise operators

Note the double use of `&` for "address of" and now AND. The unary operation is always "address of", and the binary operation is always AND, so `a & b` would **not** imply the address of `b`. If you wanted to AND `x` with `y`, shift the result 2 places to the left and assign the result to `z`, you'd use:

```
z = (x&y)<<2;
```

Let's look at a few examples. Suppose the variables `X`, `Y` and `Z` are `unsigned char`s. `X` and `Y` are set to 13 and 134, respectively. In hex, that's 0x0d and 0x86 for bit patterns of 00001101 and 10000110.

```
Z = X<<3;   // Z is 01101000 or 0x68
Z = X>>1;   // Z is 00000110 or 0x06
Z = ~X;     // Z is 11110010 or 0xf2
Z = X|Y;    // Z is 10001111 or 0x8f
Z = X&Y;    // Z is 00000100 or 0x04
Z = X^Y;    // Z is 10001011 or 0x8b
```

4.5 Setting, Clearing and Reading Register Bits

Bitwise operations may appear to be somewhat arcane to the uninitiated but are in fact commonly used. A prime use is in setting, clearing and testing specific bits in registers. One example involves configuring bidirectional ports for input or output mode via a *data direction register*, typically abbreviated DDR. Each bit of the DDR represents a specific output pin. A logic high might indicate output mode while a logic low would indicate input mode. Assuming DDR is an 8 bit register, if you wanted to set all bits except the 0th bit to input mode, you could write[5]:

```
DDR = 0x01; // set bit zero to output mode
```

---

[5] In C, bit position counting, like most sequences, starts from position 0 not position 1.

If sometime later you wanted to also set the 1ˢᵗ and 2ⁿᵈ bits to output mode while keeping everything else intact, the easy way to do it is simply to OR the bits you want:

```
DDR = DDR | 0x06;
```

The prior operation may be performed using the more common shorthand:

```
DDR |= 0x06;
```

Note that the preceding code does not affect any of the other bits so they stay in whatever mode they were originally. By the way, a set of specific bits (such as the 0x06 above) is often referred to as a *bit pattern* or *bitmask*.

To see if a specific bit is set, simply AND instead of OR. So, to see if the 1ˢᵗ bit of DDR is set for output mode, you could use something like:

```
if ( DDR & 0x02 )        // true if set
```

Clearing bits requires ANDing with a bitmask that has been complemented. In other words, all 1s and 0s have been reversed in the bit pattern. If, for example, we want to clear the 0ᵗʰ and 4ᵗʰ bits, we'd first complement the bit pattern 0x11 yielding 0xee. Then we AND:

```
DDR &= 0xee;
```

Often, it's easier to just use the logical complement operator on the original bit pattern and then AND it:

```
DDR &= (~0x11);
```

If you're dealing with a single bit, you can use the left shift operator so you don't even have to bother figuring out the bit pattern in hex. To set the 3ʳᵈ bit and then clear the 4ᵗʰ bit of DDR, you could use the following:

```
DDR |=  (0x01<<3);
DDR &= ~(0x01<<4);
```

These operations are so common that they are often invoked using an in-line expansion via a `#define`.

4.6 #define

Very often it is desirable to use symbolic constants in place of actual values. For example, you'd probably prefer to use a symbol such as PI instead of the number 3.14159. You can do this with the `#define` preprocessor directive. These are normally found in header files (such as stdio.h or math.h) or at the top of a module of C source code. You might see something like:

```
#define PI 3.14159
```

Once the compiler sees this, every time it comes across the token `PI` it will replace it with the value 3.14159. This directive uses a simple substitution but you can do many more complicated things than this. For example, you can also create something that looks like a function:

```
#define parallel((x),(y))     ((x)*(y))/((x)+(y))
```

The x and y serve as placeholders. Thus, the line

```
a = parallel( b, c );
```

gets expanded to:

```
a = (a*b)/(a+b);
```

Why do this? Because it's an *in-line expansion* or *macro*. That means that there's no function call overhead and the operation runs faster. At the same time, it reads like a function, so it's easier for a programmer to follow. OK, but why all the extra parentheses? The reason is because x and y are placeholders, and those items might be expressions, not simple variables. If you did it this way you might get in trouble:

```
#define parallel(x,y)   x*y/(x+y)
```

What if x is an expression, as in the following example?

```
a = parallel(2+b,c);
```

This would expand to:

```
a = 2+b*c/(2+b+c);
```

As multiplication is executed before addition, you wind up with 2 being added to the product of b times c *after* the division, which is not the same as the sum of 2 and b being multiplied by c, and that quantity then being divided. By using the extra parentheses, the order of execution is maintained.

Referring back to the bit field operations, here are some useful definitions for what appear to be functions but which are, in fact, bitwise operations expanded in-line:

```
#define bitRead(value, bit)   (((value) >> (bit)) & 0x01)
#define bitSet(value, bit)    ((value) |= (1UL << (bit)))
#define bitClear(value, bit)  ((value) &= ~(1UL << (bit)))
```

The 1UL simply means 1 expressed as an unsigned long. Finally, bit could also be defined as a symbol which leads to some nice looking self-documenting code:

```
#define LEDBIT 7

// more code here...

bitSet( DDR, LEDBIT );
```

#define expansions can get quite tricky because they can have nested references. This means that one #define may contain within it a symbol which is itself a #define. Following these can be a little tedious at times but ultimately are worth the effort. We shall look at a few down the road. Remember, these are done to make day-to-day programming easier, not to obfuscate the code. For now, start with simple math constant substitutions. They are extremely useful and easy to use. Just keep in the back of your mind that, with microcontrollers, specific registers and ports are often given symbolic names such as

PORTB so that you don't have to remember the precise numeric addresses of them. The norm is to place these symbolic constants in ALL CAPS.

4.7 Keywords

Here is a list of keywords in the C language:

```
auto        break       case        char        const
continue    do          default     double      else
entry       extern      float       for         goto
if          int         long        register    return
sizeof      short       static      struct      switch
typedef     union       unsigned    volatile    while
```

Figure 4.3, C language keywords

We've looked at quite a few of these already. Some that we haven't you can probably guess the use of. As stated previously, C is a "skinny" language!

4.8 Exercises

1. Write a line of code that will print the statement "The result is *x* volts" where *x* is the value given by the floating point variable `output_voltage`.

2. Write a line of code to define a constant called `RECIP2PI` that is equal to $1/(2\pi)$.

3. Write the code to determine the number of bytes required for a variable called `POWER_SUPPLY`.

4. Assume the 8 bit variable `X` exists. Write the code to set the MSB (most significant bit), leaving all other bits unchanged.

5. Assume the 8 bit variable `Y` exists. Write the code to set the LSB (least significant bit), leaving all other bits unchanged.

6. Assume the 8 bit variable `Z` exists. Write the code to clear the MSB and LSB, leaving all other bits unchanged.

7. Assume the 8 bit variable `W` exists. Write the code to complement each bit (flip 0 to 1 and 1 to 0).

8. If `X` is 0x04 and `Y` is 0x09, what are a) `X|Y`, b) `X&Y`, c) `~X`, d) `0xf1&Y`?

9. If `X` is 0xf0 and `Y` is 0x11, what are a) `X|Y`, b) `X&Y`, c) `~X`, d) `0xf1&Y`?

5. C Storage Types and Scope

5.1 Types

C has several ways of storing or referencing variables. These affect the way variables behave. Some of the more common ones are: auto, register, and static.

Auto variables are variables declared within functions that are not static or register types. That is, the `auto` keyword is the default. Normally, auto variables are created on the application's stack, although C doesn't require this. The stack is basically a chunk of memory that is allocated for the application's use when it is first run. It is a place for temporary storage, with values popped onto and pulled off of the stack in first-in, last-out order (like a stack of plates). Unless you initialize an auto variable, you have no idea what its value is when you first use it. Its value happens to be whatever was in that memory location the previous time it was used. It is important to understand that this includes subsequent calls to a function (i.e., its prior value is not "remembered" the next time you call the function). This is because any subsequent call to a function does not have to produce the same the memory locations for these variables, anymore than you always wind up with the same plate every time you go to the cafeteria.

Register variables are similar to auto types in behavior, but instead of using the usual stack method, a CPU register is used (if available). The exact implementation is CPU and compiler specific. In some case the `register` keyword is ignored and a simple auto type is used. CPU registers offer faster access than normal memory so register variables are used to create faster execution of critical code. Typically this includes counters or pointers that are incremented inside of loops. A declaration would like something like this:

```
register int x;
```

Static variables are used when you need a variable that maintains its value between function calls. So, if we need a variable that will "appear the way we left it" from the last call, we might use something like this:

```
static char y;
```

There is one important difference between auto and static types concerning initialization. If an auto variable is initialized in a function as so:

```
char a=1;
```

Then `a` is set to `1` each time the function is entered. If you do the same initialization with a static, as in:

```
static char b=1;
```

Then `b` is set to `1` **only** on the first call. Subsequent entries into the function would not incur the initialization. If it did reinitialize, what would be the sense of having a static type? This is explained by the fact that a static does not usually use the stack method of storage, but rather is placed at a fixed memory location. Again, C does not *require* the use of a stack, rather, it is a typical implementation.

Two useful but not very common modifiers are `volatile` and `const`. A `volatile` variable is one that can be accessed or modified by another process or task. This has some very special uses (typically, to prevent an optimizing compiler from being too aggressive with optimizations-more on this later). The `const` modifier is used for declaring constants, that is, variables that should not change value. In some instances this is preferred over using `#define` as type checking is now available (but you can't use the two interchangeably).

5.2 Scope

Scope has to do with where variables are "seen". We have already mentioned the idea of global and local in previous work but it is time to delve a little deeper. Generally, variables only exist within the block they are declared. While it is legal to declare variables inside of a conditional or loop block, we normally declare variables at the very beginning of a function. Consequently, these variables are known within the function. That is, their scope of reference is within the function. Nothing outside of the function knows anything about them. Thus, we say that they are local, or perhaps localized, to the function. For example, consider the two function fragments below:

```
void func1( void )
{
        int x;
        int y;
        //...some code here...
}

void func2( void )
{
        int y;
        int z;
        //...some other code here...
}
```

There is no direct way to access the `z` variable of `func2()` from `func1()`. Likewise, there is no direct way to access the `x` variable of `func1()` from `func2()`. More interestingly, the `y` variables of `func1()` and `func2()` are entirely different! They do **not** refer to the same variable. This sometimes can be confusing for new programmers but it is essential for large programs. Imagine that you were working with a team of people on a very large program, perhaps tens of thousands of lines long. If the idea of local scope did not exist, you'd have to make sure that every single variable in the program had a unique name! This would create a nightmare of confusion. By using local scope, you're saying: "I only need this variable for a job within this function. As it only needs to exist within this function, its name is only meaningful within this function."

If some form of "universally known" data item is needed, we can resort to the *global*. Globals act pretty much like statics and are usually stored the same way. If you have a program that consists of a single file, you can declare your globals by listing them at the beginning of the program before (and outside of) any functions. In this way they will be read by the compiler first and will therefore be "known" to all functions that follow. Do not get in the habit of declaring all variables as global. This is considered a bad and inefficient coding method. Get into the habit of using locals as the norm and resort to globals only when called for.

If you have a multiple file project, how do you get the functions in the second file to recognize the globals declared in the first file? In this case, you'll create a header file and use the `#include` directive. For example, suppose your project consists of two C source files named foo.c and bar.c.

In foo.c you declare the following global:

```
int m;
```

In order for the functions in bar.c to "see" m, you'll create a header file, perhaps called foo.h. foo.h will contain the following:

```
extern int m;
```

Meaning that an integer named m has been declared externally (i.e., in another file). At the top of bar.c you'll add the line:

```
#include <foo.h>
```

So, when bar.c is compiled, the compiler will first open up foo.h. It will see that the variable m has been declared elsewhere and puts it in a "known variables list". It then continues to compile the remainder of you code. When it comes across m in some function, the compiler "understands" that this is a variable that was declared in another file. No problem!

So, you can now see that header files are largely composed of definitions and declarations from other places, namely external data and function prototypes.

5.4 Exercises

1. Assume a function declares a variable like so: `static int x=0;` The function increments the variable and then prints its value. What does the function print out on the tenth call to the function? How would this change if the `static` keyword was not used?

2. Consider the following snippet of code:

```
void doug( void )
{
        int x=0;

        x=x+1;
        printf( "%d\n", x );
}
void dinsdale( void )
{
        int x=20;

        x=x+1;
        printf( "%d\n", x );
}
```

Suppose you call `doug()` five times in a row and then call `dinsdale()` five times in a row. What would the resulting output look like?

6. C Arrays and Strings

6.1 Introduction

Up to now we haven't talked much about character strings, that is, variables that contain non-numeric data such as a person's name or address. There is no string variable type in C (unlike Python). In C, strings are nothing more than arrays of characters. Arrays are a simple grouping of like variables in a sequence, each with the same name and accessed via an index number. They behave similarly to arrays in most other languages (or lists in Python). C arrays may have one, two, or more dimensions. Here are a few example declarations:

```
float results[10];      An array of 10 floats
long int x[20];         An array of 20 longs, or 80 bytes
char y[10][15];         A two-dimension array, 10 by 15 chars each, 150 bytes total
```

Note the use of square brackets and the use of multiple sets of square brackets for higher dimension arrays. Also, C arrays are counted from index 0 rather than 1.[6] For example, the first item of `results[]` is `results[0]`. The last item is `results[9]`. There is no such item here as `results[10]`. That would constitute an illegal access. You can pre-initialize arrays by listing values within braces, each separated by a comma:

```
double a[5] = {1.0, 2.0, 4.7, -177.0, 6.3e4};
```

If you leave the index out, you get as many elements as you initialize:

```
short int b[] = {1, 5, 6, 7}; /* four elements */
```

If you declare more than you initialize, the remainder are set to zero if the array is global or static (but not if auto).

```
short int c[10] = {1, 3, 20}; /* remaining 7 are set to 0 */
```

If you are dealing with character strings you could enter the ASCII codes for the characters, but this would be tedious in most cases. C let's you specify the character in single quotes and it will do the translation:

```
char m[20] = {'m', 'y', ' ', 'd', 'o', 'g', 0};
```

(The reason for the trailing 0 will be evident in a moment.) Even easier is to specify the string within double quotes:

```
char n[20]={"Bill the cat"};
```

Consider the string `n[]` above. It contains 12 characters but was declared to hold 20. Why do this? Well, you may need to copy some other string into this variable at a future time. By declaring a larger value, the variable can hold a larger string later. At this point you might be wondering how the C library functions "know" to use just a portion of the allocated space at any given time (in this case, just the first 12

---

[6] The reason for this will be apparent when we cover addresses and pointers.

characters). This is possible through a simple rule: **All strings must be null terminated.** That is, the character after the final character in use must be null, numerically 0. In the case of the direct string initialization of `n[]`, the null is automatically added after the final character, *t*. In the case of the character-by-character initialization of `m[]`, we had to do it manually. The null is extremely important. Functions that manipulate or print strings look for the trailing null in order to determine when their work is done. Without a null termination, the functions will just churn through memory until they eventually hit a null, which may cause system violations and even application or operating system crashes. Note that

```
char my_pet[] = {"fido"};
```

actually declares five characters, not four (four letters plus the terminating null). As C counts from index 0, this declaration is equivalent to:

```
my_pet[0]= 'f';
my_pet[1]= 'i';
my_pet[2]= 'd';
my_pet[3]= 'o';
my_pet[4]=  0;
```

The trailing null may also be written as `'\0'`. It is important to note that without the backslash, this has an entirely different meaning! `'\0'` means null, but `'0'` means the numeral 0.

6.2 String Manipulation

A confusing aspect of C strings for beginners (especially those coming from BASIC or even Python) is how to manipulate them. That is, how do you copy one string to another, compare strings, extract a substring, and so forth? As strings are really arrays, you can't assign one to the other as in `a[] = b[];` Instead, we rely on a series of string functions found in the string library. To use these functions, you need to link your code with the string library and use `#include <string.h>` at the start of your code. To copy one string to another, use `strcpy()`. The template is:

```
strcpy( destination, source );
```

So, if you wanted to copy the contents of `my_pet[]` into `n[]`, you could write:

```
strcpy( &n[0], &my_pet[0] );
```

If you're awake at this point, you might ask "What's with the ampersand?" Good question! What the string copy function needs are the starting addresses of the two arrays. In essence, all it does is copy a character at a time from the source to the destination. When it hits the trailing null it's done. We've already seen the "address of" (`&`) operator earlier when we looked at `scanf()`. So, all we're saying here is "For the source, start at the address of the first character of `my_pet[]`, and for the destination, start at the first character of `n[]`." This can be a little cumbersome, so C offers a shortcut. You can think of `&` and `[]` as sort of canceling each other out. We'd normally write:

```
strcpy( n, my_pet );
```

Note that it is perfectly acceptable to use an index other than zero if you need to copy over just a chunk of the string. You could start copying from index 2 if you'd like, and just get "do" instead of "fido":

```
        strcpy( n, &my_pet[2] );
```

This can also be shortcut by using:

```
        strcpy( n, my_pet+2 );
```

that is, don't start at the address of the first element of `my_pet[]`, start 2 characters later. We'll look at this sort of manipulation much closer when we examine addresses and pointers.

What happens if the source string has more characters than the destination string was allocated to? For example, what if you did this?

```
        strcpy( my_pet, n );
```

This results in a memory overwrite that can accidentally destroy other variables or functions. Very bad! Your program may crash, and in some cases, your operating system my crash. To protect against this, you can use `strncpy()`. This places a limit on the number of characters copied by adding a third argument. As the destination only has space for 5 characters, you'd use:

```
        strncpy( my_pet, n, 5 );
```

This function will stop at 5 characters. Unfortunately, it won't automatically null terminate the string if the limit is reached. To be safe, you'd need to add the following line:

```
        my_pet[4] = 0; /* force null termination */
```

Remember, as C counts from 0, index 4 is the fifth (and final) element. There are many functions available to manipulate strings as well as individual characters. Here is a short list:

`strcmp()`	Compares two strings (alphabetically)
`strcmpi()`	As above, but case insensitive
`strncmp()`	Compares two strings with max length
`strncat()`	Concatenate two strings with max length
`strlen()`	Find length of string (count of chars before null)

Figure 6.1, string functions

The following work on single characters. Again this is just a sampling to give you an idea of what's out there. Use `#include <ctype.h>`

`isupper()`	Determines if character is upper case
`isalpha()`	Determines if character is alphabetic (not numeral, punctuation, etc.)
`tolower()`	Turns character into lower case version

Figure 6.2, character functions

If you don't have library documentation, it can be very instructive to simply open various header files and look at the function prototypes to see what's available. Whatever you do though, **don't edit these files!**

Finally, if you need to convert numeric strings into integer or floating point values, use the functions `atoi()`, `atol()` and `atof()`. (ASCII to int or long int in stdlib.h, ASCII to float in math.h).

6.3 Exercises

1. Write the code to declare an array of 12 single precision real numbers.

2. Write the code to declare an array of 15 eight bit signed integers.

3. Assume that an array of 100 double precision real numbers has been declared and is named `points`. Write the code to print out the first item of `points`. Also, write the code to set the last item of `points` to 0.0.

4. Declare a string called `mammal` and initialize it to the word *woodchuck*.

5. Do you see any potential problems with this snippet of initialization code? If so, explain the issues and how they might be corrected.

```
char bird;

bird[0]='s'
bird[1]='w'
bird[2]='a'
bird[3]='l'
bird[4]='l'
bird[5]='o'
bird[6]='w'
```

7. C Conditionals and Looping

7.1 Conditionals

C uses a fairly standard if/else construct for basic conditionals. They may be nested and each portion may consist of several statements. The condition itself may have multiple elements and be formed in either positive or negative logic. The basic construct is:

```
if( test condition(s)... )
{
      //...do stuff...
}
```

The else portion is optional and looks like:

```
if( test condition(s).. )
{
      //...do stuff...
}
else
{
      //...do other stuff...
}
```

If there is only a single statement in the block (i.e., between the braces), the braces may be removed if desired:

```
if( test condition(s).. )
      //...single statement...
else
      //...do other statement...
```

The test condition may check for numerous possibilities. The operators are:

==	equality
!=	inequality
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

Figure 7.1, relational operators

It is very important to note that equality uses a double equal sign. A single equal sign is an assignment operation. Don't think "equals", think "same as", with one symbol for each word. You may also use Boolean (logic) operators, as shown in Figure 7.2.

Embedded Controllers

```
||     OR
&&     AND
!      NOT
```

Figure 7.2, logical operators

Note that the logical operators do **not** behave the same as the similarly named bitwise operators. For example, a logical AND returns TRUE if its two arguments are non-zero, not necessarily the same bits. That is 1 & 2 yields 0, but 1 && 2 yields TRUE. **TRUE is taken to be any non-zero value.** Any variable or expression that evaluates to a value other than zero is logically TRUE. If the result is zero, then it is logically FALSE. Time for some examples. The conditional is written as a fragment with an explanation following:

```
if( a==6 )
      /* taken only if the variable a is a 6 */

if( b!=7 )
      /* taken as long as the variable b isn't 7 */

if( (a==6) && (b!=7) )
      /* taken as long as a is 6 and b is something other than 7 */

if( (a==6) || (b!=7) )
      /* taken as long as a is 6 or b is something other than 7 */

if( a==0 )
      /* taken if a is zero */

if( !a )
      /* another way of saying taken if a is zero */

if( a!=0 )
      /* taken if a is not zero */

if( a )
      /* another way of saying taken if a is not zero */
```

How you word something is up to you. The following two code fragments are equivalent:

```
if( a==b )
      do_x();
else
      do_y();

if( a!=b )
      do_y();
else
      do_x();
```

It is very common for new programmers to use = when they want ==. This can have disastrous results. Consider the following code fragment:

```
if( a=b )
```

What does this do? At first glance, you might think it tests to see if `a` and `b` are the same. It does nothing of the kind! Instead, it assigns the value of `b` to `a` and then checks to see if that value is non-zero. In other words, it does this:

```
a=b;
if( a )
```

A trick to help you with this, at least with constants, is to reverse the normal order. Instead of writing `if( a==6 )`, use `if( 6==a )`. This way, if you accidentally use a single equal sign, the compiler will cough up a syntax error.

7.2 Nesting

If a multiple condition won't cut it, you can nest if/else tests. Nesting conditionals is easy:

```
if( test condition(s).. )
{
      if( other tests.. )
      {
      }
      else
      {
      }
}
else
{
      if( still other tests.. )
      {
      }
      else
      {
      }
}
```

You can go many levels deep if you desire. Note that C, unlike Python, doesn't *require* the indenting shown, but it is expected formatting. For selection of a single value out of a list, you can use the switch/case construct. The template looks like:

```
switch( test_variable )
{
      case value_1:
            //...do stuff...
            break;
      case value_2:
            //...do other stuff...
            break;
      default:
            //...do stuff for a value not in the list...
            break;
}
```

The `default` section is optional. Also, it does not have to be the final item in the list. If a `break` is left out, the code will simply fall through to the next case, otherwise code execution jumps to the closing brace. Also, cases can be stacked together. The action statements for each case may include any legal statements including assignments, function calls, `if/else`, other `switch/case`, and so on. Note that you cannot check for ranges, nor can the cases be expressions. The cases must be discrete values. The following example shows all of these. The action statements are replaced with simple comments.

```
switch( x )
{
      case 1:
            /* This code performed only if x is 1, then jump to closing
            brace */
            break;

      case 2:
            /* This code performed only if x is 2, then jump to closing
            brace */
            break;

      case 3:
            /* This code performed only if x is 3, but continue to next
            case (no break statement) */
      case 4:
      case 5:
            /* This code performed only if x is 3, 4, or 5,  */
            break;

      default:
            /* this code performed only if x is not any of 1,2,3,4, or
            5, then jump to closing brace (redundant here) */
            break;
}
```

Sometimes it is very handy to replace the numeric constants with `#define` values. For example, you might be choosing from a menu of different circuits. You would create some #define values for each at the start of the file (or in a header file) as so:

```
#define VOLTAGE_DIVIDER      1
#define EMITTER_FEEDBACK     2
#define COLLECTOR_FEEDBACK   3
/* etc... */
```

You would then write a much more legible switch/case like so:

```
switch( bias_choice )
{
      case VOLTAGE_DIVIDER:
            /* do volt div stuff */
            break;

      case EMITTER_FEEDBACK:
            /* do emit fdbk stuff */
            break;
      /* and so on... */
}
```

7.3 Looping

There are three looping constructs in C. They are `while()`, `do-while()`, and `for()`. `do-while()` is really just a `while()` with the loop continuation test at the end instead of the beginning. Therefore, you always get at least one iteration. The continuation test follows the same rules as the `if()` construct. Here are the `while()` and `do-while()` templates:

```
while( test condition(s)... )
{
      //...statements to iterate...
}

do {
      //..statements to iterate...
} while( test condition(s)... )
```

Here are some examples:

```
while( a<10 )
{
      /* Perhaps a is incremented in here.
         If a starts off at 10 or more, this loop never executes */
}

do {
      /* Perhaps a is incremented in here.
         If a starts off at 10 or more, this loop executes once */
} while( a<10 )

while( a<10 && b )
{
      /* This loop continues until a is 10 or more, or b is zero.
         Either condition will halt the loop. Variable a must be
         less than 10 and b must be non-zero for the loop to
         continue */
}
```

Usually, loops use some form of counter. The obvious way to implement a counter is with a statement like:

```
a=a+1;      /* add one to the current value of a */
```

C has increment and decrement operators, ++ and --, so you can say things like:

```
a++;  /* add one to the current value of a */
a--;  /* subtract one from the current value of a */
```

C also has a shortcut mode for most operations. Here are two examples:

```
a+=1; /* equivalent to a=a+1; or a++; */
a*=2; /* equivalent to a=a*2; */
```

You will see all three forms of increment in example and production code, although the increment and decrement operators are generally preferred.

The `for()` construct is generally preferred over the `while()` if a specific number of iterations are known. The template is:

```
for( initialization(s); termination test(s); increment(s) )
{
        ..statements to iterate..
}
```

Here is an example using the variable a as a counter that starts at 0 and proceeds to 9 by adding one each time. The loop iterates 10 times.

```
for( a=0; a<10; a++ )
{
        /* stuff to do ten times */
}
```

The following example is similar, but adds 2 at each loop, thus iterating 5 times.

```
for( a=0; a<10; a+=2 )
{
        /* stuff to do five times */
}
```

The next example uses multiples. Note the use of commas.

```
for( a=0, b=1; a<10; a++, b*=3  )
{
        /* stuff to do ten times */
}
```

In this case two variables are initialized. Also, at each loop completion, a is incremented by 1 and b is multiplied by 3. Note that b is not used in the termination section, although it could be.

If the iterated block within the braces consists of only a single statement, the braces may be left out (just like in the `if/else` construct). Loops may be nested and contain any legal C statements including assignments, conditionals, function calls and the like. They may also be terminated early through the use of the break statement. As in the `switch/case` construct, the `break` command redirects program flow to the closing brace. Here is an example:

```
for( a=0, b=2; a<7; a++ )

{

        while( b<a*10 )

                b*=2;


        if( b > 50 )

                break;

}
```

Note that the `if()` is not part of the `while()`. This is visually reinforced by the use of indents and spacing, but that's not what makes it so. The code would behave the same even if it was entered like so:

```
for( a=0, b=2; a<7; a++ ){ while( b<a*10 ) b*=2; if( b>50 ) break;}
```

Obviously, the former style is much easier to read than the later. It is **strongly** recommended that you follow the first style when you write code.

OK, what does the code fragment do? First, it sets `a` to 0 and `b` to 2. Then, the `while()` checks to see if `b` is less than 10 times `a`. 2 is not less than 0, so the `while()` doesn't iterate. Next, the `if()` checks to see if `b` is more than 50. It's not, so the `break` isn't executed. That concludes the first iteration. For the second iteration, `a` is incremented to 1 and checked to see if it's still less than 7. It is, so the loop continues and enters the `while()`. `b` is smaller than 10 times `a` (2<10), so `b` is doubled to 4. This is still smaller so it's doubled again to 4, and again to 8. Finally, it is doubled to 16. It is now larger than 10 times `a` so the `while()` loop exits. The `if()` isn't true as 16 is not larger than 50 so the `break` isn't taken. We wind up finishing iteration two by incrementing `a` to 2. The `while()` loop starts because `b` (16) is less than 10 times `a` (now 20). The loop will only execute once, leaving `b` at 32. This is still less than 50, so the `break` is ignored. The `for()` closes by incrementing `a` to 3. On the next iteration both the `while()` and `if()` are ignored as `b` is less than 10 times `a` as well less than 50. All that happens as that `a` is incremented to 4. Now that `a` is 4, the `while()` starts again (32<40). `b` is doubled to 64. That's greater than 10 times `a`, so the `while()` exits. `b` is now greater than 50 so the `if()` is taken. This results in executing the `break` statement that directs program flow to the closing brace of the `for()` loop. Execution picks up at the line following the closing brace and we are all done with the `for()` loop (no, a never gets to 7). This example is admittedly a little tricky to follow and not necessarily the best coding practice, but it does illustrate how the various parts operate.

7.4 While or For?

So, which do you use, a `while()` or a `for()`? You can make simple loops with either of them but `for()` loops are handy in that the initialization, termination, and increment are all in one spot. With `while()` loops, you only specify the termination, so you must remember to write the variable initializations before the loop as well as the increments within the loop. If you forget either of these your loop will behave erratically. It may fail to terminate altogether, resulting in an infinite loop, as shown below.

```
a=0;

while( a<10 )
{
        printf("hello\n");
}
```

This code fragment doesn't print the word *hello* ten times, it prints *hello* forever (or better to say until you forcibly terminate the program)! Although `a` was initialized and tested, it was never incremented. You need an `a++;` (or similar) within that loop.

7.5 Exercises

1. Write the code to examine the value of the variable X. If it's less than zero, the message *negative value* should be printed.

2. Write the code to examine the value of the variable X. If it's equal to zero, the message *zero value* should be printed.

3. Write the code to compare the values of the variables X and Y. The greater value should be printed.

4. Write the code to examine the values of the variables X and Y. If both X and Y are greater than zero, increment X by one.

5. Write the code necessary to print the message *Error!* six times but without using six sequential `printf()` calls.

6. Write the code required to control a loop so that it continues so long as the variable K is less than 50.

7. Write the code needed to cycle the variable R from 100 to 200 in steps of 5 (i.e., 100, 105, 110, etc.).

8. Explain the practical difference between a `while` loop and a `do-while` loop.

8. C Pointers and Addresses

8.1 Introduction

As you may recall from earlier course work, every byte of memory in a computer system is identified by a unique address. C works directly with addresses and this is one reason why it can be used to create efficient and powerful code. You can obtain the address of virtually any variable or data item using the "address of" operator, `&`. One exception to this is the `register` class variable. This is because CPU's registers don't have an address like normal memory. Also, as functions are just memory locations filled with microprocessor/microcontroller op-codes, C also makes it possible to obtain the starting address of functions.

8.2 Using Addresses and Pointers

If we declare a variable as so:

```
char a;
```

then referencing `a` will get us the value stored in `a`, as in the code `b=a;`. Using the *address of* operator, as in `&a`, will obtain the memory location of `a`, not `a`'s value or contents. This is the sort of thing we used with `scanf()` and `strcpy()`. It is possible to declare variables that are designed to hold these addresses. They are called *pointers*. To declare a pointer, you preface the variable with an asterisk like so:

```
char *pc;
```

The variable `pc` is not a `char`, it is a pointer to a `char`. That is, its contents are the address of a `char` variable. The content of *any* pointer is an address. This is a very important point. Consider the following code fragments based on the declarations above:

```
pc = a;      /* unhappy */
pc = &a;     /* just fine */
```

The first line makes no sense as we are trying to assign apples to oranges, so to speak. The second line makes perfect sense as both `pc` and `&a` are the same sort of thing, namely the address of a variable that holds a `char`. What if we want pointers to other kinds of things? No problem, just follow the examples below:

```
float *pf;        /* pointer to a float */
long int *pl;     /* pointer to a long int */
double *pd, *p2;  /* two pointers to doubles */
short int *ps, i; /* ps is a pointer to a short int */
                  /* i is just a short int */
```

As mentioned, all pointers contain addresses. Therefore, **no matter what the pointer points to, all pointers are the same size (same number of bytes)**. In most modern systems, pointers are either 32 bits (4 bytes) or 64 bits (8 bytes) although some small controllers use 16 bit addressing. When in doubt, you can check your code with `sizeof()`. If all pointers are the same size, then why do we declare different

types of pointers? There are two reasons. First, this helps with type checking. Functions that take pointers as arguments or that return pointers will be using certain forms of data. You wouldn't want to accidentally send off a pointer to a `float` when the function expects the address of a `short int` for example. Second, by specifying the type of thing the pointer points to, we can rely on the compiler to generate proper *pointer math* (more on this in a moment).

8.3 Pointer Dereferencing

Suppose you have the following code fragment:

```
char *pc, c, b;

c = 1;
pc = &c;
```

We have declared two variables, a `char` and a pointer to a `char`. We then set the contents of the `char` to 1, and set the contents of the pointer to the address of the `char`. We don't really need to know what this address is, but for the sake of argument, let's say that `c` is located at memory address 2000 while `pc` is located at memory address 3000. If we were to search the computer's memory, at address 2000 we would find the number 1. At address 3000, we would find the number 2000, that is, the address of the `char` variable. In a typical system, this value could span 32 bits or 4 bytes. In other words, the memory used by `pc` is addresses 3000, 3001, 3002, and 3003. Conversely, in a 16 bit system, `pc` would only span 3000 and 3001 (half as many bytes, but far fewer possible addresses).

As the contents of (i.e., value of) `pc` tell us where a `char` resides, we can get to that location, and thus the value of the `char` variable `c`. To do this, we *dereference* the pointer with an asterisk. We could say:

```
b = *pc;
```

Read this as "b gets the value at the address given by `pc`". That is, `pc` doesn't give us the value, rather, it tells us where to go to get the value. It's as if you went to your professor's office and asked for your grade, and instead he hands you a piece of paper that reads "I will e-mail it to you". The paper doesn't indicate your grade, but it shows you where to find it. This might sound unduly complicated at first but it turns out to have myriad uses. By the way, in this example the value of `b` will be `1` because `pc` points to `c`, which was assigned the value `1` at the start. That is, `b` gets the value at the address `pc` points to, which is simply `c`.

8.4 Pointer Math

One of the really neat things about pointers is pointer math. Returning to our example of `pc` at address 3000, if you increment `pc`, as in `pc++;` you'll get 3001. No surprise, right? If, on the other hand, you had a pointer to a `double`, `pd`, at address 3000 and you incremented it, you wouldn't wind up with 3001. In fact, you'd wind up with 3008. Why? This comes down to how large the thing is that we're pointing at. `doubles` are 8 bytes each. If you had a bunch of them, as in an array, incrementing the pointer would get you the next item in the array. This is extremely useful. Note that adding and subtracting to/from pointers makes perfect sense, but multiplying, dividing, and higher manipulations generally make no sense and are to be avoided.

8.5 Pointers and Arrays

We very often use pointers with arrays. One example is the use of strings. We noted this in earlier work. Recall that the "address of" and array index "cancel each other" so that the following are equivalent:

```
&somestring[0]     somestring
```

Let's look at an example of how you might use pointers instead of normal array indexing. We shall write our own version of `strcpy()` to copy one string to another. The function takes two arguments, the address of a source string and the address of a destination string. We will copy over a character at a time until we come to the null termination. First, the normal array index way:

```
void strcpy1( char dest[], char source[] )
{
        int i=0;                    /* index variable, init to first char */

        while( source[i] != 0 )     /* if it's not null...*/
        {
             dest[i] = source[i];   /* copy the char */
             i++;                   /* increment index */
        }
        dest[i] = 0;                /* null terminate */
}
```

Looks pretty straightforward, right? There are some minor improvements you can make such as changing the loop to `while( source[i] )`, but that's not a big deal. Now in contrast, let's write the same thing using pointers.

```
void strcpy2( char *dest, char *source )
{
        while( *dest++ = *source++ );
}
```

That's it. Here's how it works. `dest` and `source` are the starting addresses of the strings. If you say:

```
*dest = *source;
```

then what happens is that the value that `source` points to gets copied to the address referred to by `dest`. That copies one character. Now, to this we add the post increment operator `++`:

```
*dest++ = *source++;
```

This line copies the first character as above and then increments each pointer. Thus, each pointer contains the address of the next character in the array (you've got to love that pointer math, this will work with any sized datum). By placing this code within the `while()` loop, if the content (i.e., the character copied) is non-zero, the loop will continue. The loop won't stop until the terminating null has been copied. As you can imagine, the underlying machine code for `strcpy2()` will be much simpler, more compact, and faster to execute than that of `strcpy1()`. As was said at the outset of the course, you can do a lot in C with just a little code!

8.6 Exercises

1. Declare a pointer to a floating point variable, naming it `fptr`.

2. Declare a pointer to a signed character variable, naming it `cptr`.

3. Consider the following snippet of code:

```
unsigned char c, *p;
```

Explain the difference between `c` and `p`.

4. Consider the following snippet of code:

```
unsigned char *p;
double *p2;
```

Assume that the value of `p` is currently 1000 and the value of `p2` is 2000. What are their values after the following piece of code is executed?

```
p++;
p2++;
```

5. Explain the difference between the `*` and `&` operators in relation to pointers.

6. Consider the line of code below.

```
a = b*c;
```

Is the `*` operator a pointer dereference or a multiply? How do we know?

7. Consider the line of code below.

```
a = b**c;
```

What do you think this line does? How might you alter this line to mark the intent more clearly and less prone to error or misinterpretation?

8. Explain the difference between the two lines of code below.

```
a*=b;
a=*b;
```

9. C Look-up Tables

9.1 Introduction

Sometimes we use tools to make things without thinking about how the tools themselves are made. In the world of software, sometimes *how* things are done (the implementation) can have a huge impact on performance. It turns out that sometimes we can trade performance in one area for another. For example, a certain technique might be very memory efficient but rather slow to execute, or vice versa. We're going to take a look at a common programming technique that is very fast (i.e., it's "computationally efficient"). Sometimes it can require a lot of memory, sometimes not. It's called a look-up table.

9.2 "Yes, I'd Like a Table for 360, Please"

Consider the common C trig function, `sin()`. Not much to it, really. You pass it an argument and you get back the sine of the argument. But how is it implemented? It could be implemented as a Taylor Series expansion that requires several multiplies and adds. A single sine computation won't take long but what if you need to do millions of them? All of those multiplies and adds add up, so to speak. Consider the following problem: You need to get the sine of an angle specified to the nearest degree, and *fast*. Basically, you have 360 possible answers (0 degrees through 359 degrees)[7]. Now suppose you create an array of 360 values which consists of the sine of each angle in one degree increments, starting at 0 degrees and working up to 359 degrees. It might look something like this:

```
double sine_table[] = { 0.0, 0.01745, 0.034899,  0.05234,
 /* and so on to the last entry */ -0.01745 };
```

You can now "compute" the sine like so, given the argument `angle`:

```
answer = sine_table[ angle ];
```

Because of the duality of arrays and pointers, you can also write this as:

```
answer = *(sine_table + angle);
```

Without an optimizing compiler, the second form will probably generate more efficient machine code. In either case, this is very fast because it's nothing more than reading from a memory location with an offset. The result is just one add plus the access instead of a dozen multiply/adds. It does come at the cost of 360 double floats, which, at eight bytes each, is nearly 3k of memory. By recognizing that the sine function has quarter wave symmetry, you can add a little code to check for the quadrant and reduce the table to just 90 entries. Also, `floats` might have sufficient precision for the application which will cut the memory requirements in half again when compared to `doubles`.

---

[7] Outside those bounds you can always perform some minor integer math to get within the bounds (e.g., if the angle is 370, just mod by 360 to get the remainder of 10 degrees, effectively the "wrap around").

To make the above just a little spiffier, you can always make it look like a function via a `#define` as follows:

```
#define fast_sin(a)  (*(sine_table+(a)))
```

Of course, a down side to this operation is that it only deals with an integer argument and is only accurate to a single degree. You can alter the definition to allow a floating point argument and round it to the nearest whole degree as follows:

```
#define fast_sin(a)  (*(sine_table+(int)((a)+0.5)))
```

You could also turn this into a true function and add code to interpolate between adjacent whole degree entries for a more accurate result. At some point the extra refinements will slow the function to the point where a more direct computation becomes competitive.

9.3 Waving Quickly

So what's all this business about needing to do this sort of thing very fast? One application might be the direct digital synthesis of arbitrary waveforms. The idea is to create a waveform of an arbitrary shape; not just the usual sines, squares and triangles (although that's possible, too). Arbitrary waveforms can be realized using analog oscillator techniques coupled with wave shaping circuits but it can be a challenge to do well. Instead, consider creating a large table of integer values. Typically, the table size would be a nice power of two, like 256. Each entry in the table would be the digitized value of the desired waveform. A simple ramp might look like this:

```
unsigned short int ramp_table[] = { 0, 1, 2, 3, /* and so on */};
```

A more complicated wave might look like this:

```
unsigned short int squiggly_table[] = { 0, 21, 15, 33, /* etc */};
```

These values could then be sent sequentially to a digital-to-analog converter (DAC) to create the desired waveform. Once we get to the end of the table, we simply loop back to the start to make the next cycle. With a 256 entry table, we can use an `unsigned char` as the table index and once it reaches 255, incrementing it will cause it to roll over back to 0, automatically. The post increment operator is ideal for this. For the code below, assume PORT is the memory location of the DAC we are writing to.

```
unsigned char i = 0;

// loop forever
while ( 1 )
{
      PORT = ramp_table[i++];

      // now wait between each value, dependent on sample rate
      delay();
}
```

9.4 Error Correction via Table Translation

Another possible use for a look-up table is for error correction. Again, let's limit this to a nice 256 entry table. Suppose you are reading a sensor with an 8 bit (256 level) analog-to-digital converter (ADC). Maybe this is a temperature sensor and at the extremes of the temperature range it tends to go out of calibration. You can use the input value from the sensor (perhaps appropriately scaled and then turned into an integer) as the index into a 256 element table that contains the corrected values.

As an example, to keep it simple let's say the sensor reads a temperature ranging from 0 ºC to 250ºC. You calibrate it by placing it in a known 150ºC oven and the sensor reads 145º instead of the ideal 150º. You repeat this process at several other temperatures and discover that it reads 166º when it's really 170º, 188º when it's really 195º, and so on. So you create a table where the 145th entry is 150, the 166th entry is 170, the 188th entry is 195, etc. Now use the sensor value as the index into the array. The value you access is the corrected result. The table effectively translates your input into a calibrated output.

```
corrected_temp = calibration_array[ sensor_value ];
```

This is a very fast process and as accurate as your calibration measurements. As long as the sensor data is repeatable (e.g., it always reads 145ºC in a 150ºC oven), you'll get good results.

10. C Structures

10.1 Introduction

C allows compound data called structures, or `struct` for short. The idea is to use a variety of the basic data types such as `float` or `int` to describe some sort of object. Structures may contain several of each type along with pointers, arrays, and even other structures. There are many uses for such a construct and structures are very common in production C code.

As an example, we may wish to describe an electronic component such as a transistor. What sort of things do we need? There are several performance parameters that may be used such as current gain, breakdown voltage and maximum power dissipation. All of these items may be represented as `double` variables. There will be a model number. This will probably be a string as it may contain letters (such as "2N3904"). There will need to be a manufacturer's code. This could be an `int`. A real world device will have many more parameters than these five, but these will suffice for our purposes. If you only have one transistor to deal with, five separate variables is not a big deal to keep track of. On the other hand, what if you have a great number of parts as in a database? Perhaps there are 1000 devices. Creating 5000 separate variables and keeping them straight presents a bit of a challenge. It would be nice if we could combine the five items together into a "super variable". Then, all we have to worry about is creating 1000 of them for the database (perhaps with an array, although there are other techniques). There shouldn't be a problem of getting the current gain of one device confused with that of another. This is where structures come in. Below is an example of how we would define this transistor structure and associated instances.

```
struct transistor {
        double      currentgain;
        double      breakdown;
        double      maxpower;
        short int   manufacturer;
        char        model[20];
};

struct transistor my_transistor;
struct transistor *ptransistor;
```

We have defined a structure of type `transistor`. We have also declared an instance of a `struct transistor` called *my_transistor*, along with a pointer to a `struct transistor` called *ptransistor*. The five elements are referred to as the *fields* of the structure (e.g., the `currentgain` field). Note that this structure contains an array of characters for the model name/number. The model cannot exceed 19 characters (19 plus terminating null yields 20 declared). It is unlikely that we'll ever have model name/number this long, but if by chance we do, we will have to truncate it.

To set or retrieve values from an instance, we use a period to separate the structure name from the field of interest. Here are some examples:

```
my_transistor.currentgain = 200.0;
my_transistor.maxpower = 50.0;
my_transistor.manufacturer = 23;
```

In the last assignment, it may be better to use a `#define` rather than a hard number. For example, place the following definition in a header file and then use the assignment below:

```
#define MOTOROLA 23

my_transistor.manufacturer = MOTOROLA;
```

To set the model field, you could do something like this:

```
strcpy( my_transistor.model, "2N3904" );
```

Remember, `strcpy()` needs addresses. The double quote string literal produces this automatically. For the model field, we are using the shortcut described in earlier work. The line above is equivalent to:

```
strcpy( &(my_transistor.model[0]), "2N3904" );
```

If you need to use a field in a computation or comparison, the access is unchanged:

```
if( my_transistor.breakdown > 75.0 )
        printf("Breakdown voltage is at least 75 volts!\n");
```

A good question at this point is whether of not the declared order of the fields makes any difference. This depends on the compiler and target hardware. In some processors, multiple-byte variables such as long and short integers, floats and pointers must be *word aligned*. For example, a short int may be required to start on an even address or a float might be required to start on an address divisible by four. In such a system, a structure declared with the order of char, float, char, int will need *pad bytes* between some fields to ensure alignment and will take up more memory space than if the structure was organized as float, int, char, char. This is of particular importance if large arrays of structures are to be used.

10.2 Pointers and Structures

It is generally not a good practice to send entire structures to functions as arguments. The reason is because you wind up copying a lot of data. The transistor structure above contains three `doubles` at 8 bytes each, a `short int` at 2 bytes, and 20 bytes for the `char` array, leaving a total of 46 bytes of memory that need to be copied if we pass this to a function. It would be much more efficient if we simply passed the starting address of the structure to the function. That is, we tell the function where to find the structure by using a pointer (this is called "passing by reference" versus the more familiar "passing by value"). This is why we declared `ptransistor`. We initialize it like so:

```
ptransistor = &my_transistor;
```

To access the various fields, we can longer use the period because we no longer have a `struct transistor`; we have a pointer to one. For pointers, we access the fields via the pointer token, which is made up of a dash followed by a greater than sign: `->` Thus, we might say:

```
ptransistor->currentgain = 200.0;
strcpy( ptransistor->model, "2N3904" );
```

Below is a function that simply prints out the values of the various fields.

```
void print_transistor( struct transistor *pt )
{
     printf("For model: %s\n", pt->model );
     printf("Current gain is %lf\n", pt->currentgain );
     printf("Breakdown voltage is %lf\n", pt->breakdown );
     printf("Maximum power is %lf\n", pt->maxpower );
}
/* note use of %s for string and %lf for "long float" i.e., double */
```

We pass the function a pointer to a transistor structure like so:

```
print_transistor( &my_transistor );

/* we could also use print_transistor( ptransistor );
   if we initialized it as above */
```

10.3 Structures, Arrays, and So On

We have seen that it is possible to have arrays within structures. It is also possible to have structures within structures and pointers within structures. Here are some examples:

```
/* The structure definitions */
struct foo {
     float x;
     float y;
};

struct bar {
     double *pd;
     struct foo littlefoo;
     struct foo *pf;
};

/* The variable declarations */
struct foo my_foo;
struct bar my_bar;
struct bar *pbar = &my_bar;
double z=1.0;
```

The `bar` structure contains a pointer to a `double`, a pointer to `struct foo`, and a `struct foo`. We would access them as follows:

```
my_bar.pd = &z; /* pd isn't a double but the address of one, hence & */
my_bar.littlefoo.x = 2.2;
pbar->littlefoo.y = 3.3;
pbar->pf = &my_foo;
pbar->pf->x = 4.4;
```

Note that if you didn't say `pbar->pf = &my_foo;` first, then `pbar->pf->x = 4.4;` would be very evil! Without assigning `my_foo` to `pf`, this pointer would contain some random number. The second

statement would then use that number as the starting address of `struct foo`, and write the number 4.4 where the `x` field should be. As it's highly unlikely that this random number is the starting address of a `struct foo`, the number 4.4 overwrites something else. That might mean other data or even code gets destroyed. Your program behaves erratically or crashes.

Pointer Rule Number One: Never dereference[8] an uninitialized pointer!

Only bad, evil things will happen and you will become a very sad programmer.

At the beginning of the transistor example we noted that we might want to create a bunch of transistors. One possibility is to use an array. There are other ways, as we shall see. Here's how you'd declare an array of 1000 transistor structures, given the definition above:

```
struct transistor transistors[1000];
```

You would access the field as follows:

```
transistors[0].currentgain = 200.0; /* set 1st device's gain to 200 */
transistors[2].breakdown = 85.0; /* set 3rd device's breakdown to 85 */
```

Finally, it is also possible to create an array of pointers to transistor structures:

```
struct transistor *ptarray[1000];
```

Note that we do not have 1000 transistor structures, but rather 1000 pointers. Each of these would need to point to an appropriate transistor structure. Assuming you had declared one named `my_transistor` as we did earlier, you could write:

```
ptarray[0] = &my_transistor;
```

And you could access fields like so:

```
ptarray[0]->maxpower = 25.0;
```

Although this may look a little odd at first, this sort of construct does have some good uses in more advanced applications. To stretch your mind just a teensy bit further, C makes it possible to create something like an array of pointers to structures which contain a structure which in turn contains an array of pointers to still other structures. Read that again, imagine what that might look like as a memory map, and then write some possible definitions/declarations. If you can do that, you've pretty well mastered the idea.

10.4 Exercises

1. Declare a structure of type `Quest` called `Grail` that contains a float called `X`, a long integer called `Y` and an unsigned character called `Z`.

2. Given the structure of problem one, will the order of the three fields have any effect or importance? How might we determine if it does?

---

[8] i.e., try to access the fields of.

11. C Linked Lists

11.1 Introduction

A linked list is an alternate way to collect together a number of instances of a given data type. Compared to arrays, linked lists offer the advantages of not requiring contiguous memory for the collection and an easy way to re-order the collection by simply swapping pointers. As we shall see later, linked lists are also very flexible when it comes to adding or deleting items to the collection. On the downside, linked lists require somewhat more memory than arrays (since space for pointers must be included), and arrays offer consistent fast access to any member of the array (linked lists require that the list be "walked along" in order to get to a given member). To create a linked list, a pointer to the structure type is included in the definition of the structure. Once the instances of the structure type are created, they are strung together by placing appropriate addresses in the pointer field.

A graphical example is shown below with four instances of some structure:

| ptr=&B | → | ptr=&C | → | ptr=&D | → | ptr=0 |

A   B   C   D
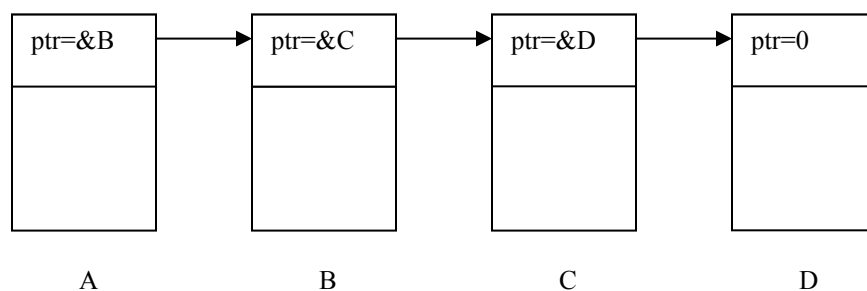
Figure 11.1, linked list

Imagine that each structure is 100 bytes in size. Further, assume that A is located at memory address 1000 (spanning 1000 to 1099), B is a located at address 2000, C at 3000, and D at 4000. Each structure contains a pointer to the next one in the list. A contains a pointer to the address of B, B to C, and so forth. For example, the value of A's pointer is 2000 (the memory address of B), while the value of C's pointer is 4000 (the address of D).

Note that &A is the top of the list, and that the last item signifies that no items remain by having its pointer set to 0. Further, these four items could be linked in any manner by simply changing the pointer values. Finally, items A, B, C, and D may reside anywhere in the memory map; they need not be contiguous. In contrast, if we were to make an array out of these structures, they would need to be packed into memory in sequence in order to be indexed properly. That is, if A were located at address 1000, B would have to be at 1100, C at 1200, and D at 1300, as they are each 100 bytes in size[9]. If you wanted to rearrange the structures (e.g., to sort them), add new ones or delete existing ones, this would require quite a bit of work

---

[9] Array indexing works by simply multiplying the index value by the size of the arrayed item, and then using this value as an offset from the starting address. Thus, in this example, item[2] (i.e., C) is found by multiplying the index of 2 by the size of 100 bytes to achieve a 200 offset from the starting location of address 1000, or 1200. Remember, item[0] is A, while item[1] is B, and item[2] is C.

with an array. These tasks are much easier with a linked list because the structures themselves aren't manipulated, only the associated pointers.

11.2 Linked List Example

Now for a concrete example. A typical structure definition may look something like this:

```
struct Marmot (
      struct Marmot *NextMarmot;
      float Age;
      float Weight;
};
```

We could declare three instances of Marmots and link them together as follows:

```
struct Marmot Larry = { 0, 3.4,  19.7 };
struct Marmot Jane = { &Larry, 2.5, 13.1 };
struct Marmot Felix = { &Jane, 2.9, 15.6 };
```

`Felix` is at the top of the list, while `Larry` is at the bottom. Note that the items must be declared in inverse order since the pointer reference must exist prior to assignment (i.e., `Jane` must exist in order for `Felix` to use `Jane`'s address). It is common to also declare a pointer to use as the head of the list. Thus, we might also add:

```
struct Marmot *MarmotList = &Felix;
```

Thus, the following are true:

```
Felix.NextMarmot  points to Jane
MarmotList->NextMarmot  points to Jane
Jane.NextMarmot  points to  Larry
MarmotList->NextMarmot->NextMarmot  points to  Larry
Larry.NextMarmot is 0
MarmotList->NextMarmot->NextMarmot->NextMarmot  is  0
```

The final line of pointers to pointers is not very practical. To get around this, we can use a temporary pointer. Below is an example function that takes the head of a `Marmot` list as its argument, and then prints out the ages of all of the `Marmot`s in the list.

```
void PrintMarmotAges( struct Marmot *top )
{
      struct Marmot *temp;

      temp = top; /* initialize pointer to top of list */

      while( temp ) /* true only if marmot exists */
      {
            printf( "%f\n", temp->Age );
            temp = temp->NextMarmot );
      }
}
```

It would be called like so:

```
PrintMarmotAges( MarmotList );
```

Note that we could've reused `top` rather than use the local `temp` in this case. If the head of the list will be needed for something else in the function though, then the local variable will be required (i.e., since `temp = temp->NextMarmot` effectively erases the prior value of `temp`, we lose the head of the list as we walk down the list).

11.3 Exercise

A bipolar transistor can be described (partially) with the following information: A part number (such as "2N3904"), a typical beta, and maximum ratings for P_d, I_c, and BV_{ceo}. Using the data below, create a program that would allow the user to search for devices that meet a minimum specified requirement for beta, P_d, I_c, or BV_{ceo}. Devices that meet the performance spec would be printed out in a table (all data fields shown). If no devices meet the spec, an appropriate message should be printed instead. For example, a user could search for devices that have a P_d of at least 25 watts. All devices with $P_d >= 25.0$ would be printed out.

Device	Beta	Pd(W)	Ic(A)	BVceo(V)
2N3904	150	.35	.2	40
2N2202	120	.5	.3	35
2N3055	60	120	10	90
2N1013	95	50	4	110
MPE106	140	15	1.5	35
MC1301	80	10	.9	200
ECG1201	130	1.3	1.1	55

12. C Memory

12.1 Introduction

Up until now, whenever we have needed variables we simply declared them, either globally or locally. There are times, however, when this approach is not practical. Consider a program that must deal with a large amount of data from external files such as a word processor, or graphics or sound editor. In all instances the application may need to open very large files, sometimes many megabytes in size. The issue is not merely the size. After all, you could declare a very large array, or perhaps several of them. The problem is that the data is both large and variable in size. For example, you might edit a sound file that's 100k bytes in size, but you might also need to edit one that's 100 times larger. It would not be wise to declare a 10 megabyte array when you only need 100k. Further, you can guarantee that if you do declare 10 megabytes, the day will come when you'll need 11 megabytes. What is needed is some way of dynamically allocating memory of the size needed, when needed.

12.2 Free Memory Pool

In a given computer, memory is used by the operating system as well as by any running applications. Any memory left over is considered to be part of the "free memory pool". This pool is not necessarily contiguous. It may be broken up into several different sized chunks. It all depends on the applications being run and how the operating system deals with them. The total amount of free memory and the locations of the various chunks will change over time. C offers ways of "asking" the operating system for a block of memory from the free pool. If the operating system can grant your request, you will have access to the memory and can use it as you see fit. When you are through using the memory, you tell the operating system that you are done with it so that it can reuse it elsewhere. Sounds simple, right? Well, it is!

12.3 Allocating Memory

To use the memory routines, include the `stdlib.h` header in your code and be sure to link with the standard library. There are two main memory allocation functions. They are `malloc()` and `calloc()`. Here are their prototypes:

```
void * malloc( unsigned int size );
void * calloc( unsigned int num_item, unsigned int item_size );
```

`malloc()` takes a single argument: The number of bytes that you wish to allocate from the free pool. `calloc()` takes two arguments: The number of items that you want to fit into the memory block and their size in bytes. Basically, `calloc()` just calls `malloc()` after multiplying the two arguments together. It is used for convenience. Both functions return a pointer to a type `void`. What is this? A `void` pointer can be thought of as a generic, one-size-fits-all pointer. It prevents possible type size clashes. You can assign a `void` pointer to another type of pointer and not get a type mismatch. If the memory request cannot be made (not enough memory) then the functions will return `NULL`. **Always check for the `NULL` return! Never assume that the allocation will work!**

If you want to obtain space for 100 bytes, you'd do something like this:

```
char *cp;

cp = malloc( 100 );
if( cp )
{
      /* memory allocated, do stuff... */
}
else
{
      /* not allocated, warn user and fail gracefully... */
}
```

If you need space for 200 `double`s, you'd do something like this:

```
double *dp;

if( dp = calloc( 200, sizeof(double) ) )  /* assign and if test in 1 */
{
      /* memory allocated, do stuff... */
}
else
{
      /* not allocated, warn user and fail gracefully... */
}
```

Note the use of the `sizeof()` operator above. If you had a structure and needed to create one (for example, to add to a linked list), you might do this:

```
struct foo *fp;

if( fp = calloc( 1, sizeof(struct foo) ) )
{
      /* remainder as above ... */
```

12.4 Using Memory

The pointer that is returned from the allocation function is used as the base of the object or array of objects in which you're interested. Keeping it simple, suppose you want to allocate an array of three integers. If you want to set the first element to 0, and the second and third elements to 1, do the following (code fragment only, error processing not shown):

```
int *ip;

if( ip = calloc( 3, sizeof(int) ) )
{
      *ip = 0;
      *(ip+1) = 1;       /* could also say ip[1] = 1; */
      *(ip+2) = 1;       /* could also say ip[2] = 1; */
}
```

Note the freedom that we have with the pointer. It can be used as a normal pointer or thought of as the base of an array and indexed accordingly. Similarly, we might need to allocate a structure and initialize its fields. Here is a function that we can call to allocate a `struct foobar,` initialize some fields, and return a pointer to it.

```
struct foobar {
      double d;
      int i;
      char name[20];
};

/* other code... */

struct foobar * alloc_foobar( void )
{
      struct foobar *fp;

      if( fp = malloc( sizeof(struct foobar) ) )
      {
            fp->d = 12.0;      /* just some stuff to show how... */
            fp->i = 17;
            strcpy( fp->name, "Timmy" );
      }
      return( fp );
}
```

12.5 Freeing Memory

Once you're done using memory, you must return it to the free memory pool. If you don't, no other application (nor the operating system) can use it. The memory will be effectively lost until the system is rebooted. This is known as a memory leak. To return memory that you have no further use for, use `free()`. Here is the prototype:

```
int free( void *p );
```

p is the pointer that you initially received from either `malloc()` or `calloc()`. The return value of the `free()` function is 0 for success or −1 on error. Normally this function never fails if it is given a valid pointer. If it does fail, there is little that you can do about it (at least not at this level). **Remember: Every block that you allocate eventually must be freed!** You might wonder why the `free()` function does not need to know the size of the block to free. This is because along with the memory they pass to you, `malloc()` and `calloc()` actually allocate a little bit more for use by the operating system. In this extra memory that you don't see are stored items such as the size of the block. This saves you a little house keeping work.

12.6 Operating System Specific Routines

Often the standard routines examined above are augmented with special routines unique to a given operating system. These might give you control over using virtual memory, presetting memory values, or allow you to obtain access to special kinds of memory (e.g., graphics memory).

12.7 Exercises

1. Write the code to allocate 1000 bytes of memory.

2. Write the code to allocate space for an array of 500 single precision floating point values.

3. Write the code to free the memory allocated in problems one and two.

13. C File IO

13.1 Introduction

High level fileio in C uses functions such as `fopen()`, `fclose()`, `fread()`, `fwrite`, `fprintf()`, `fgetc()`, and so on. These utilize a variable of type `FILE` to access disk files. They allow you to read and write data from/to files on disk in a manner very similar to sending data to the computer screen via `printf()` or retrieving data from the keyboard via `scanf()`.

Closer to home, we have low level fileio. These use a file descriptor, which is basically just an integer. High level functions actually call the low level functions anyway. There are five things you need to do with files. First, you open them to obtain access to an existing file or to create a new one. Next, you read or write data from/to the file. You may also need to move around in a file, perhaps to reread data or to skip over data. Finally, when you are finished using the file, you must close it. To open a file use:

```
fh = open( name, mode );
```

where

```
char *name: /* disk name of file */
int fh:     /* file descriptor */
int mode;   /* a define */
```

`fh` is the file descriptor that is needed for subsequent read/write calls. It will be `>= 0` if all is OK, `-1` on error.

Example modes:

```
O_RDONLY  read only
O_WRONLY  write only
O_CREAT   create if not exists
```

To read/write data, use:

```
count = read( fh, buffer, len );
count = write( fh, buffer, len );
```

`fh` is the file descriptor returned from `open()`, `buffer` is the address of where to find/place data (i.e., the thing you're copying to disk or making a copy of from disk), `len` is the number of bytes to read/write, `count` is the actual number of bytes read/written.

A common construct is:

```
if( (count = read( fh, buf, len )) != len )
{
        //...there was an error, process it...
}
```

You can also skip around in a file:

```
apos = lseek( fh, rpos, mode );
```

where

```
long apos:  absolute position (-1 on error)
long rpos:  relative position
mode:       0 for relative to beginning of file (rpos >= 0)
            1 for relative to current position
            2 for relative to the end (rpos <= 0)
```

Note that your present position is = lseek(fh, 0, 1);

When you are done with the file, you must close it:

```
error = close( fh );
```

error is 0 if all went OK, -1 otherwise.

C allows you to have multiple files open simultaneously for reading and writing. You will need one file descriptor for each file open at the same time (they can be reused later, if desired).

Below is a program that can be used to investigate the contents of almost any kind of file. It uses a variety of techniques that we have examined through the course. Some lines have wrapped.

```c
/*                      headdump.c

   This program spits out the first 128 bytes of a file in hex, decimal, and
   string form (non printable chars are printed out as periods). */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define CHUNKSIZE 128

unsigned char buf[CHUNKSIZE];

char *szerrmsgs[] = {
   "No errors on %s\n",
   "USAGE: %s <filename>\n",
   "Could not open %s\n",
   "Seek error on %s\n",
   "Position error on %s\n",
   "Rewind error on %s\n",
   "Read error on %s\n"
   };

void my_exit( FILE *fp, int err, char *pc )
{
   if( fp )    fclose( fp );
   if( err )   printf( szerrmsgs[err], pc ); /* don't bother if all OK */
   exit( err );
}
```

```
void main( int argc, char *argv[] )
{
   int    size, c=0, x;
   FILE   *fp=0;

   if( (argc < 2) || ( !strcmp(argv[1], "?") ) )
      my_exit( fp, 1, argv[0] );

   if( fp = fopen( argv[1], "r" ) )
   {
      /* Find out how big the file is. If it's < CHUNKSIZE then read in
         what's available */
      if( -1 != fseek( fp, 0, 2 ) )     /* seek to end */
      {
         if( -1 != (size = ftell( fp )) )
         {
            if( size > CHUNKSIZE )  size = CHUNKSIZE;
            if( -1 != fseek( fp, 0, 0 ) )     /* seek to start */
            {
               if( fread( buf, 1, size, fp ) == (unsigned int)size )
               {
                  /* print this out as 8 chars by 16 (or so) lines, first hex,
                     then decimal, then string */
                  while( c < size )
                  {
                     /* print out line as hex */
                     printf("%3d: %02x%02x %02x%02x %02x%02x %02x%02x ",
c,buf[c],buf[c+1],buf[c+2],buf[c+3],buf[c+4],buf[c+5],buf[c+6],buf[c+7] );

                     /* print out line as decimal */
                     printf("   %03d%03d %03d%03d %03d%03d %03d%03d ",
buf[c],buf[c+1],buf[c+2],buf[c+3],buf[c+4],buf[c+5],buf[c+6],buf[c+7] );

                     /* print out line as string. check the chars and if not
                        printable, replace with periods */
                     for( x=0; x<8; x++ )
                     {
                        if( !isprint( buf[c+x] ) )
                           buf[c+x] = '.';
                     }
                     printf("   %c%c%c%c%c%c%c%c\n",buf[c],buf[c+1],buf[c+2],
buf[c+3],buf[c+4],buf[c+5],buf[c+6], buf[c+7] );

                     c+=8;
                  }
               }
               else
                  my_exit( fp, 6, argv[0] );
            }
            else
               my_exit( fp, 5, argv[0] );
         }
         else
            my_exit( fp, 4, argv[0] );
      }
      else
         my_exit( fp, 3, argv[0] );
   }
   else
      my_exit( fp, 2, argv[1] );

   my_exit( fp, 0, argv[1] );
}
```

14. C Command Line Args and More

14.1 Command Line Arguments

Question: How do various utilities "read" the arguments that you place after them on the command line? For example, you might have a utility that archives (compresses) a file, creating a new (compressed) file. You might use it like this from a DOS or shell prompt:

```
C:>archive foo.txt foo.arc
```

The program is called `archive` `(archive.exe)`, and you're telling it to compress the file `foo.txt` and create a new file called `foo.arc`. This is much faster than using `scanf()` type input from within the program (i.e., having the user run the program, at which point the program then prompts for the two file names). C allows a very simple method of obtaining these command line arguments. This requires a modification to the declaration of `main()`:

```
void main( int argc, char *argv[] )
```

The first parameter is called the argument count and tells you how many items (strings) where entered on the command line. In the example above, `argc` would be 3. The second parameter is called the argument vector. It is an array of pointers to strings. These strings are the items entered on the command line. Thus `argv[0]` points to "archive", `argv[l]` points to "foo.txt", and `argv[2]` points to "foo.arc".

If `argc` is 1, then no arguments were added after the executable name. Below is a simple echo example. This will echo whatever was typed in on the command line.

```
void main( int argc, char *argv[] )
{
        int x;

        for( x=0; x<argc; x++ )
               printf( "Argument %d is %s\n", x, argv[x] );
}
```

Note that since `argv` is an array of pointers, then `argv[x]` is also a pointer, and is treated just like any other pointer to characters. Thus, you can use functions like `strcpy()` or `strcmp()` on it. For numeric arguments (such as "archive blah 23"), you may convert the ASCII string ("23") into either an integer, long integer, or float via the `atoi()`, `atol()`, and `atof()` functions, respectively.

Other possibilities include printing out directions if `argc` is 1 or if the only added argument is "?". It is possible to take this further by adding "switches" or "flags" to the argument list so that arguments can be presented in any order. This is precisely what is done if the compiler or linker is run directly rather than via the IDE.

In summation, command line arguments are a very handy and quick way of getting values into a program. As an exercise, alter one of your previous programs to utilize command line arguments rather than the `scanf()` approach.

14.2 Conditional Compilation

Suppose for a moment that you wish to create two or more versions of a program that differ only in mild ways. While it is possible to maintain multiple sets of source code for each, this can be a pain when it comes time to fix a bug or add a feature since alterations will need to be made to each of the source files. It would be much easier to somehow "flag" parts of the source code as belonging to different versions of the program and then have the compiler or preprocessor do the work for you. This is exactly what conditional compilation is. To do this, we exploit the power of the preprocessor. We have already looked at the `#define` directive as a way of defining constants or creating macros, but we can take it further. Look at the following example fragment:

```
#define VERSION_A

#ifdef VERSION_A
char title[] = "This is version A";
#else
char title[] = "This is some other version";
#endif
```

The `#if/#else/#endif` directives act similarly to the `if/else` commands. Note that parentheses are not used to block multi-line sections (hence the need for the `#endif` directive). In the example above, the `char` array title is initialized to `"This is version A"`. If we commented out the `#define VERSION A` line, then `title` would be initialized to `"This is some other version"`. In some IDEs it is possible to create variations on projects. Within each project you can define certain symbols (such as `VERSION_A`). This way you wouldn't even have to comment/uncomment the `#define`, but just select the desired project variation. Note that it is possible to nest `#if/else` directives. Further, you are not limited to simply altering global declarations. This technique can be used on code as well as data. In fact, entire functions may be added or excluded in this way. Here is another typical use:

```
void some func( void )
{
        //...code stuff...

#ifdef DEBUG
    printf("Error in some_func, x=%d\n", x );
#endif

//...rest of function...
}
```

If `DEBUG` is defined, the `printf()` call will be included in the executable. If it is not defined, it is as if the `printf()` call never existed.

14.3 Exercise

Add `DEBUG printf()` statements to any of your existing programs. Compile with and without the `DEBUG` directive.

15. Embedded Programming

15.1 Introduction

As mentioned earlier, it is possible to break down computer programming into two broad camps: desktop applications and embedded applications. The embedded application market is ubiquitous but somewhat hidden to the average user. A typical person doesn't even realize that they're running an embedded program while they're using their cell phone, DVD player or microwave oven. Certainly, the trappings of a "normal" computer generally do not exist in these instances; there's usually no monitor or keyboard to speak of. From a programmer's perspective, what's different about the two and how is program development and testing affected?

15.2 Input/Output

Consider a typical embedded application such as a programmable or "intelligent" thermostat. Unlike a normal electro-mechanical thermostat, these devices allow the home owner to automatically change temperature at different times of the day in order to save energy. After all, why have the heat or air conditioner running when no one's home? Certainly, these devices do not come with a monitor or keyboard. In their place may be a small LCD display with a few fixed messages and a two digit numeric display for the temperature. For input there may be as few as two or three buttons for programming (set item plus up and down). By comparison, a microwave oven will probably have a complete numeric keypad with a series of special function buttons along with multiple seven-segment displays, or possibly several alpha-numeric displays for short messages. In any case, these devices are far different from the standard desktop computer. Consequently, a programmer's approach to input and output processing will be much different in the embedded case.

To start with, it is unlikely that there will be `printf()` and `scanf()` style functions. They are largely worthless in this world. What use would `printf()` be if all you have for output hardware is a bunch of LEDs? For input, you often need to read the state of switches, pushbuttons, and possibly some form of level control such as a rotary knob. For output, you often need to simply light an LED or set a value on a seven-segment display. For "fixed" style messages, these also need only a single signal to turn them on or off, such as an LED. In more advanced applications, a multi-line alphanumeric display may be available so setting individual letters is a possibility. In almost all cases these chores are handled by setting or clearing bits on specific output or inputs ports on the microcontroller. Some ports may be set up as a byte or word. Further, some ports may be bi-directional, meaning that they can behave as either input or output depending on some other register setting. Ports are little more than pins on the microcontroller that are hooked up to external circuitry. Thus, if a port is connected to an LED circuit, setting the output of the port HIGH could light the LED while setting the port LOW could turn off the LED. The obvious question then is "How do you read from or write to a port?" In many cases ports will be *memory mapped*. That is, a specific address in the memory map is allocated to a given port. You read and write from/to it just like any other variable. Further, development systems sometimes disguise these addresses as pre-defined global variables. They might also include a library of specific routines to utilize the ports. Thus setting a certain port (let's call it the "A" port) to a high might be as simple as `PORT_A = 1;` or `set_portA(1);`. Reading from a port might be something like `a = PORT_A;` or `a = get_portA();`. Consequently, embedded code is often all about reading and writing to/from ports and then branching to the requested chores.

There are some tricks to this. For example, how do you know if a key has been pressed? Calling `get_portA()` tells you the state of the switch connected to port A at the instant you call it. There is no "history" here if this is a simple momentary pushbutton instead of a toggle switch. In a case like this you might "poll" the port waiting for something to happen. This involves using a simple loop and repeatedly reading the port state in the loop. The code breaks out when the state changes:

```
while( get_portA() );
```

This will keep looping until `get_portA()` returns 0. Of course if you need to monitor several ports as is typical, you'll need to read a value from each for the test. This form of monitoring while waiting for something to happen is called an *event loop*. It may not be evident, but your house and car are probably filled with devices running event loops, just waiting for you to do something! These loops execute fairly fast so a time lag between your push and the resulting action is not noticed. On the output end, a port normally stays at the value you set it, so there is no need for a loop to "keep it set".

For more complicated displays such as a seven segment or alpha-numeric device, you may need to create a table of values indicating bit patterns for each numeral or letter to be displayed. These patterns, or words, would then be sent out various ports that are in turn connected to the displays.

For variable input devices such as a volume control, the external device (knob, slider, sensor, etc.) will be connected to an *analog to digital converter* or ADC. This might be a separate circuit or the controller may have these built-in. The converter turns the analog voltage into a numeric value that you can read from a port. For example, a volume control may be just a potentiometer connected to a fixed voltage. As the knob is moved, the voltage at the wiper arm will change. The A/D converter may encode this into a single byte. A byte ranges from 0 to 255 in value. Thus, if the volume is at maximum, the port will read 255. If the volume is at halfway the port will read about 127, and finally 0 when the volume is off.

15.3 Math

Usually, embedded code is not math intensive. There are some exceptions to this rule, but generally code for a microwave oven doesn't need something like a cosine function. Many embedded systems do not have or need floating point math. All math operations are performed using integers. Look-up tables may be used to speed processing in some cases. You will sometimes hear of "fixed point" math versus floating point. This is a fairly simple idea. Suppose you need to work with variables to the precision of tenths but you only have integers. Simply treat your variables as having an unseen decimal point after the first digit and think of all values as being in tenths. Thus, the number 17.3 would be stored and manipulated as 173. If the result of some calculation is say, 2546, then you know the real answer is 254.6.

15.4 Memory and Hardware

Most embedded applications just run one piece of code. Therefore, you can think of a program as "owning" everything. There's no sharing of resources. This makes life easy in many regards. For example, there's not much need for an operating system. Also, the system is "known" in that your code will be running on fixed hardware. Execution times are very predictable. Of course, the computational power of the processors tends to be much less than in the desktop world. Still, you can do things that are not practical in the desktop world due to hardware variation. A classic example is a "timing loop". Sometimes you need to create a time delay or to "waste" a certain amount of time, perhaps for

synchronization to some external hardware. You can do this with a simple loop that iterates a specific number of times:

```
for( c=0; c<1000; c++ );
```

This loop does nothing but count, but each count will require a certain number of clock cycles from the microcontroller, and thus a specific time. These are usually determined experimentally. You could sit down with the processor manuals and figure out how long a loop will take, but it's usually easier to just write the thing and try a few values. The result will depend on the specific microcontroller used as well as its clock frequency.

15.5 Code Development

The real "kicker" is that you can't do *native development* with embedded code. In other words, you can't program the microcontroller just using the microcontroller the way you can create desktop applications using a desktop computer. Instead, you need to have a *host* and a *target*. The host is the computer you use for development (such as a normal desktop unit) while the target is the thing you're developing for (the embedded application). The compiler that you use is technically referred to as a *cross-compiler* because it creates machine code for a processor other than the one the host uses. For example, your PC might use a Pentium processor, but the cross-compiler that runs on it creates machine code for a specific Atmel AVR microcontroller. To test your code, you need to either simulate the target on the host, or you can download the compiled code to the target and test it there. This is an extra, but unavoidable, step.

16. Hardware Architecture

16.1 Introduction

Arguably, the first commercial microprocessor was the Intel 4004 released in 1971. It was a four bit processor and was comprised of a series of ICs. A year later, Intel released the 8008, an eight bit processor on a single chip. The remainder of the decade saw the development of numerous microprocessor and microcontroller lines from various manufacturers including the 8088, Z80, 6502, 6809 and 68000 to name a few. These served as the core of the first generation of home computers. Their performance, while impressive for the day, pales in comparison with modern processors. Many used clock frequencies in the 1 to 5 MHz region and could access only modest amounts of memory (typically less than one megabyte). Integration of electronic control into everyday consumer items such as cars, microwave ovens and TVs, gained momentum during this time yielding the *microcontroller*. A microcontroller can be thought of as a specialized microprocessor. Typically, they do not have the computational power of microprocessors but instead feature other functional blocks that are useful when interfacing to sensors, actuators, display devices and the like that are found in consumer or industrial electronics.

While the design of any particular microcontroller will be unique, there are certain themes and elements that are consistent. Unlike a microprocessor used in a personal computer, a microcontroller is generally an all-in-one solution designed to reduce cost and save space. For example, the code and data space requirements for an embedded application tend to be quite small compared to traditional desktop applications. Consequently, all memory is included on the microcontroller (not to be confused with the *cache memory* found on microprocessors which exists primarily to increase the computational speed of the unit). Further, the microcontroller will include functional blocks such as independent programmable timers, general purpose digital input/output (GPIO) ports and analog input ports featuring ADCs with eight or more bits of resolution. Some controllers also contain digital to analog converters for direct analog signal generation or other specialized blocks for dealing with serial communications, wireless communications and so forth. As a microcontroller can have a large number of these interface blocks, it is not always cost effective to dedicate external IC pins for each and every input and output function. Consequently, pins are usually multiplexed, each offering multiple possible uses. For example, a specific pin might be programmed for digital input, digital output or a pulse-width modulated output (obviously, not simultaneously).

As mentioned in Chapter Two, processors might use a shared memory scheme for code and data (Von Neumann architecture) or a split memory scheme (Harvard architecture). Harvard architecture is somewhat less flexible but it allows for faster processing as the processor can fetch new instructions very quickly and without any collision with data access. In the case of microcontrollers, it is worth noting that the memory holding the code must be *non-volatile* otherwise the control program will cease to exist once power is cycled.

The other major item of importance in microprocessor/microcontroller architecture is the type of instruction set used. The two basic choices are CISC (Complex Instruction Set Computing) and RISC (Reduced Instruction Set Computing). A CISC processor contains instructions that might offer several low-level steps rolled into one. While this sounds very convenient, the down side is that these instructions often take several clock cycles to execute. By comparison, RISC architectures do not use these complex instructions, instead focusing on getting each instruction to execute in a single cycle. These are usually *pipelined* as well, meaning that while one instruction is executed, the next instruction is being loaded from program memory. RISC processors are used in everything from simple embedded applications to cell

phones to supercomputers. Two examples of embedded RISC processors are the Atmel AVR and the ARM. These can be found in applications such as tablet devices, smart phones, game consoles, game controllers and automotive controllers. The AVR is the core of the ATmega processors used in most Arduino controller boards. We shall examine the Arduino system in the following chapters.

16.2 Atmel AVR Core

The Atmel AVR processor core uses Harvard architecture with simple pipelining. A block diagram of the AVR series core is shown below.
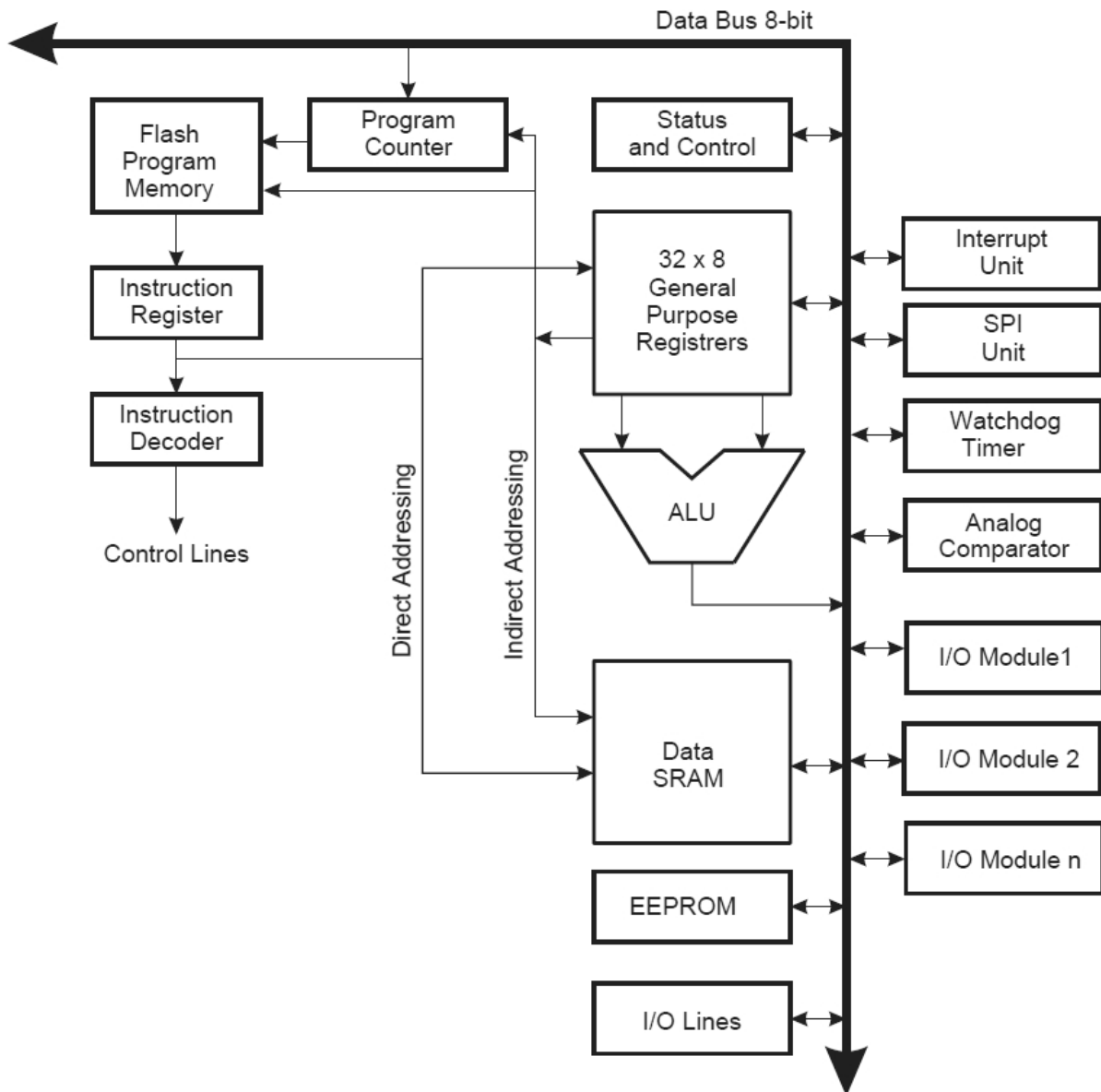

Figure 16.1, AVR block diagram (Atmel 2014)

Embedded Controllers

The first thing to notice is that most of the blocks are interconnected via an eight bit *data bus* (i.e., a collection of parallel connections, in this case, eight of them). The data bus is bidirectional meaning that data can be placed onto it (written) by one block and pulled off (read) by another. Obviously, some form of "traffic control" needs to be applied and that is usually handled by appropriate timing signals and tri-state buffers[10].

The ALU, or Arithmetic Logic Unit, is responsible for basic computational functions such as integer addition and subtraction, bit operations, comparisons and the like. In association with this are 32 eight bit general purpose registers. The ALU performs operations on the values in the registers, not directly on values in general memory. For example, suppose you want to add two variables and place the result in a third variable. The ALU has to transfer values from memory to the registers where the computation is performed and then transfer the result back to the final location. A large number of registers is a common feature of RISC processors. Early CISC processors had very few registers (indeed, many had a single "accumulator" for these sorts of operations). There are also many specialized registers not shown here as they are associated with IO modules, timers and so forth (more on this in a moment). Three important specialized registers are the *status register*, *program counter* and *stack pointer*. The program counter (PC) keeps track of the currently executing instruction (i.e., the current location of the code flow). The stack pointer (SP) records the current memory address of the top of the stack (i.e., the location in memory used for temporary variables and such). The status register (SR) contains a series of bits reflecting the most recent results of an ALU operation and the general state of the processor. The AVR status register contains eight bits, all of which are read/write and initialized at 0:

Bit	7	6	5	4	3	2	1	0
Function	I	T	H	S	V	N	Z	C

Figure 16.2, AVR status register (Atmel 2014)

The bits are as follows from the Atmel documentation:

Bit 7 – I: Global Interrupt Enable
The Global Interrupt Enable bit must be set for the interrupts to be enabled. The individual interrupt enable control is then performed in separate control registers. If the Global Interrupt Enable Register is cleared, none of the interrupts are enabled independent of the individual interrupt enable settings. The I-bit is cleared by hardware after an interrupt has occurred, and is set by the RETI instruction to enable subsequent interrupts. The I-bit can also be set and cleared by the application with the SEI and CLI instructions, as described in the instruction set reference.

Bit 6 – T: Bit Copy Storage
The Bit Copy instructions BLD (Bit LoaD) and BST (Bit STore) use the T-bit as source or destination for the operated bit. A bit from a register in the Register File can be copied into T by the BST instruction, and a bit in T can be copied into a bit in a register in the Register File by the BLD instruction.

Bit 5 – H: Half Carry Flag
The Half Carry Flag H indicates a Half Carry in some arithmetic operations. Half Carry Is useful in BCD arithmetic.

Bit 4 – S: Sign Bit, S = N $\oplus$ V
The S-bit is always an exclusive or between the Negative Flag N and the Two's Complement Overflow Flag V.

---

[10] A tri-state buffer simply repeats its input signal at its output like an ordinary buffer but also offers a third "high Z" state. When the high Z state is enabled, no output signal is present and the output impedance of the buffer goes to a very high value, effectively disconnecting the buffer from the bus.

Figure 16.3, AVR status register bits (Atmel 2014)

It is important to remember that the state of the status register is not saved when entering an interrupt routine or restored when returning from an interrupt. This must be handled in software. We shall see how this is handled in future chapters.

16.3 Memory

Another important aspect is memory. Note that the AVR contains several different kinds of memory including flash program memory, static data memory or SRAM (Static Random Access Memory) and EEPROM (Electrically Erasable Programmable Read-Only Memory). Typical embedded applications run a single program repeatedly. This program must run from the moment the device is turned on until it is turned off. In many applications, this program is never updated (a microwave oven, perhaps). In other applications, it may be updated but only by qualified technicians, not by the consumer (for example, an automotive engine management system). Consequently, the program memory needs to be non-volatile, that is, it must be able to survive without power. Historically, this was done via ROM (Read-Only Memory) or PROM (Programmable Read-Only Memory). In both of these, individual bits can be set or cleared through the use of fuse/anti-fuse links. The difference is that ROMs are programmed at the time of manufacture while PROMs are programmed after manufacture. Both are permanent and cannot be reprogrammed. ROM is less expensive for large production runs but more expensive in small quantities. EEPROM has the advantage of being erasable and thus is a form of non-volatile read/write storage. Flash RAM is similar but is much less expensive. It has the downside that it must be programmed in blocks whereas EEPROM is byte-programmable. Therefore, Flash RAM (also called NVRAM) is ideal as a storage medium for the main program and EEPROM is useful as a medium for storing data that must survive a power cycle. A possible example use of EEPROM would involve saving user preference settings for a digital camera. When power is turned off and back on, the user expects the device to be "as they left it", not reverting back to a default state. Typically, special instructions or procedures are needed to write to or read from EEPROM.

In contrast to Flash RAM and EEPROM, SRAM is volatile, just like the common DRAM (Dynamic RAM) variants found in personal computers. Static RAM typically is made of six transistors arranged as a flip-flop for each bit of storage. DRAM, by comparison, typically consists of a single transistor-capacitor combination per bit. SRAM is less dense and more expensive per bit than DRAM but it is very fast and does not need to refreshed (the charge in a DRAM cell will leak over time requiring the cell to be refreshed at regular intervals). As a result, SRAM is used for general purpose registers and special purpose/IO registers. The ATmega 328P used in the Arduino Uno development board features 32k bytes

of Flash RAM, 1k of EEPROM and 2k of SRAM. The Uno is detailed in the following chapter. By comparison, the ATmega 48A consists of 4k Flash, 256 bytes of EEPROM and 512 bytes of SRAM. Clearly, these are very modest memory footprints when compared to personal computers yet they are sufficient to power a wide range of applications. The 328P is by no means at the top of the heap, so to speak. Other controllers boast megabytes of addressable memory. There is, however, no need to over-specify a controller for a given embedded application. Extra memory that goes unused will not make the application faster; it will merely lead to more expensive hardware.

Continuing on, all of the registers in the AVR core are memory mapped, that is, they are tied to specific addresses in the SRAM. For example, the high (ADCH) and low (ADCL) bytes of the 328P's analog to digital converter output are found at addresses 0x79 and 0x78. Similarly, the output of port B (PORTB) is found at address 0x25 while the data direction register of port B (DDRB) is found at 0x24. A detailed table of all registers is found in the Atmel 2014 documentation under "Register Summary". A portion of these are available in the Appendix.

Finally, a series of other blocks such as the interrupt unit and the IO modules complete the design. We shall take a much closer look at these blocks in the upcoming chapters.

17. AVR ATmega 328P Overview

Or "Controlablanca: A Film Noir Microcontroller"[11]

"I need some assistance." Rick peered over the top of his work bench. It wasn't often that a dame walked into the lab, and a real looker at that. Beautiful, intelligent women had a way of turning his life into a burning hell and this one was a potential forest fire in heels. "Damn!" he cussed as he shook his finger, now blistering from its unfortunate collision with the hot soldering iron. Rick sidled up to her, somewhat wary. She was a knock-out, that's for sure; the kind of woman who could make Ingrid Bergman look bad on her best day and from her demeanor Rick assumed she was a grad from the polytechnical institute. Confident, tall and slender, she had the longest legs Rick had ever seen. "From her hips all the way down to the floor", as his companion, Frankie the lab rat, would say. What could she possibly want from his little lab? "What can I do you for, er, I mean, what can I do for you?" he stammered. "I've got a problem", she said in a voice so husky it could pull a dog sled. "I've got control issues and I need an expert." "Expert, eh? I think I know someone, Miss…Miss…What did you say your name was?" "I didn't", she responded. "It's Miss C." Rick furrowed his brow. "Missy? Missy who?" "Just Miss C", came her curt reply.

Rick looked over his shoulder at his lab partner. "Frankie, we're going down the hall to visit The Italian. Finish up the prototype but don't Bogart that 'scope 'cause Louie needs it." He turned to the dame, "Follow me", he said. They walked down the hall in silence until they came to a door with a small red, white and green flag sticker on it. Rick knocked. The door opened revealing a young man with dark hair and a five day old beard that was perfectly trimmed around the edges. Inside the office sat a large framed photo of a Ferrari, a few postcards from San Marino and what appeared to be spare parts for some manner of medieval coffee machine. "Si?" said the young man, one eyebrow raised, looking like a cross between a GQ model and Mr. Spock.

"Arduino, this is Missy", said Rick. "Che?" came the young man's response. "Not Kay, C", said the dame. Arduino looked at her and asked, "Not Casey, but who?" She sensed a possible language barrier and tried to meet him halfway. "Miss C, si?" she said. "Missy C", nodded Arduino. "No", she replied, her exasperation increasing, "Not CC, what am I, a compiler?" Arduino was getting confused now. "Nazi see what??" he asked, his eyes scanning up and down the hallway. The dame tried a different approach. "OK, suppose I was your mother's sister. I'd be aunt C", she countered. "Yeah, well if you was my mother's sister I'd be antsy, too", added Rick. The dame moved in close to Arduino and almost whispered through clenched teeth, "Look, just call me Miss C, see?" "Got it!" said Arduino. "Please come in. How can I help you Miss Seesi?" Rick shook his head as he lowered his gaze to the ground. How had the script devolved into a bad parody of an Abbott and Costello routine? Puns were more to his liking. Surely he'd have to Warn her Brothers about this. Of all the labs in all the colleges and universities in the world, why did she have to walk into his?

"What seems to be the problem, Miss Seesi?" asked Arduino. The dame was beginning to get a headache, the kind that builds from behind the eyes until it feels like your skull is filled with monkeys playing dodge ball. She took a deep breath and exhaled slowly. "I've got an application that's in trouble", she said. "*Trouble*," thought Rick, "yeah, I'll bet this dame knows all about *trouble*. It's part of the package with these ones, must be in their DNA. Why, if I was a bug she'd be a regular *Venus De Flytrap* – beautiful *and*

---

[11] If you're not familiar with the 1942 film classic *Casablanca* starring Humphrey Bogart and Ingrid Bergman, as well as the *film noir* genre, it is strongly suggested that you watch the movie and a film noir title before proceeding. The attitudes expressed here are not necessarily those of the author.

deadly." Arduino clasped his hands together and smiled. "Trouble? No trouble! You've come to the right place. Tell me more about this *trouble*." "Well, I've got to control a bunch of devices; LEDs, motors, actuators, the usual crew", she started, "and I've got to obtain some information from the person running the system, you know the kind, a real operator. Lots of settings; pushbuttons, switches, potentiometers, the whole megillah. I tell ya, I'm in over my head."

Arduino walked across the office and nibbled at some biscotti. He turned back to her, the twice-baked biscuit still at his fingertips, held softly the way an orchestra conductor might balance a baton. "This application…" he said, his head slightly raised and cocked to one side revealing the beginning of a knowing smile, "…it needs to be flexible, expandable and inexpensive, too?" He had piqued the dame's interest but she couldn't let it show. "Yes. Yes it does", she responded coolly. "In that case", said Arduino, "allow me to introduce Uno, the One." He tossed the remnants of the biscotti to Rick and picked up a small blue printed circuit board containing a few ICs, a bunch of headers with numbered pins and what looked like USB and power connectors.
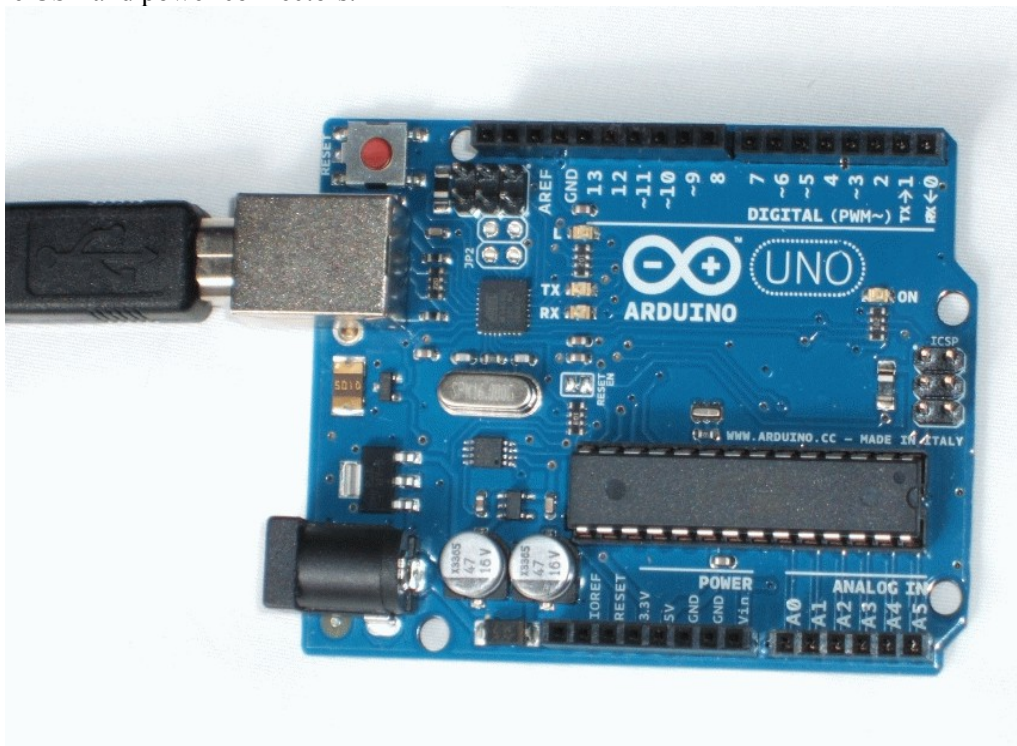
"Pretty spiffy", she said. "What is it?" Arduino's eyes lit up. "It's an open source microcontroller development board, and when I say 'open source' I mean both software *and* hardware. The software distribution includes extensive examples, complete library source code, and all the data you could want on the AVR." "What's an AVR?" came the dame's reply. "AVR is a line of microcontrollers from Atmel. The Uno uses an ATmega 328P with 32k of on-chip programmable flash memory, 2k of SRAM data memory and 1k of EEPROM. 32 general purpose registers. Pipelined Reduced Instruction Set Computer with Harvard architecture. Most instructions require only one tick of the 16 MHz clock." Arduino's concentration was broken by a hissing sound coming from behind a large stack of books. "Ahhh! Can I interest either of you in a cup of espresso?" he asked. "Never after lunch", said Rick. He knew Arduino's hyper-caffeinated concoction would have him bouncing off the walls all afternoon. "And you, Miss Seesi?" Arduino queried. "Maybe just this once," she relied, "with milk." Arduino smiled. "Cappuccino it is!"

Sipping the steaming beverage, she looked over the board intently. "The labeled port pins on the headers are convenient- seems pretty small though. Are you sure it can handle the job?" she asked. "No problem!" said Arduino. He pulled out a couple of slightly crumpled and dog-eared pieces of paper from under a stack of CD-ROMs and placed them on the desk. First was a block diagram of the 328P as used on the Uno, the second a more detailed schematic of the board:
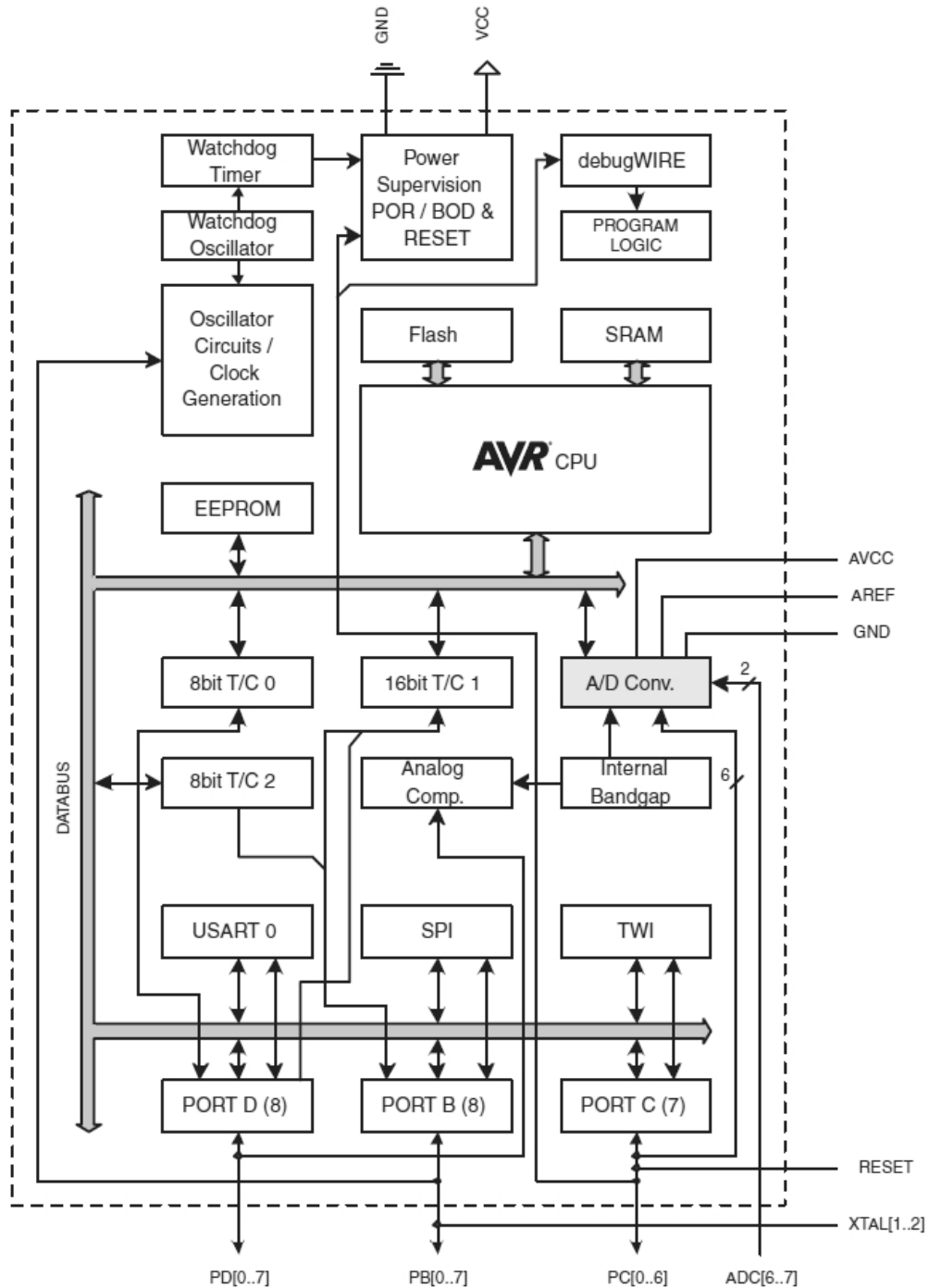
Arduino(TM) UNO Rev3
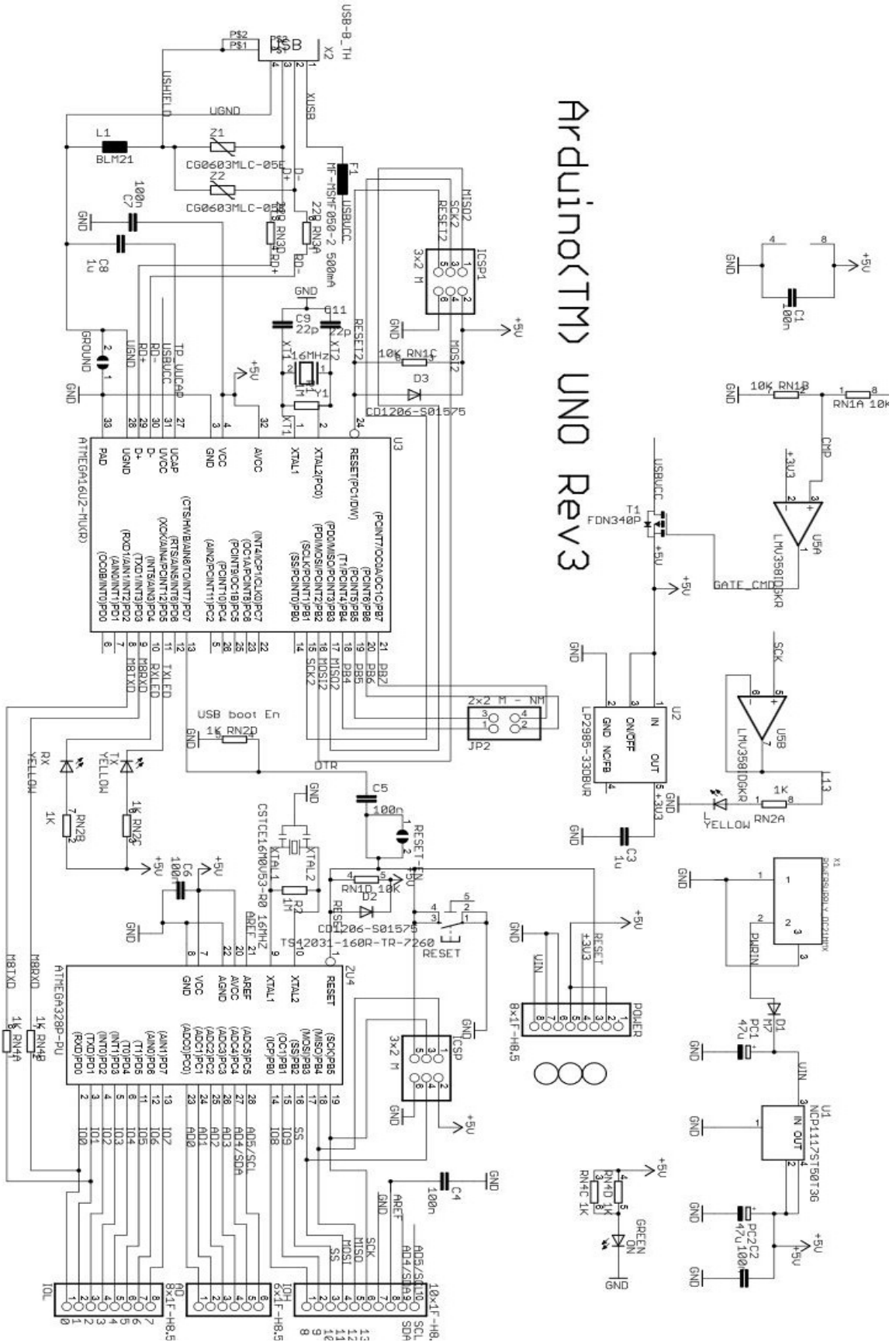
"You've got access to three IO ports", he began, "B, C and D. Digital IO with programmable pull-down resistor. All IO is memory mapped so you can read from and write to external devices as if they were just ordinary variables using the C programming language. A Universal Synchronous-Asynchronous Receiver-Transmitter, or USART, already programmed to communicate with a host computer via USB." She interrupted him. "So I can send text back and forth between the two? That would make debugging pretty easy." "You bet!" said Arduino, obviously getting excited. "Not only that, but you've got six 10 bit analog to digital converter channels and three timer-counters; two eight bit and one 16 bit. Plus, six outputs can generate pulse width modulation signals." "What about interrupts?" she asked. "I'll need interrupts." "Multiple levels," said Arduino, "internal and external. There's a reset button already on the board."

"What about power?" she asked. "Multiple options", said Arduino. "You can power it from the USB cable. You know, USB will supply up to 100 mA to an un-enumerated device and up to 500 mA to an enumerated device. If that's not enough, you can also plug in an external supply, the wall-wart kind, or even hook in a regulated five volt source so you can supply power to relays, motors, whatever. You'd power the big wire items off-board, of course. No need to run those currents through the board traces. Whatever you choose, though, the Uno will intelligently figure out where to get its power from."

The dame was getting more interested and needed further detail. "Great, but what will the chip deliver, you know, fan out or drive." "Generally speaking," Arduino began, "the IO pins can sink or source up to 40 mA each. The entire microcontroller should be limited to 200 mA total draw to stay within thermal limits so you obviously can't drive a whole bunch of pins to maximum current capability at the same time. No big deal, that's why they make drive circuits, right?" Rick and the dame both nodded knowingly.

The office was silent as the three of them peered at the little board. The dame looked up. "What about the programming interface? Command line or IDE?" Arduino waved his hand and responded, "The IDE runs under Windows, Mac OS and Linux. You can bypass that if you want and go command line, but…" "Fine, fine", she interrupted, "and this library, it's a straight jacket, right?" "No, no!" insisted Arduino. "It's a nice library and you can use as much or as little of it as you want. Even insert assembly op codes." Assembly op codes. The thought sent a shiver down Rick's spine. He had spent a month one day trying to debug a device driver written in 8086 assembly. Never again.

It seemed that Arduino was holding something back. "Sounds good but you're not giving me the whole story are you?" she asked him. "What's the catch?" "No catch," replied Arduino, "but there's one thing. One small thing that sometimes bites the beginners." The dame raised her eyebrows and demanded, "And?"

"Like most controllers," Arduino began, "the AVR uses a memory mapped IO address for writing to a port. For example, you might write to PORTB to light an LED." "So?" the dame responded, "That's not odd." "Very true," said Arduino, "but when it comes to reading from those same physical pins, they use a different address, in this case PINB, instead of reusing PORTB."

"Wait," stammered the dame, "it's the same physical pin and if I write to it, I use PORTB but if I read from it I use PINB?"

"Don't confuse pins and ports", said Arduino as a look of melancholy crept across his face. "You must remember this, a port is just a port, a pin is just a pin. The fundamental things apply, as the clock ticks by." The dame was shaken. "But I…" she started. "Look, you gotta get the IO straight, understand?" said Arduino. His expression grew serious and he looked at her squarely. "You gotta get it plain or you'll regret it. Maybe not today, maybe not tomorrow, but soon and for the rest of your life!" Her hand brushed lightly across the Uno as she turned, and wiping a tear from her eye, she headed out the office door. Rick

watched as she walked briskly down the hall. Some distance past the lab her silhouette disappeared into a fog that had mysteriously formed out of nowhere accompanied by the dull roar of a DC3's idling engines.

"She's got the Uno", said Rick. Arduino nodded, "Yes, I know. I've got plenty more where that one came from. All I need is a good lab tech to help me build more prototypes for Project Falcon using them. You interested?" A wry smile grew on Rick's face. "This Falcon, is it Maltese?" "Why yes, yes it is" came the response. "Arduino," Rick said, "I think this is the beginning of a beautiful friendship."[12]

---

[12] Complete information on the Arduino Uno development board may be found at http://arduino.cc/en/Main/ArduinoBoardUno

The Arduino system language reference may be found at http://arduino.cc/en/Reference/HomePage

Arduino tutorial info may be found at http://arduino.cc/en/Tutorial/HomePage

Arduino software downloads are found here: http://arduino.cc/en/Main/Software

When in doubt, just go here: http://www.arduino.cc and start looking.

18. Bits & Pieces: #include and #define

18.1 Introduction

Welcome to Bits & Pieces (catchy name, eh?). This sequence of chapters delves into a variety of aspects of programming the Atmel AVR ATmega 328P microcontroller via the Arduino Uno board and associated libraries. The focus is on IO (Input-Output) programming. Specifically, we'll be looking at ways of coding the interface between the Uno and external circuitry. This is code-only; the hardware aspect is dealt with separately in lab. We will be exploring the IO code used to read to and write from digital and analog ports so that we can perform tasks such as reading the state of switches, lighting LEDs, reading continuously variable data such as force or temperature, timing or counting events, and controlling devices such as motors. The simplest way of performing these items is via the library functions that come with the Arduino system[13]. Sometimes, though, these functions are not fast or efficient enough, or we need to use a non-Arduino platform. For those reasons, we'll be diving down into the library functions themselves to see how they work. To the uninitiated, scouring through library source code can be a daunting task, so think of Bits & Pieces as a series of guided tours covering major parts of the library. Granted, it's probably not as much fun as, say, a series of guided tours of tropical islands of the South Pacific, but you can only get so much for your tuition. And while there is little doubt that there might be interesting uses for embedded controllers on said islands, a field trip is *right out*.

The first things we need to examine are some commonly used include files. Recall that include files (also known as header files, i.e., the ones that end in ".h") are used to collect up function prototypes, references to global variables, structure definitions, and those wonderful (and sometimes confusing) `#define`s. Remember, the C language is fairly "skinny" and the functions that we call are either written by us or come from a library. Prototypes for the functions are found in the library's header file. An interesting twist to this is that some so-called library functions aren't functions at all. Instead, they are simply inline expansions via `#define`s. Many different libraries are available, so to make life even easier, systems often include a "master" include file that contains directives to reference other include files (boy, those professional programmers are constantly looking for ways to make every keystroke count).

18.2 Universal Stuff, Common to All Controllers

Let's start with the biggie: `arduino.h`. Here are the first few lines (some parts omitted):

```
#ifndef Arduino_h
#define Arduino_h

#include <stdlib.h>
#include <string.h>
#include <math.h>

#include <avr/pgmspace.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include "binary.h"
#define HIGH  0x1
```

---

[13] Complete details on the library and lots of other goodies including example code can be found at www.arduino.cc. The Reference page in particular will prove useful.

```
        #define LOW    0x0
        #define true   0x1
        #define false 0x0

        #define INPUT        0x0
        #define OUTPUT       0x1
        #define INPUT_PULLUP 0x2

        #define PI          3.1415926535897932384626433832795
        #define HALF_PI     1.5707963267948966192313216916398
        #define TWO_PI      6.283185307179586476925286766559
        #define DEG_TO_RAD  0.017453292519943295769236907684886
        #define RAD_TO_DEG 57.295779513082320876798154814105
```

The first two lines prevent accidental re-entry. That is, if the file has already been included it won't be included again (if it was, you'd get a whole bunch of redefinition errors). After this we see a selection of other commonly used library header files and then a series of constant definitions. There's nothing too crazy yet. Following this are a series of what look like functions but which are inline expansions. That is, the preprocessor replaces your "function call" with a different bit of code. This is done because inline expansions do not incur the overhead of function calls and thus run faster and use less memory. Notice how some of these make use of the ternary if/else construct, such as `min()`, while others make use of recently defined constants (`radians()`). Some, such as `noInterruprs()`, perform a substitution which itself will perform a further substitution (in this case, the `cli()` "function" will turn into a single assembly language instruction that turns off interrupts globally).

```
        #define min(a,b) ((a)<(b)?(a):(b))
        #define max(a,b) ((a)>(b)?(a):(b))
        #define abs(x)    ((x)>0?(x):-(x))
        #define constrain(amt,low,high) ((amt)<(low)?(low):((amt)>(high)?(high):(amt)))
        #define round(x)      ((x)>=0?(long)((x)+0.5):(long)((x)-0.5))
        #define radians(deg) ((deg)*DEG_TO_RAD)
        #define degrees(rad) ((rad)*RAD_TO_DEG)
        #define sq(x)         ((x)*(x))

        #define interrupts()   sei()
        #define noInterruprs() cli()

        #define clockCyclesPerMicrosecond() ( F_CPU / 1000000L )
```

Further down we find some `typedef`s, namely `uint8_t`, which is shorthand for an unsigned 8 bit integer, i.e., an `unsigned char`. This `typedef` was written in another header file but notice that we now have new `typedef`s based on that original `typedef` in the third and fourth lines! Thus, an `unsigned char` may now be declared merely as `boolean` or `byte`, and finally, an `unsigned int` may be declared as a `word`.

```
        #define lowByte(w)  ((uint8_t) ((w) & 0xff))
        #define highByte(w) ((uint8_t) ((w) >> 8))
        typedef uint8_t boolean;
        typedef uint8_t byte;
        typedef unsigned int word;
```

Common procedures in IO programming include checking, setting and clearing specific bits in special registers. Typically this is done through bitwise math operators. For example, if you want to set the 0th bit of a register called DDRB while leaving all other bits intact, you'd bitwise OR it with 0x01 as in:

```
DDRB = DDRB | 0x01;
```

You could also define specific bit positions like so:

```
#define LEDBIT   0
#define MOTORBIT 1
#define ALARMBIT 2
```

So if you want to set the bit for the motor, you would write:

```
DDRB = DDRB | (0x01<<MOTORBIT);
```

In other words, left shift a one in the zero bit location one place (resulting in the 1st vs. 0th bit being high) and OR the register contents with it. This is a little cumbersome unless you turn it into a function call:

```
bitSet( DDRB, MOTORBIT );
```

Now that's pretty easy and obvious, so check out the lines below:

```
#define bitRead(value, bit) (((value) >> (bit)) & 0x01)
#define bitSet(value, bit) ((value) |= (1UL << (bit)))
#define bitClear(value, bit) ((value) &= ~(1UL << (bit)))
#define bitWrite(value, bit, bitvalue) (bitvalue ? bitSet(value, bit) :
bitClear(value, bit))

#define bit(b) (1UL << (b))
```

Further along in the file we come across a bunch of function prototypes for items we will be using:

```
void pinMode(uint8_t, uint8_t);
void digitalWrite(uint8_t, uint8_t);
int  digitalRead(uint8_t);
int  analogRead(uint8_t);
void analogReference(uint8_t mode);
void analogWrite(uint8_t, int);

unsigned long millis(void);
unsigned long micros(void);
void delay(unsigned long);
void setup(void);
void loop(void);
```

The last two are particularly important. The Arduino development system has a pre-written `main()` function. It makes calls to `setup()` and `loop()`, so we'll be writing these as our primary entry points.

18.3 Controller Specific Stuff

Moving on to other header files, we must recall that there are dozens and dozens of models in a given processor series like the AVR. Each of these controllers will have different memory capacities, IO capabilities and so forth, so we need to distinguish which one we're using while also trying to keep the code as generic as possible. Normally, this is done by creating specific header files for each controller. The IDE then gives you an option to select which controller you're using and #defines a controller ID for it (see the list in the Arduino IDE under *Tools>>Board*). Consider the following chunk of avr/io.h:

```
#ifndef _AVR_IO_H_
#define _AVR_IO_H_

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

We find a (huge) series of conditional includes, each looking for the one pre-defined processor symbol set by the Arduino IDE:

```
#if defined (__AVR_AT94K__)
#   include <avr/ioat94k.h>
#elif defined (__AVR_AT43USB320__)
#   include <avr/io43u32x.h>
#elif defined (__AVR_AT43USB355__)
#   include <avr/io43u35x.h>
```

… and so on until we get to the ATmega 328P for the Arduino Uno:

```
#elif defined (__AVR_ATmega328P__)
#   include <avr/iom328p.h>
```

… and we continue until we get to the end:

```
#elif defined (__AVR_ATxmega256A3B__)
#   include <avr/iox256a3b.h>
#else
#   if !defined(__COMPILING_AVR_LIBC__)
#     warning "device type not defined"
#   endif
#endif

#include <avr/portpins.h>
#include <avr/common.h>
#include <avr/version.h>

#endif /* _AVR_IO_H_ */
```

So what's in avr/iom328p.h you ask? This includes a bunch of things that will make our programming lives much easier such as definitions for ports, registers and bits. We're going to be seeing these over and over:

```
#ifndef _AVR_IOM328P_H_
#define _AVR_IOM328P_H_ 1
```

```
/* Registers and associated bit numbers */
```

This is an 8 bit input port and associated bits

```
#define PINB _SFR_IO8(0x03)
#define PINB0 0
#define PINB1 1
#define PINB2 2
#define PINB3 3
#define PINB4 4
#define PINB5 5
#define PINB6 6
#define PINB7 7
```

This is an 8 bit data direction register and associated bits

```
#define DDRB _SFR_IO8(0x04)
#define DDB0 0
#define DDB1 1
#define DDB2 2
#define DDB3 3
#define DDB4 4
#define DDB5 5
#define DDB6 6
#define DDB7 7
```

This is an 8 bit output port and associated bits

```
#define PORTB _SFR_IO8(0x05)
#define PORTB0 0
#define PORTB1 1
#define PORTB2 2
#define PORTB3 3
#define PORTB4 4
#define PORTB5 5
#define PORTB6 6
#define PORTB7 7
```

…and so on for ports C and D. Now for analog to digital converter (ADC) goodies:

```
#ifndef __ASSEMBLER__
#define ADC      _SFR_MEM16(0x78)
#endif
#define ADCW     _SFR_MEM16(0x78)

#define ADCL _SFR_MEM8(0x78)
#define ADCL0 0
#define ADCL1 1
#define ADCL2 2
#define ADCL3 3
#define ADCL4 4
#define ADCL5 5
#define ADCL6 6
#define ADCL7 7
```

```
#define ADCH _SFR_MEM8(0x79)
#define ADCH0 0
#define ADCH1 1
#define ADCH2 2
#define ADCH3 3
#define ADCH4 4
#define ADCH5 5
#define ADCH6 6
#define ADCH7 7

#define ADCSRA _SFR_MEM8(0x7A)
#define ADPS0 0
#define ADPS1 1
#define ADPS2 2
#define ADIE 3
#define ADIF 4
#define ADATE 5
#define ADSC 6
#define ADEN 7

#define ADCSRB _SFR_MEM8(0x7B)
#define ADTS0 0
#define ADTS1 1
#define ADTS2 2
#define ACME 6

#define ADMUX _SFR_MEM8(0x7C)
#define MUX0 0
#define MUX1 1
#define MUX2 2
#define MUX3 3
#define ADLAR 5
#define REFS0 6
#define REFS1 7
```

… and so on for the remainder of the header file. OK, so just what is this nugget below?

```
#define PORTB _SFR_IO8(0x05)
```

Most controllers communicate via *memory-mapped IO*. That is, external pins are written to and read from as if they are ordinary memory locations. So if you want to write to a port, you can simply declare a pointer variable, set its value to the appropriate address, and then manipulate it as needed. For the ATmega 328P, the address of IO Port B is 0x25. You could write the following if you wanted to set bit 5 high:

```
unsigned char *portB;

portB = (unsigned char *)0x25; // cast required to keep compiler quiet
*portB = *portB | 0x20;
```

You could also use the bit setting function seen earlier which is a little clearer:

```
bitSet( *portB, 5 );
```

The problem here is that you have to declare and use the pointer variable which is a little clunky. There's also the error-prone business of pointer de-referencing in the assignment statement (the `*` in front of the variable which beginning programmers tend to forget). Besides, it requires that you look up the specific address of the port (and change it if you use another processor). So to make things more generic we place the addresses (or more typically, offsets from a base address) in the processor-specific header file.

The header file declares an item, `PORTB`, for our convenience. It is defined as `_SFR_IO8(0x05)` but what's that `_SFR_IO8` function? In sfr_defs.h it's defined as follows:

```
#define _SFR_IO8(io_addr) _MMIO_BYTE((io_addr) + __SFR_OFFSET)
```

and `__SFR_OFFSET` is defined as 0x20. In other words, this item is 0x05 above the base offset address of 0x20, meaning it's the address 0x25 as we saw earlier. But what the heck is `_MMIO_BYTE()`? That looks a little weird at first glance:

```
#define _MMIO_BYTE(mem_addr) (*(volatile uint8_t *)(mem_addr))
```

This is just a cast with a pointer de-reference. It says this item is a pointer to an unsigned char which is being de-referenced (de-referenced by the first `*` and don't forget the earlier `typedef` for the `uint8_t`). By placing all of these items in header files we've managed to make the IO programming generic while at the same time removing the need for pointer variable declarations and the need for pointer de-referencing[14].

Therefore, if we want to set bit 5 high, we can now just say

```
PORTB = PORTB | 0x20;   // or more typically: PORTB |= 0x20;
```

or

```
bitSet( PORTB, 5 );     // note lack of "*" in both lines of code
```

All the business about exactly where PORTB is located in the memory map is hidden from us and there will be no accidental errors due to leaving out the `*`. The same code will work with several different processors, and to top it off, it's operationally more efficient than dealing with pointers as well. Consequently, we will normally use these symbolic register and port names rather than hard addresses in future work. Simply lovely. Time for a snack; something fruity perhaps, something that reminds us of tropical islands…

---

[14] If you're wondering about `volatile`, recall that it's a modifier that indicates that the variable can be changed by another process (possibly an interrupt). This will prevent overly aggressive assembly code optimizations by the compiler.

19. Bits & Pieces: Digital Output Circuitry

19.1 Introduction

To fully appreciate the software interface to microcontrollers and to gain insight into how to interface controllers to external devices, it is useful to examine the underlying circuitry. In this chapter we shall investigate the basic digital output circuitry. Anything that can be controlled with a simple logical yes/no, true/false input can be a target. Although controllers usually have limited output current and voltage capability, they can control devices such as power transistors which can in turn control more demanding loads. A simplified diagram of the General Purpose Input-Output (GPIO) circuitry can be seen in Figure 19.1.

PUD:	PULLUP DISABLE
SLEEP:	SLEEP CONTROL
clk$_{I/O}$:	I/O CLOCK

WDx:	WRITE DDRx
RDx:	READ DDRx
WRx:	WRITE PORTx
RRx:	READ PORTx REGISTER
RPx:	READ PORTx PIN
WPx:	WRITE PINx REGISTER
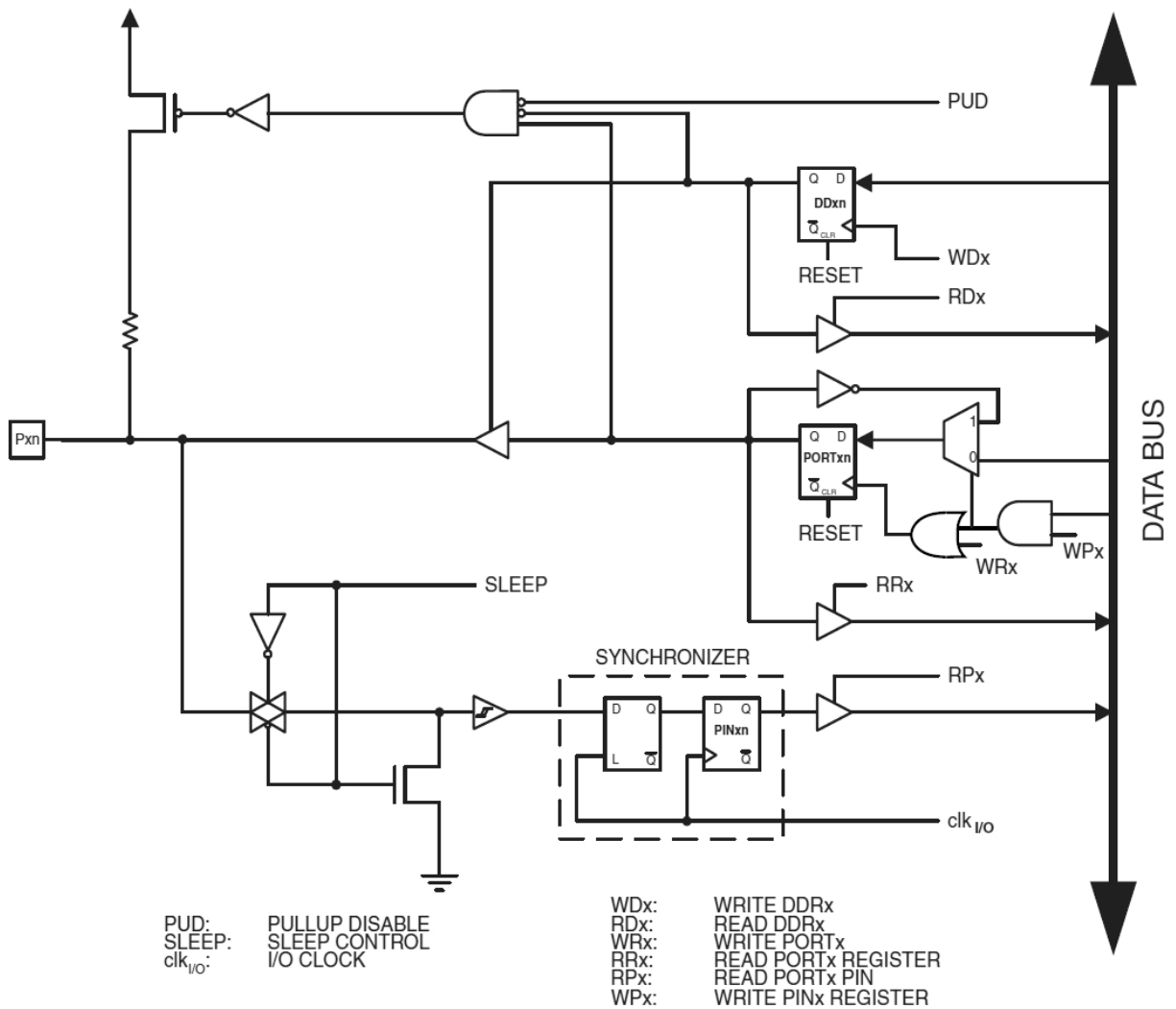
Figure 19.1, GPIO circuitry (from Atmel 2014)

The circuit represents a single bit. An IO port consists of eight bits, typically. Therefore this circuitry would be replicated seven more times to accommodate a single byte-wide port, of which there may be several. Also, in order to reduce the external pin count of the IC, pins may be multiplexed with other functions. The multiplexers are not shown in this schematic. Pxn, at the far left of the schematic, represents the physical output pin. The data bus shown along the right edge is the source for data being written to the physical pin and also the destination for data being read from the pin. For now we shall focus on writing to the output and examine input functionality in a later chapter. While this circuit is specific to the ATmega series, it is representative of GPIO circuitry found in many microcontrollers.

19.2 Output Circuitry

Figure 19.2 presents a further simplification focusing solely on the output portion.

Figure 19.2, output circuitry simplified (from Atmel 2014)

First off, the PUD (Pull Up Disable), SLEEP and CLK (CLocK) control lines are common to all bits and ports. We may ignore them for our purposes. The lowest section is removed because it is involved with input functionality. The same goes for the upper section surrounding the MOSFET. Finally, a series of two gates, an inverter and a multiplexer clustered around the middle section have been removed. These serve a particular (and perhaps somewhat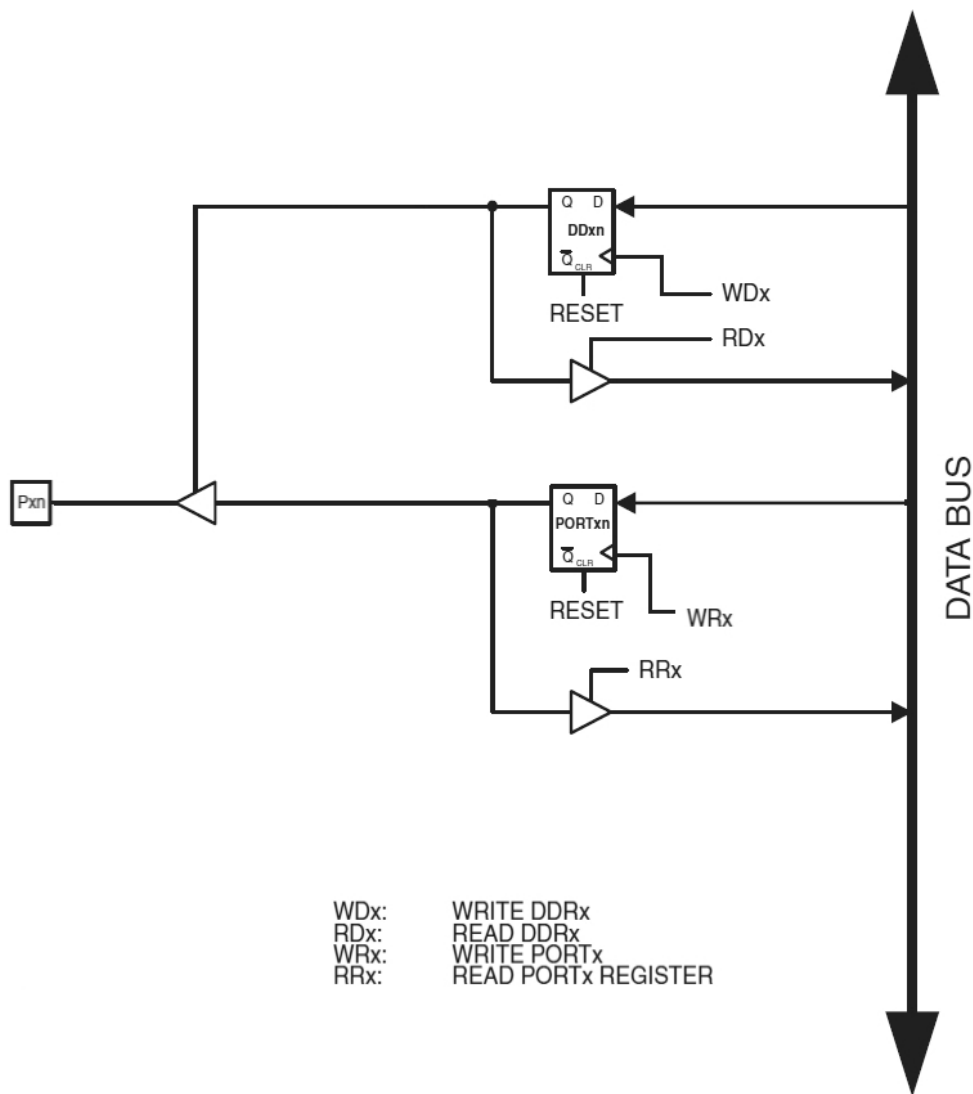 esoteric) function, namely the ability to rapidly toggle a bit. Again, this section is not needed for typical functioning.

Upon simplification we are left with a circuit that centers around two D flip-flops and a few tri-state buffers. The primary signals of interest are WDx and WRx which feed the DDxn and PORTxn flip-flops. DDxn is the Data Direction bit. It determines whether the physical pin is configured for output (writing) or input (reading). PORTxn presents the data that needs to be written. Note that the "x" refers to the port letter, as in Port <u>B</u>, while the "n" refers to the bit number within that port. Thus, the physical pin for bit number two in Port B would be denoted here as PB2 (or alternately, PORTB.2). The collection of all eight bits of DDxn is referred to as the Data Direction Register, or DDR. The Data Direction Register for Port B would be referred to as DDRB. Each port will have these registers mapped in memory. That is, for Port B there will be a DDRB for direction control and a PORTB for writing data (there is also a PINB for reading, more on that later). Similarly there will a DDRC and PORTC (and PINC) for Port C, and so on for as many ports as the controller has (B, C and D for the ATmega 328P).

To understand how the circuit works, recall that a D flip-flop's Q output simply echoes the logic level present at the D input when the control signal transitions from low to high (i.e., positive edge trigger). Note that the output of PORTxn feeds a tri-state buffer which in turn feeds the physical pin. In order to write data to the pin, a logic high is first placed on the data bus. The WDx signal is pulsed which transfers the logic high to Q of DDxn. This high level enables the tri-state buffer connected to Pxn. The port bit is now configured for output and it will stay in this mode until WDx is re-asserted. The desired data (high or low bit) is now placed on the data bus. The WRx signal is pulsed which transfers the logic level to Q of PORTxn. This feeds the afore-mentioned tri-state which transfers the data bit to the output. This level will remain until WRx is re-asserted. If WDx and WRx are never re-asserted, the output pin level will never change. In order to write new data to the output pin, the desired data bit is placed on the bus and WRx is pulsed. It is not necessary to re-load DDxn and re-assert WDx each time.

For example, suppose we wish to flash an LED several times. We could do this by attaching an LED to the pin and then toggling the pin from high to low repeatedly. First, we would write a high to DDxn to establish output (write) mode. Then we would write a high to PORTxn, turning on the LED. After a short wait we would write a low to PORTxn, turning off the LED. After a further short wait we would write another high to PORTxn (turning the LED back on) and continue the process in like manner for as many flashes as we need.

The final two tri-state buffers associated with the RDx and RRx signals allow us to read the current states of the direction and port bits.

20. Bits & Pieces: Digital Input Circuitry

20.1 Introduction

In this chapter we shall investigate the basic digital input circuitry. This can be used to sense the state of external switches and other two-state devices. These devices can be active, that is, generating a high or low voltage, or they can be simple passive switches connecting to ground through the use of an optional internal pull-up resistor. A simplified diagram of the General Purpose Input-Output (GPIO) circuitry can be seen in Figure 20.1. As in the previous chapter, this is for a single bit. Once again, we shall remove sections that are not pertinent to the input function or for enhanced clarity.
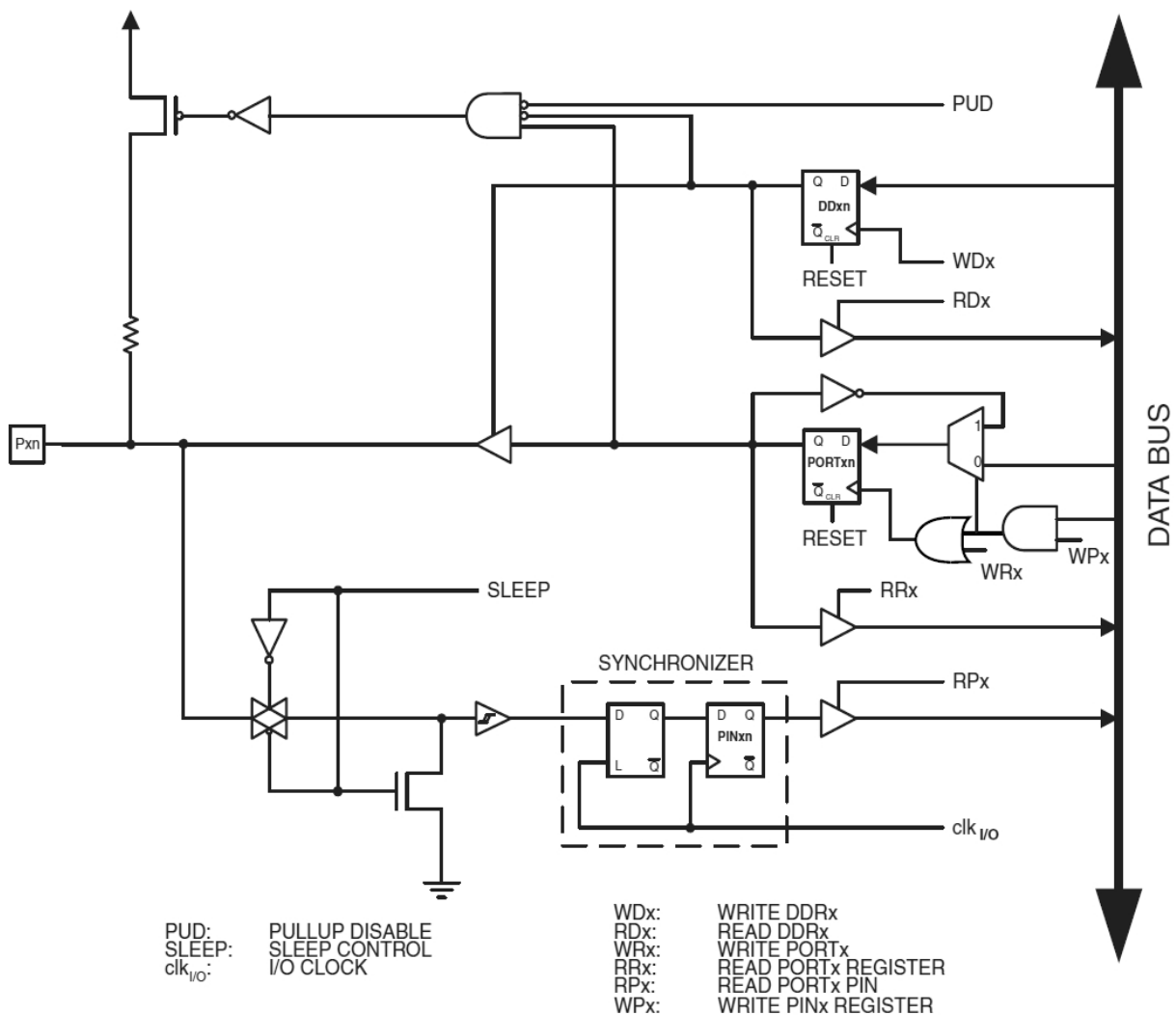
Figure 20.1, GPIO circuitry (from Atmel 2014)

20.2 Input Circuitry

Figure 20.2 presents a further simplification focusing solely on the input portion.

WDx: WRITE DDRx
RDx: READ DDRx
WRx: WRITE PORTx
RRx: READ PORTx REGISTER
RPx: READ PORTx PIN
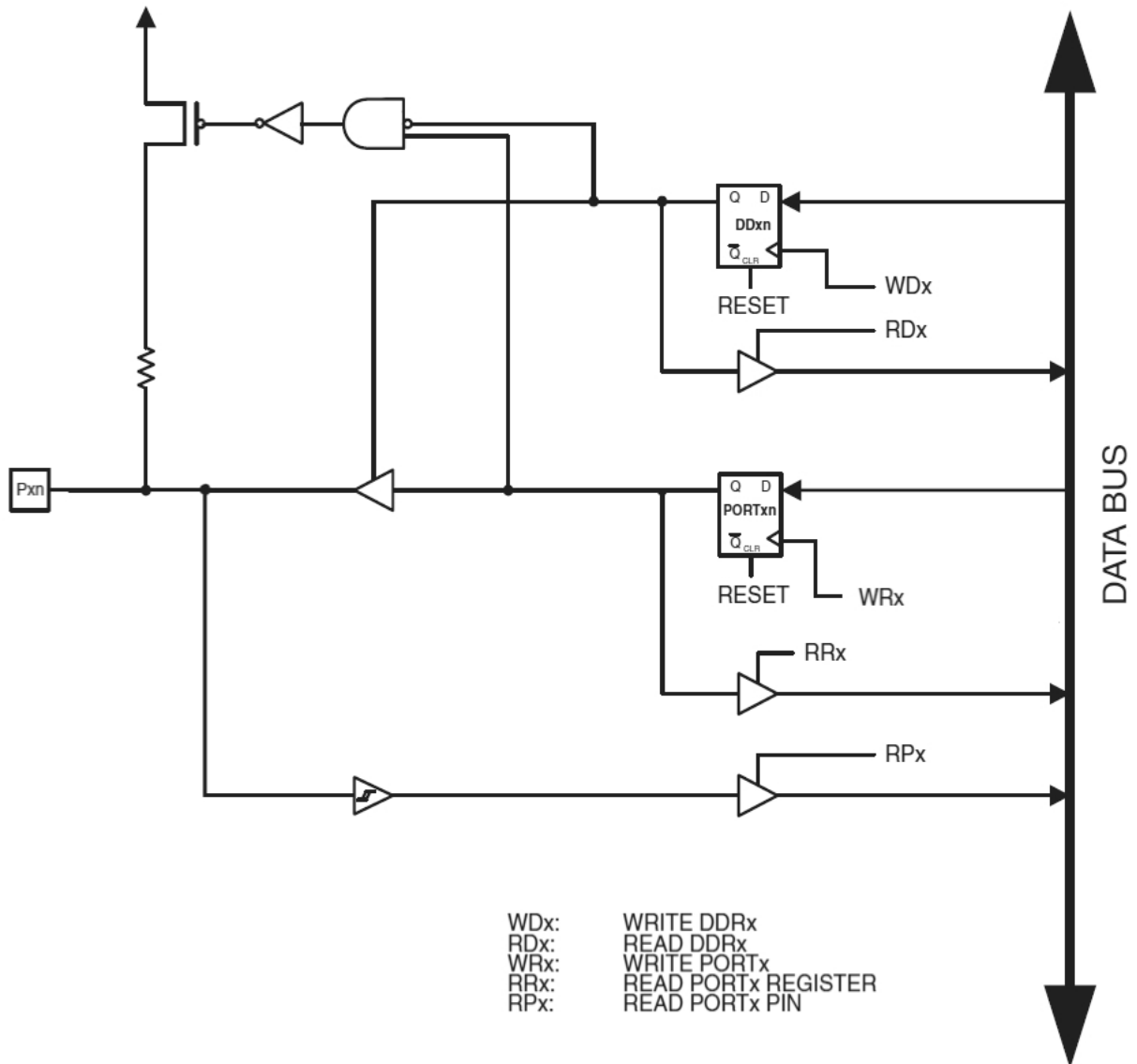
Figure 20.2, input circuitry simplified (from Atmel 2014)

Once again we have removed the gates surrounding PORTxn that create the bit toggle function. Further, we have removed the PUD, SLEEP and CLK signals and simplified the lower section leaving just the Schmitt Trigger. The read process will be similar to the write process examined in the previous chapter.

To read a signal on an external pin, we will need to write a logic low to the data direction bit DDxn. This will disconnect PORTxn from the physical pin, Pxn, because the tri-state buffer will go to high-Z state. If

PORTxn was left connected, the external device would wind up trying to drive the active output of PORTxn. This would produce unpredictable results at best.

Now that PORTxn is disconnected from Pxn, the signal existing at Pxn drives the Schmitt Trigger located along the bottom edge of the diagram. This signal will be passed on to the data bus once the RPx (Read Pin x) control signal is asserted on the lower tri-state buffer. A Schmitt Trigger is used as an intermediary here as we cannot be certain of clean, noise-free input signals. Also, note that the input signals are read from a PIN register (e.g., PINB or PINC). This is in contrast to the PORT registers used when writing data. Thus, if you want to write data to the outside world, you write to a PORT register. If you want to read data from the outside world, you read from a PIN register. Not all microcontrollers use this naming convention. Some use a single PORT register for both writing and reading. A good mnemonic for remembering whether PIN or PORT is used for input or output is to associate the "in" in PIN and the "o" (for output) in PORT. Typically, the PORT, PIN and DDR registers are adjacent in the memory map. For example, using port B we find PINB at 0x23, DDRB at 0x24 and PORTB at 0x25. Check the register map in the Appendix for more.

We are still left with the MOSFET section. This is used to generate an optional pull-up resistor. There are two basic ways to generate input signals. The first is with an active circuit. For example, the output of a logic gate could be connected to the external pin. The voltage produced by the gate would be fed to the Schmitt Trigger as described previously and the signal would make its way to the data bus. It is not always convenient to generate an external voltage with an active circuit, though. If you wanted to read the state of a simple push button, you couldn't just connect it between the pin and ground as there would be no signal to sense. Instead you'd have to connect it to an external power supply through a limiting resistor, requiring more components and space. The pull-up takes care of this for you. When activated, the MOSFET connects its associated resistor to the internal power line. If a switch is now connected between the external pin and ground, and is closed, the pin is pulled to ground (i.e., logic low). If the switch is opened, the resistor "pulls up" the pin voltage to the supply rail (i.e., logic high). Thus, we can determine the state of the switch with no other external circuitry.

To activate the pull-up, first the pin must be in input mode (DDxn at logic low which feeds the AND gate). Also, the output of PORTxn must be logic high. These two signals will drive the AND gate's output high, turning on the MOSFET and engaging the pull-up resistor. It might seem a little odd to be writing to the PORT bit for read mode but this is just a neat way of making the most of the available hardware. After all, in read mode, the PORT register is otherwise unused. Why let it go to waste and require yet another register to control the pull-up?

In summary, to read from the external pin, a low is written to the appropriate DDR bit, placing this circuit in read (input) mode. If a pull-up is desired, a high is written to the appropriate PORT bit. If a pull-up is not desired, a low is written to the PORT bit. Once this is done, the external value can be read from the PIN register. As with the output circuitry, the DDR and PORT do not have to be rewritten prior to each subsequent read from the PIN. The D flip-flops will hold their values until they are rewritten, thus maintaining the current data direction and pull-up status.

21. Bits & Pieces: pinMode()

Or "Programming Port Directions and Your Bicycle"

In this tour, we're going to start looking at digital IO. Digital IO will allow us to read the state of an input pin as well as produce a logical high or low at an output pin. Examples include reading the state of an external switch and turning an LED or motor on and off.

If every potential external connection between a microcontroller and the outside world had a dedicated wire, the pin count for controller packages would be huge. The ATmega 328P in the Uno board has four 8 bit ports plus connections for power, ground and the like, yet it only has 28 physical pins. How is this possible? Simple, we'll *multiplex* the pins, that is, make multiple uses for each. If, for example, we were to look at the 0th bit of IO port B, this leads to a single external pin. This pin can be programmed to behave in either input (read) mode or output (write) mode. In general, each bit of a port can be programmed independently; some for input, some for output, or all of them the same. Obviously, before we use a port we need to tell the controller which way it should behave. In the Arduino system this is usually done via a call to the library function `pinMode()`. Here is the description of the function from the on-line reference[15]:

---

pinMode()

Description

Configures the specified pin to behave either as an input or an output. See the description of digital pins for details on the functionality of the pins.

As of Arduino 1.0.1, it is possible to enable the internal pullup resistors with the mode INPUT_PULLUP. Additionally, the INPUT mode explicitly disables the internal pullups.

Syntax

pinMode(pin, mode)

Parameters

pin: the number of the pin whose mode you wish to set

mode: INPUT, OUTPUT, or INPUT_PULLUP. (see the digital pins page for a more complete description of the functionality.)

Returns

None

Figure 21.1, pinMode docs

---

[15] http://www.arduino.cc/en/Reference/PinMode

So we'd first have to think in terms of an Arduino pin number instead of a port bit number. Below is a table of Arduino Uno pin designations versus ATmega 328P port and pin naming.

Arduino Designator	General Purpose IO Designator	Comment
A0	PORTC bit 0	ADC input 0
A1	PORTC bit 1	ADC input 1
A2	PORTC bit 2	ADC input 2
A3	PORTC bit 3	ADC input 3
A4	PORTC bit 4	ADC input 4
A5	PORTC bit 5	ADC input 5
0	PORTD bit 0	RX
1	PORTD bit 1	TX
2	PORTD bit 2	
3	PORTD bit 3	PWM
4	PORTD bit 4	
5	PORTD bit 5	PWM
6	PORTD bit 6	PWM
7	PORTD bit 7	
8	PORTB bit 0	
9	PORTB bit 1	PWM
10	PORTB bit 2	PWM
11	PORTB bit 3	PWM
12	PORTB bit 4	
13	PORTB bit 5	Built-in LED

Figure 21.2, Arduino Uno Pin Definitions

The naming convention is reasonably logical but less than perfect. Note that the A0 through A5 designators are for the analog inputs and the remaining are for digital IO. A0 through A5 are, in fact, predefined global constants that map back to numeric values (e.g., A0 is 14 for the Uno, see the file pins_arduino.h for details). It should also be noted that the analog channels are input-only. The controller cannot produce continuously variable analog voltages on its own. This is not to say that it's impossible to get analog control; just that it's going to take a little more work (as we shall see soon enough).

Practically speaking the pin naming convention isn't all that bad as the pins are labeled right on the board, see Figure 21.3.
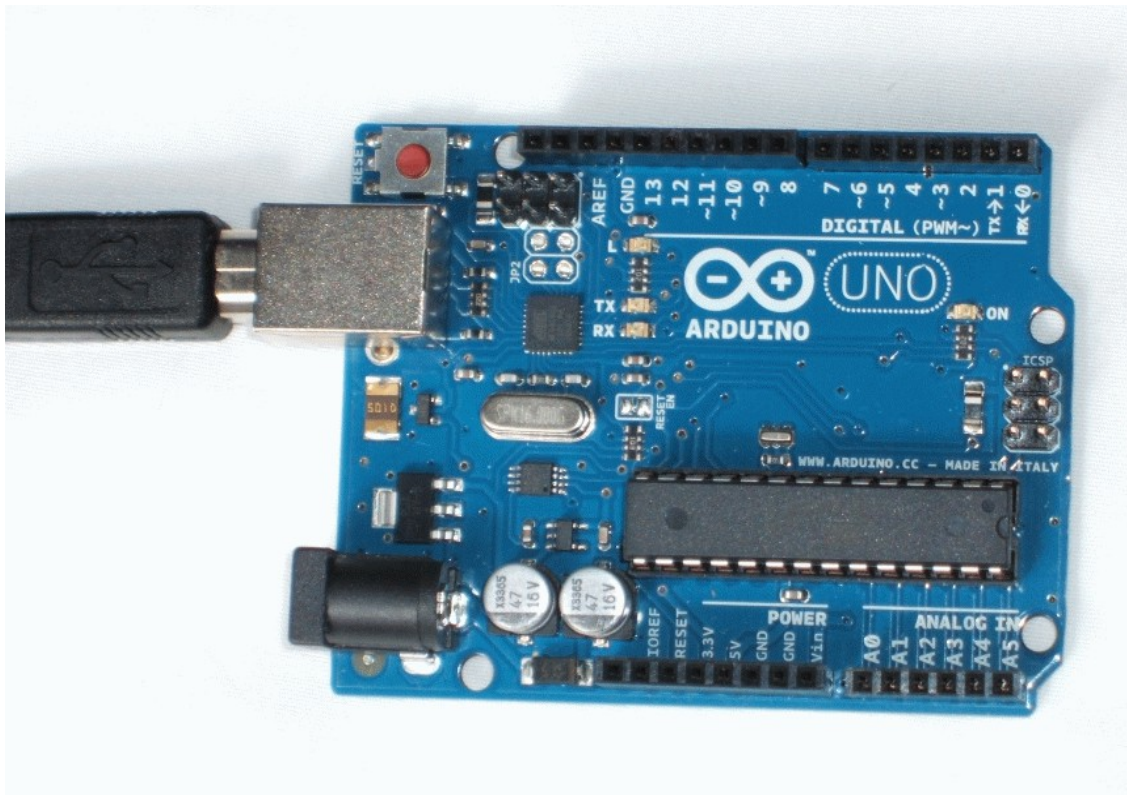
Figure 21.3, Arduino Uno

For example, directly above the Arduino Uno logo you can spot an "8" next to a pin located at the edge of a 10 pin header. According to the table above, this is bit 0 of port B. To set this connector to output mode to drive an external circuit, you could write:

```
pinMode( 8, OUTPUT );
```

OK, so what if we're using this controller on a non-Arduino system, and is there a faster way to accomplish this (code execution-wise)? Let's take a look at the code for this function. It's found in a file called wiring_digital.c. Here are the relevant bits and pieces (slightly altered to make some portions a little more clear):

```
void pinMode(uint8_t pin, uint8_t mode)
{
    uint8_t bit, port, oldSREG;
    volatile uint8_t *reg, *out;


    bit = digitalPinToBitMask( pin );
    port = digitalPinToPort( pin );

    if (port == NOT_A_PIN) return;  // bad pin # requested

    reg = portModeRegister( port );
    out = portOutputRegister( port );
```

Embedded Controllers

```
    if (mode == INPUT)
    {
          oldSREG = SREG;
          cli();
          *reg &= ~bit;
          *out &= ~bit;
          SREG = oldSREG;
    }
    else
    {
          if (mode == INPUT_PULLUP)
          {
                oldSREG = SREG;
                cli();
                *reg &= ~bit;
                *out |= bit;
                SREG = oldSREG;
          }
          else // must be OUTPUT mode
          {
                oldSREG = SREG;
                cli();
                *reg |= bit;
                SREG = oldSREG;
          }
    }
}
```

Figure 21.4, pinMode code

After the declaration of a few local variables, the specified Arduino pin number is decoded into both a port and its associated bit mask by the `digitalPinTo…()` functions. If a pin is specified that does not exist on this particular processor, `port` will be set to an error value so we check for this and bail out if it's so. What's a port bit mask? That's just an 8 bit value with all 0s except for a 1 in the desired bit position (sometimes it's the complement of this, that is, all 1s with a 0- it depends on whether you intend to set, clear, AND or OR). The `port…Register()` functions perform similar decoding operations. For example, the so-called "port mode register" is better known as the data direction register, or DDR. The ATmega 328P has three of these, `DDRB` through `DDRD`. Setting a bit sets the mode to output (write) while clearing puts it in input (read) mode.

These four "functions" are really look-up tables disguised as functions. It's merely an access to an array filled with appropriate values. Accessing an array is much faster than calling a function. Consider the `portModeRegister()` function, which given a port designator, will return the actual data direction port which we can manipulate (e.g., `DDRC`). This is defined as:

```
#define portModeRegister(P)
      ( (volatile uint8_t *)( pgm_read_word( port_to_mode_PGM + (P))) )
```

`port_to_mode_PGM` turns out to be an array[16] filled with the pre-defined error symbol and the addresses of the appropriate data direction registers:

```
const uint16_t PROGMEM port_to_mode_PGM[] = {
        NOT_A_PORT,
        NOT_A_PORT,
        (uint16_t) &DDRB,
        (uint16_t) &DDRC,
        (uint16_t) &DDRD,
};
```

`pgm_read_word` is a special function used to read values from the program space versus the normal data space (remember, this controller uses a Harvard architecture with split memory).

The end result is that we'll get back the address of the required data direction register and that's what we'll need to access in order to set input or output mode.

The next chunk of code is just three if/else possibilities, one for each of the modes we could request. Let's take a look at `OUTPUT` mode first.

```
        else
        {
                oldSREG = SREG;
                cli();
                *reg |= bit;
                SREG = oldSREG;
        }
```

Probably the single most important register on a microcontroller is the *status register*, here called `SREG`. It contains a bunch of flag bits that signify a series of states. For example, there is a bit to indicate that an arithmetic operation overflowed (i.e., there were too few bits to hold the result). There's a bit to indicate the result is negative, and so forth. One of the bits controls whether or not the system will respond to *interrupts*. An interrupt is a high priority event that can halt (interrupt) the execution of your code while the interrupt code (ISR, or interrupt service routine) runs instead. An interrupt can occur at any time, even when your code is in the middle of something important. To prevent this from happening, we first save the current contents of the status register and then clear the status bit that enables interrupts from occurring. That's the `cli()` call (which, as it turns out, is an in-line expansion to a single assembly instruction called, you guessed it, `CLI`). Once we know we won't be interrupted, we can fiddle with the DDR. `reg` is the DDR for this particular port. We OR it with the bit mask of interest, in other words, we set that bit. This selects the direction as output. Finally, we restore the original contents of the status register, enabling interrupts (assuming it had been set to begin with).

So, when originally we wrote

```
        pinMode( 8, OUTPUT );
```

The function decoded Arduino pin 8 as being bit 0 of port B (i.e., `PORTB`). It also determined that the corresponding DDR is `DDRB`. The bit 0 mask is 0x01 and it was ORed with the contents of `DDRB`, thus selecting the direction for that bit as output mode.

---

[16] These are found in avr/pgmspace.h and pins_arduino.h

The clause for `INPUT` mode is similar:

```
if (mode == INPUT)
{
        oldSREG = SREG;
        cli();
        *reg &= ~bit;
        *out &= ~bit;
        SREG = oldSREG;
}
```

Note that here the DDR is ANDed with the complement of the bit mask to clear the bit, thus placing it in input mode. This code also clears the same bit in the output register which disables the pull-up resistor at the external pin. In contrast, note that `INPUT_PULLUP` mode sets this bit.

Great stuff for sure, but what if you're not using an Arduino or you need to get this done quicker? If you don't have any interrupt code running you can safely twiddle directly with the port and DDR. Remember, we wanted to make bit 0 of port B ready for output. That means we need to set bit 0 of `DDRB`.

```
DDRB |= 0x01;
```

or use the macro

```
bitSet( DDRB, 0 );
```

Either would do the trick, the latter probably a little more clear and less error prone. And here's something very useful to remember: What if you need to set a bunch of pins or even an entire port to input or output mode? Using `pinMode()` you'd have to make individual calls for each pin. In contrast, if you needed to set the bottom six bits of port B to output mode[17] you could just do this:

```
DDRB |= 0x3f;       // set Uno pins 8 through 13 to OUTPUT mode
```

Does this mean that we should never use `pinMode()`? No! We have seen that this function ties in perfectly with the Arduino Uno board and is both more robust and more flexible. It's just that sometimes a bicycle will serve us better than a motor vehicle and it's good that we have the option.

---

[17] You might do this in order to write values out of the port in parallel fashion, several bits at a time, for example to feed a parallel input DAC.

22. Bits & Pieces: digitalWrite()

"Writing to a port" implies that we wish to control some external device. This might involve setting or clearing a single bit to turn on an LED or motor. A set of bits might also be required, for example, to send out ASCII code byte-by-byte or to write data words to an external digital to analog converter (DAC). It is important to remember that the microcontroller has a limited amount of sink/source current available per pin (40 mA for each pin of the ATmega 328P but no more than 200 mA total for the chip). Thus, it is possible to drive a handful of LEDs to 10 mA each with a direct connection consisting of a current limiting resistor and the LED, but not possible to turn on even a relatively small DC motor. Higher current (or voltage) loads will require some manner of driver or interface circuit. We will forgo that discussion and just focus on the code portion here.

As the output port pins can only be in either a high or low state, they are referred to as digital outputs. While it is possible to generate analog signals, either through pulse width modulation or additional external circuitry, the digital high/low nature of port pins is all there is. Generally speaking, most microcontrollers do not produce continuously variable analog voltages at their port pins. There are exceptions to this, though. For example, the Arduino Due[18] board utilizes the Atmel SAM3X8E ARM Cortex-M3 CPU which contains two internal 12 bit DACs.

Before writing, the port has to be set up for the proper direction. This means using either `pinMode()` or the associated data direction register, `DDRx`, to set the mode to output **before** we can consider writing data to an external device. If a single port bit is all that's required, `pinMode()` is very straightforward and robust. If several bits need to be controlled together, it may be easier to go directly to `DDRx`.

Just as there are two methods to set the output mode, the same is true for writing the data itself; one effective method for single bits and another for sets of bits.

To write a single bit, the `digitalWrite()` function is a good choice. Here is the on-line documentation for it, found at http://arduino.cc/en/Reference/DigitalWrite:

---

digitalWrite()

Description

Write a HIGH or a LOW value to a digital pin.

If the pin has been configured as an OUTPUT with **pinMode**(), its voltage will be set to the corresponding value: 5V (or 3.3V on 3.3V boards) for HIGH, 0V (ground) for LOW.

If the pin is configured as an INPUT, writing a HIGH value with digitalWrite() will enable an internal 20K pullup resistor (see the tutorial on digital pins). Writing LOW will disable the pullup. The pullup resistor is enough to light an LED dimly, so if LEDs appear to work, but very dimly, this is a likely cause. The remedy is to set the pin to an output with the pinMode() function.

---

[18] http://arduino.cc/en/Main/ArduinoBoardDue

NOTE: Digital pin 13 is harder to use as a digital input than the other digital pins because it has an LED and resistor attached to it that's soldered to the board on most boards. If you enable its internal 20k pull-up resistor, it will hang at around 1.7 V instead of the expected 5V because the onboard LED and series resistor pull the voltage level down, meaning it always returns LOW. If you must use pin 13 as a digital input, use an external pull down resistor.

Syntax

digitalWrite(pin, value)

Parameters

pin: the pin number

value: HIGH or LOW

Returns

Example

```
int ledPin = 13;                     // LED connected to digital pin 13

void setup()
{
  pinMode(ledPin, OUTPUT);      // sets the digital pin as output
}

void loop()
{
  digitalWrite(ledPin, HIGH);   // sets the LED on
  delay(1000);                  // waits for a second
  digitalWrite(ledPin, LOW);    // sets the LED off
  delay(1000);                  // waits for a second
}
```

Figure 22.1, digitalWrite docs

From the example code, this is pretty easy to use. Simply set the direction first, then write to the appropriate Arduino pin designator. Remember, the pin designators are the numbers written next to the headers on the Uno board, they're **not** the port bit numbers on the ATmega 328P! The reference note regarding Arduino pin 13 is worth re-reading. Pin 13 is hardwired to an on-board surface mount signaling LED located right next to said pin. This also means that the total source current available for pin 13 is somewhat reduced as LED current will also always be applied. Arduino pin 13 is PORTB bit 5 (sometimes written shorthand as PORTB.5).

So, just what does the digitalWrite() function do? Here is the code for digitalWrite(), slightly cleaned up for your viewing pleasure[19]:

---

[19] The original may be found in the file wiring_digital.c

```
    void digitalWrite( uint8_t pin, uint8_t val )
    {
        uint8_t timer, bit, port, oldSREG;
        volatile uint8_t *out;

        timer = digitalPinToTimer( pin );
        bit =   digitalPinToBitMask( pin );
        port =  digitalPinToPort( pin );

        if (port == NOT_A_PIN) return;

        if (timer != NOT_ON_TIMER)    turnOffPWM( timer );

        out = portOutputRegister( port );

        oldSREG = SREG;
        cli();

        if (val == LOW)    *out &= ~bit;
        else               *out |= bit;

        SREG = oldSREG;
    }
```

Figure 22.2, digitalWrite code

Let's take this apart, piece by piece, bit by bit. After the initial data declarations we see a group of function calls that serve to translate the Arduino pin designator into appropriate ATmega 328P ports, bits and timers. (The timer business we'll address shortly.) This section finishes with an error check. If the specified pin doesn't exist, the function bails out and does nothing.

```
        timer = digitalPinToTimer( pin );
        bit =   digitalPinToBitMask( pin );
        port =  digitalPinToPort( pin );

        if (port == NOT_A_PIN) return;
```

The AVR series of controllers, like most controllers, contain internal timers/counters. These allow the controller to precisely time events or produce pulse signals (specifically, pulse width modulation). The Arduino system pre-configures six of the available outputs with timers for use with the `analogWrite()` function. As not all pins have this ability, we need to translate the Arduino pin to an associated timer with the `digitalPinToTimer()` function. We will take a closer look at timers later, but for now it is only important to understand that any associated timer needs to be turned off before we can use our basic digital write function.

```
        if (timer != NOT_ON_TIMER)    turnOffPWM( timer );
```

The code to turn off the PWM functionality is little more than a `switch/case` statement. The `cbi()` call is used to clear a specific bit in a port, in this case the associated timer-counter control register (`TCCRx`). More info can be found at http://playground.arduino.cc/Main/AVR.

```
static void turnOffPWM( uint8_t timer )
{
        switch (timer)
        {
                case TIMER0A:    cbi(TCCR0A, COM0A1);    break;
                case TIMER0B:    cbi(TCCR0A, COM0B1);    break;
                case TIMER1A:    cbi(TCCR1A, COM1A1);    break;
                case TIMER1B:    cbi(TCCR1A, COM1B1);    break;
                case TIMER2A:    cbi(TCCR2A, COM2A1);    break;
                case TIMER2B:    cbi(TCCR2A, COM2B1);    break;
        }
}
```

After this, the specified Arduino port is translated to an output register, that is, `PORTx`.

```
out = portOutputRegister( port );
```

As was seen with `pinMode()`, the status register is saved, the global interrupt bit is cleared to disable all interrupts via the `cli()` call, the desired port (`PORTx`) is ANDed with the complement of the bit mask to clear it (i.e., set it low) or ORed with the bit mask to set it (i.e., set it high). The status register is then restored to its original state:

```
oldSREG = SREG;
cli();

if (val == LOW)    *out &= ~bit;
else               *out |= bit;

SREG = oldSREG;
```

That's pretty much it. Now, it you want to write a bunch of bits to a group of output port connections, you can simply look up the corresponding port for those Arduino pins (i.e., `PORTx`) and set or clear them directly. For example, if you want to clear bits 0, 1, and 4 of port B (i.e., 00010011 or 0x13), you could do the following (assuming no timer is active):

```
PORTB &= ~0x13;    // clear bits
```

or to set them:

```
PORTB |= 0x13;     // set bits
```

This would leave the other bits completely untouched. In contrast, suppose you want to set the bottom four bits (i.e., 0 through 3) to the binary pattern 0101. That's equivalent to 0x05. You could do the following:

```
PORTB = (PORTB & 0xf0)| 0x05;
```

The first chunk clears the bottom four bits and the trailing part sets the binary pattern. This would be preferable to clearing bits 1 and 3 and then setting bits 0 and 2 as that would cause two distinct sets of output voltage patterns at the external pins. Granted, the first set will exist for only a very short time but this can create problems in some applications.

Embedded Controllers   **115**

23. Bits & Pieces: delay()

or How to Waste Time

Sometimes our code needs to wait for things or time events. For example, we might want to turn an LED on for a few seconds and then turn it off. We've seen how to control an LED with `digitalWrite()` but how do we wait for a few seconds? One simple method is to create an empty loop. This is a loop that really does nothing but waste time. For example, if we know that simply incrementing, testing and branching in a loop takes a microsecond, we could write a function like this:

```
void CheesyDelay( unsigned long msec )
{
  volatile unsigned long i;
  unsigned long endmsec = msec * 1000;

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

Note that we specify the number of milliseconds we'd like to waste. Since each iteration of the loop takes one microsecond, we multiply by 1000 to achieve milliseconds. The `volatile` modifier is important here. This tells the compiler not to aggressively optimize the code for us because I could be changed by code running elsewhere (for example, in an interrupt). Otherwise, the compiler might figure out that it can achieve the same end result by ignoring the loop and doing a simple addition. The problem with this function is that the resulting delay is highly dependent on the microcontroller used and its clock frequency. If you just need a quick and dirty delay this will work fine, but a far more accurate delay is available with the `delay()` function and its sibling `delayMicroseconds()`, whose reference material is repeated below.

---

delay()[20]

Description

Pauses the program for the amount of time (in miliseconds) specified as parameter. (There are 1000 milliseconds in a second.)

Syntax

delay(ms)

Parameters

ms: the number of milliseconds to pause (*unsigned long*)

Returns

nothing

---

[20] http://arduino.cc/en/Reference/Delay

Example

```
int ledPin = 13;                    // LED connected to digital pin 13

void setup()
{
  pinMode(ledPin, OUTPUT);      // sets the digital pin as output
}

void loop()
{
  digitalWrite(ledPin, HIGH);   // sets the LED on
  delay(1000);                  // waits for a second
  digitalWrite(ledPin, LOW);    // sets the LED off
  delay(1000);                  // waits for a second
}
```

Caveat

While it is easy to create a blinking LED with the delay() function, and many sketches use short delays for such tasks as switch debouncing, the use of delay() in a sketch has significant drawbacks. No other reading of sensors, mathematical calculations, or pin manipulation can go on during the delay function, so in effect, it brings most other activity to a halt. For alternative approaches to controlling timing see the millis() function and the sketch sited below. More knowledgeable programmers usually avoid the use of delay() for timing of events longer than 10's of milliseconds unless the Arduino sketch is very simple.

Certain things *do* go on while the delay() function is controlling the Atmega chip however, because the delay function does not disable interrupts. Serial communication that appears at the RX pin is recorded, PWM (analogWrite) values and pin states are maintained, and interrupts will work as they should.

See also

- millis()

- micros()

- delayMicroseconds()

- Blink Without Delay example

delayMicroseconds()

Description

Pauses the program for the amount of time (in microseconds) specified as parameter. There are a thousand microseconds in a millisecond, and a million microseconds in a second.

Currently, the largest value that will produce an accurate delay is 16383. This could change in future Arduino releases. For delays longer than a few thousand microseconds, you should use delay() instead.

Syntax

delayMicroseconds(us)

Parameters

us: the number of microseconds to pause (*unsigned int*)

Returns

None

Caveats and Known Issues

This function works very accurately in the range 3 microseconds and up. We cannot assure that delayMicroseconds will perform precisely for smaller delay-times.

As of Arduino 0018, delayMicroseconds() no longer disables interrupts.

Figure 23.1, delay docs

These functions are also tied in with two other functions, micros() and millis(), which are repeated below:

millis()

Description

Returns the number of milliseconds since the Arduino board began running the current program. This number will overflow (go back to zero), after approximately 50 days.

Parameters

None

Returns

Number of milliseconds since the program started (*unsigned long*)

Tip:

Note that the parameter for millis is an unsigned long, errors may be generated if a programmer tries to do math with other datatypes such as ints

Returns the number of microseconds since the Arduino board began running the current program. This number will overflow (go back to zero), after approximately 70 minutes. On 16 MHz Arduino boards (e.g. Duemilanove and Nano), this function has a resolution of four microseconds (i.e. the value returned is always a multiple of four). On 8 MHz Arduino boards (e.g. the LilyPad), this function has a resolution of eight microseconds.

Parameters

None

Returns

Number of microseconds since the program started (*unsigned long*)

Figure 23.2, millis and micros docs

All of these functions rely on the Arduino system configuring the timers the moment the board is reset. One of these will be used to generate an interrupt when the counter overflows. The time to overflow will take a predetermined amount of time based on the clock speed. The interrupt will in turn update three global variables that will keep track of how long the program has been running.

First let's consider the initialization code along with some definitions and global variable declarations. Besides this timer, the init code also sets up the other timers for pulse width modulation duties (via the `analogWrite()` function). The code is reasonably well commented and is presented without further explanation, save for a reminder that `sbi()` is a macro that will reduce to a single assembly language instruction that sets a specific register bit.

```
#include "wiring_private.h"

// the prescaler is set so that timer0 ticks every 64 clock cycles, and the
// the overflow handler is called every 256 ticks.
#define MICROSECONDS_PER_TIMER0_OVERFLOW (clockCyclesToMicroseconds(64 * 256))

// the whole number of milliseconds per timer0 overflow
#define MILLIS_INC (MICROSECONDS_PER_TIMER0_OVERFLOW / 1000)

// the fractional number of milliseconds per timer0 overflow. we shift right
// by three to fit these numbers into a byte. (for the clock speeds we care
// about - 8 and 16 MHz - this doesn't lose precision.)
#define FRACT_INC ((MICROSECONDS_PER_TIMER0_OVERFLOW % 1000) >> 3)
#define FRACT_MAX (1000 >> 3)

volatile unsigned long timer0_overflow_count = 0;
volatile unsigned long timer0_millis = 0;
static unsigned char timer0_fract = 0;
```

```
void init()
{
    // this needs to be called before setup() or some functions won't
    // work there
    sei();

    // set timer 0 prescale factor to 64
    sbi(TCCR0B, CS01);
    sbi(TCCR0B, CS00);

    // enable timer 0 overflow interrupt
    sbi(TIMSK0, TOIE0);

    // timers 1 and 2 are used for phase-correct hardware pwm
    // this is better for motors as it ensures an even waveform
    // note, however, that fast pwm mode can achieve a frequency of up
    // 8 MHz (with a 16 MHz clock) at 50% duty cycle

    TCCR1B = 0;

    // set timer 1 prescale factor to 64
    sbi(TCCR1B, CS11);
    sbi(TCCR1B, CS10);

    // put timer 1 in 8-bit phase correct pwm mode
    sbi(TCCR1A, WGM10);

    // set timer 2 prescale factor to 64
    sbi(TCCR2B, CS22);

    // configure timer 2 for phase correct pwm (8-bit)
    sbi(TCCR2A, WGM20);


    // set a2d prescale factor to 128
    // 16 MHz / 128 = 125 KHz, inside the desired 50-200 KHz range.
    // XXX: this will not work properly for other clock speeds, and
    // this code should use F_CPU to determine the prescale factor.
    sbi(ADCSRA, ADPS2);
    sbi(ADCSRA, ADPS1);
    sbi(ADCSRA, ADPS0);

    // enable a2d conversions
    sbi(ADCSRA, ADEN);

    // the bootloader connects pins 0 and 1 to the USART; disconnect
    // them here so they can be used as normal digital i/o;
    // they will be reconnected in Serial.begin()
    UCSR0B = 0;
}
```

Figure 23.3, timer setup code

Now let's take a look at the interrupt service routine. Each time the counter overflows (i.e. the 8 bit counter tries to increment 255 and wraps back to 0) it generates an interrupt which calls this function. Basically, all it does is increment the global variables declared earlier.

```
SIGNAL( TIMER0_OVF_vect )
{
      // copy these to local variables so they can be stored in
      // registers (volatile vars are read from memory on every access)
      unsigned long m = timer0_millis;
      unsigned char f = timer0_fract;

      m += MILLIS_INC;
      f += FRACT_INC;

      if (f >= FRACT_MAX)
      {
            f -= FRACT_MAX;
            m += 1;
      }

      timer0_fract = f;
      timer0_millis = m;
      timer0_overflow_count++;
}
```

Figure 23.4, timer interrupt code

As you might now guess, all the millis() and micros() functions do is access these global variables and return their values. Because an interrupt can occur during this process, the value of the status register (SREG) is copied, the status register's global interrupt enable bit is cleared with the cli() call, the access performed (plus a little extra calculation for micros()) and the status register returned to its prior state. The retrieved value is then returned to the caller.

```
unsigned long millis()
{
      unsigned long m;
      uint8_t oldSREG = SREG;

      // disable interrupts while we read timer0_millis or we might get
      // an inconsistent value (e.g. in the middle of a write to
      // timer0_millis)

      cli();
      m = timer0_millis;
      SREG = oldSREG;

      return m;
}
```

```
unsigned long micros()
{
     unsigned long m;
     uint8_t t, oldSREG = SREG;

     cli();
     m = timer0_overflow_count;
     t = TCNT0;
     if ((TIFR0 & _BV(TOV0)) && (t < 255))     m++;
     SREG = oldSREG;

     return ((m << 8) + t) * (64 / clockCyclesPerMicrosecond());
}
```

Figure 23.5, millis and micros code

So the `delay()` function itself is pretty straightforward. It simply retrieves the current time since reset and then goes into a "busy wait" loop, constantly checking and rechecking the time until the difference between the two reaches the requested value.

```
void delay(unsigned long ms)
{
     uint16_t start;

     start = (uint16_t)micros();

     while (ms > 0)
     {
          if (((uint16_t)micros() - start) >= 1000)
          {
               ms--;
               start += 1000;
          }
     }
}
```

Figure 23.6, delay code

In a way, this is just a slightly more sophisticated version of our initial cheesy delay function. It is more precise because it uses the accurate internal counters which are operating from a known clock frequency. The microseconds version of the delay is a little trickier, especially for short delays. This also does a busy wait but does so using in-line assembly code. Even with this, the delays are not particularly accurate for periods of only a few microseconds. In the in-line comments are instructive:

```
/* Delay in microseconds. Assumes 8 or 16 MHz clock. */

void delayMicroseconds(unsigned int us)
{
      // for the 16 MHz clock on most Arduino boards
      // for a one-microsecond delay, simply return.  the overhead
      // of the function call yields a delay of approximately 1 1/8 us.
      if (--us == 0)
            return;

      // the following loop takes a quarter of a microsecond (4 cycles)
      // per iteration, so execute it four times for each microsecond of
      // delay requested.
      us <<= 2;

      // account for the time taken in the preceding commands.
      us -= 2;

      // busy wait
      __asm__ __volatile__ (
            "1: sbiw %0,1" "\n\t" // 2 cycles
            "brne 1b" : "=w" (us) : "0" (us) // 2 cycles
      );
}
```

Figure 23.7, delayMicroseconds code

The major problem with using delay() is noted in its on-line documentation, namely, that during a busy wait loop no other work can be done. The controller is effectively "spinning its wheels". A more effective way to delay is to make direct use of the millis() function. The basic idea is to check the time using millis() and then do what you need to do inside a loop, checking the elapsed time on each iteration. Here is a snippet of example code.

```
unsigned long currentMillis, previousMillis, intervalToWait;

// intervalToWait could be a passed variable, global or define

// initialize to current time
previousMillis = millis();
currentMillis = millis();

while ( currentMillis - previousMillis < intervalToWait )
{
      // do whatever you need to do here

      currentMillis = millis();
}
```

In essence you've built your own "kind of" busy wait loop but with requisite code inside.

24. Bits & Pieces: digitalRead()

24.1 Introduction

The discussion that follows deals strictly with two-state high/low logic level sensing. For continuously variable analog signals see the Bits & Pieces entry covering `analogRead()`. Through the `pinMode()` function, or by directly accessing the appropriate data direction register bits (`DDRx`), the general purpose IO connections can be configured to read the state of external switches or logic levels. With the Arduino Uno, 5 volts represents a logic high while 0 volts represents a logic low. An added capacity of the ATmega 328P on the Uno is the ability to include an optional internal pull-up resistor at the input pin. This allows connection of a simple passive short-to-ground switch (i.e. the input pin floats high when the switch is open and goes low when the switch is engaged).

To read individual pin inputs the Arduino system offers the `digitalRead()` function. Multiple pins can be read simultaneously by directly accessing the appropriate register, which we will examine afterward. Below is a copy of the online documentation for the `digitalRead()` function:

digitalRead()[21]

Description

Reads the value from a specified digital pin, either <u>HIGH</u> or <u>LOW</u>.

Syntax

digitalRead(pin)

Parameters

pin: the number of the digital pin you want to read (*int*)

Returns

<u>HIGH</u> or <u>LOW</u>

Example
```
int ledPin = 13; // LED connected to digital pin 13
int inPin = 7;   // pushbutton connected to digital pin 7
int val = 0;     // variable to store the read value

void setup()
{
  pinMode(ledPin, OUTPUT);    // sets the digital pin 13 as output
  pinMode(inPin, INPUT);      // sets the digital pin 7 as input
}
void loop()
{
  val = digitalRead(inPin);   // read the input pin
  digitalWrite(ledPin, val);  // sets the LED to the button's value
}
```

---

[21] http://arduino.cc/en/Reference/DigitalRead

Sets pin 13 to the same value as the pin 7, which is an input.

Note

If the pin isn't connected to anything, digitalRead() can return either HIGH or LOW (and this can change randomly).

The analog input pins can be used as digital pins, referred to as A0, A1, etc.

See also

- [pinMode](){}()
- [digitalWrite]()()
- [Tutorial: Digital Pins]()

Figure 24.1, digitalRead docs

A slightly cleaned-up version of the source code follows (found in the file wiring_digital.c):

```
int digitalRead( uint8_t pin )
{
     uint8_t timer, bit, port;

     timer = digitalPinToTimer( pin );
     bit =   digitalPinToBitMask( pin );
     port =  digitalPinToPort( pin );

     if (port == NOT_A_PIN)              return LOW;

     if (timer != NOT_ON_TIMER)          turnOffPWM(timer);

     if (*portInputRegister(port) & bit) return HIGH;

     return LOW;
}

static void turnOffPWM( uint8_t timer )
{
     switch ( timer )
     {
          case TIMER0A:   cbi( TCCR0A, COM0A1 );    break;
          case TIMER0B:   cbi( TCCR0A, COM0B1 );    break;

          // and so forth for all available timers, not shown
     }
}
```

Figure 24.2, digitalRead code

The first three lines convert the Arduino pin designator to the appropriate ATmega 328P port, bit number and timer. If the port is invalid, the function exits.

```
timer = digitalPinToTimer( pin );
bit =   digitalPinToBitMask( pin );
port =  digitalPinToPort( pin );

if (port == NOT_A_PIN)              return LOW;
```

The timer is important because the Arduino system preconfigures the Uno's three on-board timers for use with the `analogWrite()` function through a pulse width modulation scheme. This affects six of the possible pins. For proper operation of the digital read, these timers need to be turned off. We saw this same bit of code inside the `digitalWrite()` function.

```
if (timer != NOT_ON_TIMER)         turnOffPWM(timer);
```

At this point the contents of the input register are read (the direct name of the input register is `PINx`) and then ANDed with the requested bit. This removes all of the other bits so we can return either a simple high or low.

```
if (*portInputRegister(port) & bit) return HIGH;

return LOW;
```

The function used to turn off the pulse width modulation timers is little more than a `switch/case` statement. If the specified Arduino pin is hooked up to a timer internally, that timer is found in the switch statement and a `cbi()` call is executed on the appropriate timer-counter control register. The `cbi()` function translates to a single assembly language instruction to clear the specified bit in the control register, thus turning off that timer.

```
static void turnOffPWM( uint8_t timer )
{
    switch (timer)
    {
        case TIMER0A:     cbi(TCCR0A, COM0A1);     break;
        case TIMER0B:     cbi(TCCR0A, COM0B1);     break;
        case TIMER1A:     cbi(TCCR1A, COM1A1);     break;
        case TIMER1B:     cbi(TCCR1A, COM1B1);     break;
        case TIMER2A:     cbi(TCCR2A, COM2A1);     break;
        case TIMER2B:     cbi(TCCR2A, COM2B1);     break;
    }
}
```

In some applications, several bits need to be read at once, for example when reading parallel data. This can be performed through a direct access of the appropriate `PINx` register. `PINx` is rather like the fraternal twin of the `PORTx` register. While `PORTx` is used to write digital data to an external connection, `PINx` is where you read digital data from an external connection. Just as there are four output port registers, A through D, there are four input pin registers, A through D. Not all microcontrollers are configured this way. Some of them use the same register for both reading and writing (the function being controlled by the associated data direction register).

Here is how to directly access a single bit on a non-timer connected pin. First, clear the desired data direction register bit to activate input mode. Second, if desired, set the same bit in the associated port

register to enable the optional pull-up resistor. If you don't want the pull-up, leave that bit clear. Finally, read the desired bit in the pin register and AND it with the bit mask to remove the other bits. For example, to read bit 4 (0x10 or 00010000 in binary) on port B:

```
DDRB &= (~0x10);        // activate input mode
PORTB |= 0x10;          // enable pull-up or use the bitSet macro
                        // bitSet( PORTB, 4 );
value = PINB & 0x10;    // retrieve data
```

It is only minor work to alter this for multiple bits. To read both bits 0 and 4 but without the pull-up resistors (bit pattern 00010001 or 0x11):

```
DDRB &= (~0x11);        // activate input mode
PORTB &= (~0x11);       // disable pull-up
value = PINB & 0x11;    // retrieve data bits
```

or if you want the bits separately:

```
value0 = PINB & 0x01;   // retrieve data bit 0
value4 = PINB & 0x10;   // retrieve data bit 4
```

24.2 A Practical Example: Round-Robin Switch

Let's take a look at how we might implement a *round-robin* switch. A round-robin switch works in a circular manner; stepping through a sequence of settings and returning back to the start once the sequence is completed. For our example, we shall consider a single momentary contact pushbutton that cycles through a series of fan speeds: off, low, medium and high. The system will start in the off state and each successive press of the button will advance the state from low through high. Once high is reached, a further press will cycle the state back to off and the process continues in like manner. Each speed setting will be indicated via its own LED and only one LED will be on at any given time. To keep the code clean, will only examine the pushbutton and LED portion of the code and not consider how we might control the speed of a fan. Further, we shall assume that an appropriate hardware debounce circuit is incorporated with the pushbutton (e.g., a 74HC14 Schmitt with RC network) and that depressing the button will produce a logical high at our Arduino input pin. Also, the LED drivers are assumed to be active high (i.e., a high on an output pin of the Arduino will light an LED).

The first thing to remember is that a typical microcontroller is capable of checking the state of an input pin in a fraction of a microsecond. Consequently, we cannot simply check the pin to see if it is a logical high, and if so, cycle through to the next fan speed. In the fraction of a second it would take for a human to press and release the button, the microcontroller could loop through the pin checking code thousands of times, cycling the fan speed for each loop's check. The final fan speed would be anyone's guess. Instead, we need to look for the low to high transition as this happens only once with each button press. Indeed, this also points up the need for switch debouncing. Without it, a single press could result in over a dozen low to high edges due to switch contact chatter resulting in a random fan setting. One way to perform the positive going edge detection is to use two variables, one for the current state of the switch and a second for the prior state of the switch (i.e., the state that was measured immediately prior to the current state). If the current state is high (on or pressed) and the prior state is low (off or not pressed) then we know we have detected a positive going edge and we can perform our processing. Note that on the next iteration of the loop, the current state will be high but the prior state will now also be high (i.e., the former current state from the prior loop iteration) and thus no processing is performed. At this point it would not matter how long the user kept their finger on the button as all current and prior states will be high. When the user

finally releases the button the current state will be low with a prior state of high. This is the negative going edge and, again, we have no need to process it. The next loop iteration will produce current and prior states of low with no need to process. Eventually, when the user presses the button again we'll see a current state of high with a prior state of low and we'll process this edge.

We'll need three variables for the states, namely the current and prior pushbutton states and another for the LEDs (which indicate fan speed). The pushbuttons can be thought of as Boolean variables (0/1) while the LED variable will range from 0 through 3, 0 being "off" and 3 being "high". For simplicity, we'll use globals for these, unsigned chars will do nicely. The four LEDs will be positioned off of Arduino Uno pins 8 through 11 (PORTB.0:3) and the pushbutton will be connected to pin 12 (PORTB.4). The setup() function will have to set the proper directions for these port bits. This would require five separate calls to pinMode() but only two simple operations if we operate directly. Here is the beginning portion:

```
// declare the state variables and initialize at 0 for "off"
unsigned char currentstate=0;
unsigned char priorstate=0;
unsigned char ledstate=0;

// define some values for the LED/fan state. These are conveniently
// chosen to be the FAN/LED bit positions in PORTB
#define FAN_OFF   0
#define FAN_LO    1
#define FAN_MED   2
#define FAN_HIGH  3

// declare bit masks for the four LED bits and pushbutton
#define LED_OFF  0x01
#define LED_LOW  0x02
#define LED_MED  0x04
#define LED_HIGH 0x08
#define PBUTTON  0x10

// the LED_XXX masks could also be defined like so:
// #define LED_OFF (1<<FAN_OFF)

// a convenience
#define LED_ALL  (LED_OFF|LED_LOW|LED_MED|LED_HIGH)


setup()
{
     // set for output
     DDRB |= LED_ALL;
     // set for input
     DDRB &= ~PBUTTON;

     // light up the "off" LED
     PORTB |= LED_OFF;
     // by default, outputs are 0 but if for some reason the other
     // LEDs could be on, be sure to set them to off before continuing
     // PORTB &= ~(LED_LOW|LED_MED|LED_HIGH);
}
```

The looping code needs to check for the proper current state/prior state pair. If it is not found, there is nothing to do. If it is found, we need to turn off the existing LED and then check the current LED/fan state, increment to the next state and light that LED. There are a couple ways to do this. The first is perhaps an obvious solution using a switch/case construct. The second method is a bit more compact (pun intended).

```
loop()
{
        // get current button state
        currentstate = PINB & PBUTTON;

        // do we have a positive going edge?
        if( currentstate && !priorstate )
        {
                switch( ledstate )
                {
                        case FAN_OFF:
                                PORTB &= ~LED_OFF;   // turn off old LED
                                PORTB |= LED_LOW;    // turn on new LED
                                ledstate = FAN_LOW;  // increment speed
                                break;

                        case FAN_LOW:
                                PORTB &= ~LED_LOW;
                                PORTB |= LED_MED;
                                ledstate = FAN_MED;
                                break;

                        case FAN_MED:
                                PORTB &= ~LED_MED;
                                PORTB |= LED_HIGH;
                                ledstate = FAN_HIGH;
                                break;

                        case FAN_HIGH:
                                PORTB &= ~LED_HIGH;
                                PORTB |= LED_OFF;
                                ledstate = FAN_OFF;
                                break;
                }
        }

        // update state for next loop iteration
        priorstate = currentstate;
}
```

The alternate version can be found on the next page. Note the code savings. Instead of checking for each fan state, we turn off all LEDs and increment the fan state. If the state would increment past the maximum then we make sure to set it to the off state. This new state tells us which LED to light. This version works with little code because of the way in which we chose our bit positions. This is not always possible (or even desirable) but can be a handy technique from time to time.

```
loop()
{
      // get current button state
      currentstate = PINB & PBUTTON;

      // do we have a positive going edge?
      if( currentstate && !priorstate )
      {
            // increment to new state
            if( ledstate < FAN_HIGH )
                  ledstate++;
            else
                  ledstate = FAN_OFF;

            // lazy: turn off all LEDs
            PORTB &= ~LED_ALL;
            // turn on new LED
            PORTB |= (1<<ledstate);
      }

      // update state for next loop iteration
      priorstate = currentstate;
}
```

24.3 Exercise

In closing, here's an interesting question. If the round-robin switch had numerous states, say a dozen or so, a single increment button might prove a little frustrating to the user. For example, if they accidentally go past the desired setting then they're forced to go all the way around again to effectively "back up". To alleviate this we could add a decrement pushbutton alongside our existing increment pushbutton. How would the code example presented above need to be altered to respond to a two button configuration? If at first this appears to be too daunting of a challenge, then break it into two parts. Initially, consider how the preceding code would need to be altered in order to change the operation of the fan speed increment button into a fan speed decrement button. That is, instead of button pushes producing the sequence *off-low-med-high-off* etc., implement the button to produce *off-high-med-low-off* etc. Once this is completed, determine how to combine this new implementation with the existing example code.

http://xkcd.com/156/
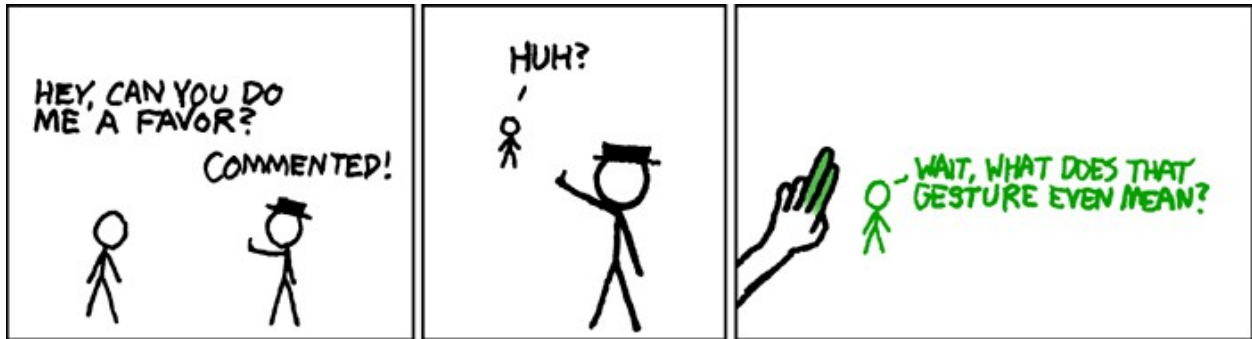
25. Bits & Pieces: Analog Input Circuitry

25.1 Introduction

As useful as digital input/output ports are, there is also a need for continuously variable or analog ports. Controllers are, of course, inherently digital devices but that doesn't mean that analog signals are out of bounds. There is considerable variation in what is available on any given microcontroller, though. On some units, analog to digital and digital to analog circuitry must be added to the controller system as peripheral devices. This is particularly true for less expensive controllers or for more specialized applications that require extreme performance (for example, a high definition digital audio recorder or playback device). Most general purpose controllers have an analog to digital converter (ADC) and some also include a digital to analog converter (DAC). The resolution and speed of these converters can vary quite a bit from controller to controller. In this chapter we shall focus on the ADC system found in the ATmega series. While our discussion will be quite specific, the concepts presented apply to many other controllers made by other manufacturers. Only the performance and operational details may differ.

A typical use of an ADC is to capture the value of an external sensor or user input device at a particular point in time. For instance, we may wish to measure the output voltage from a temperature or light sensor. Regarding a user interface device, one possibility is to measure the voltage developed across a potentiometer (the outer terminals connected to power and ground, the wiper attached to the analog input port). As the user rotates the pot, the voltage shifts. This voltage could represent all manner of variables from the loudness setting for music playback to the speed of a motor. These sorts of applications can be considered "snapshot" or single conversion uses. That is, we don't spend all of our time continuously converting the port voltage to digital values the way we would with, say, a waveform capture device. In the case of something like a temperature sensor, we simply perform conversions as we need them. For user interface devices, as long as we can obtain values at a sufficiently fast rate, the control will appear continuous to the user. In some applications this may require no more than a dozen or so conversions per second. This is compared to tens of thousands of conversions per second for something like digitizing audio signals.

Of course, we may need to have several user interface devices. Consider something as common as a stereo receiver. Typically we would find knobs for volume, balance, bass, treble and so forth. It would be very impractical to have a single knob that controls everything and a series of buttons alongside it to indicate the current function of the knob. Alternately, in an industrial setting we might need to monitor temperatures at several locations along with other environmental parameters. While we could use one ADC for each of these, it makes far more sense to use a single ADC and multiplex it to several external pins. Think of these as multiple input channels. We select the channel (sensor) we want, make the measurement and continue on.

This is not to say that we only use this "snapshot" mode. There are other applications that require continuous monitoring of the external signal and other techniques of obtaining the measurement. Consequently, ADC systems can be configured for a single on-demand conversion as above, they can be free-running or the conversion can be triggered by some other event. Because of these demands, the circuitry and programming interface for analog inputs tends to be much more involved (and much more flexible) than we've seen for digital inputs. For example, ADC systems often have several registers and bits devoted to specifying parameters such as the sample acquisition mode, the conversion speed, data justification, pin muxing and so forth.

25.2 ATmega ADC system

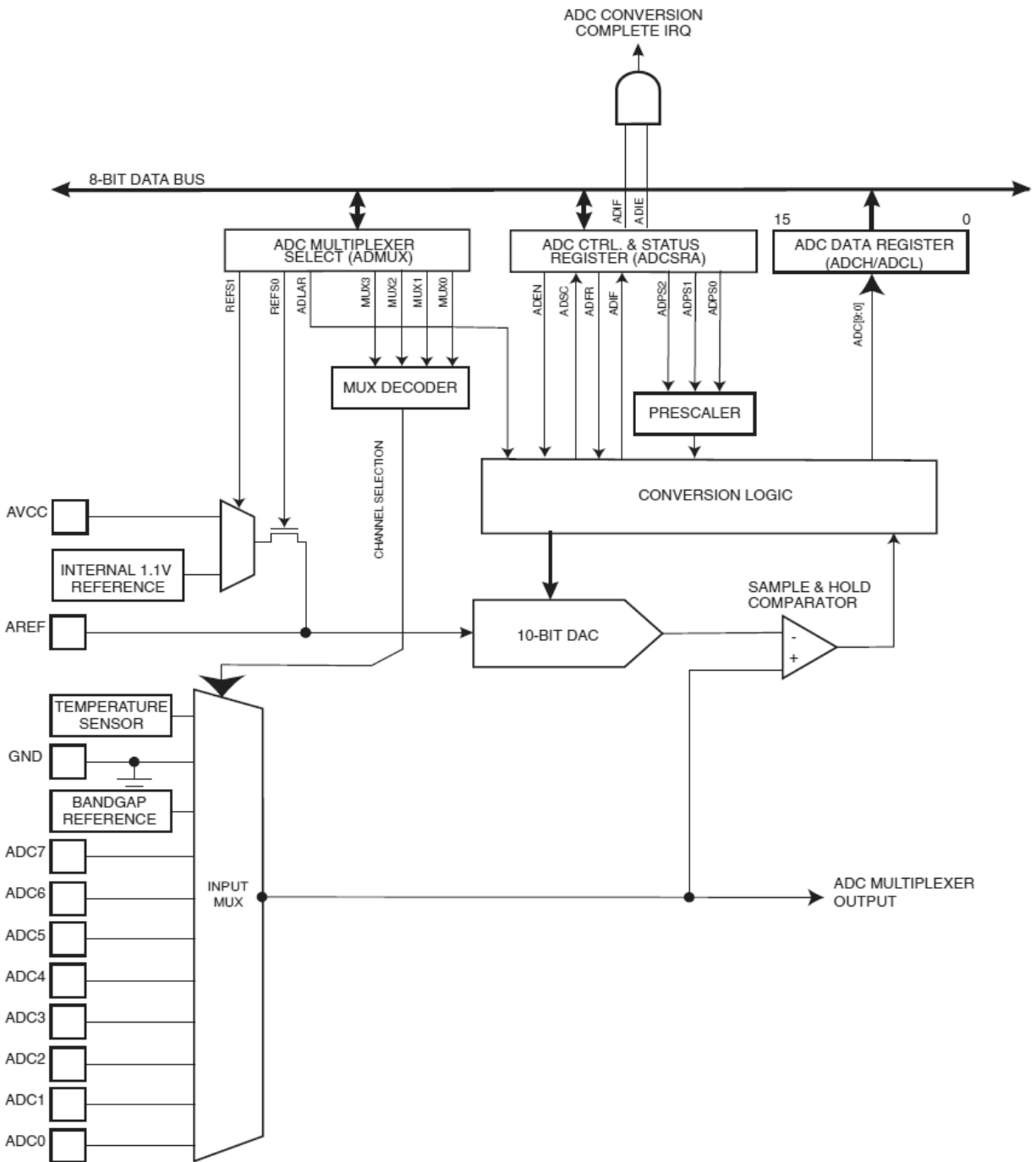A block diagram of the ATmega ADC system is found in Figure 25.1.

Figure 25.1, ADC circuitry (Atmel 2014)

The heart of the system is a single successive approximation DAC with 10 bit resolution. Maximum conversion speed at full resolution is about 15 k samples per second. Clearly this is insufficient for applications such as CD quality audio or video but is more than fast enough for most sensor and user interface monitoring. The converter performance specifications include 0.5 LSB integral non-linearity error and ±2 LSB absolute accuracy. An on-board reference is available although an external reference may also be used. The converter includes a sample-and-hold system to stabilize the measurement for rapidly changing input signals. There is an eight channel mux although only six of these channels are used on some packages (the package used on the standard configuration of the Arduino Uno board uses six). The input circuitry is optimized for sources that have an output impedance of 10 KΩ or less. It is also very important to recognize that the system is unipolar and will only respond to positive voltages. If the signal to be measured is bipolar, some manner of DC level shifting will be required.

There are several registers associated with the ADC system. These are shown in Figure 25.2.

Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ADCSRA	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
ADCSRB	–	ACME	–	–	–	ADTS2	ADTS1	ADTS0
ADMUX	REFS1	REFS0	ADLAR	–	MUX3	MUX2	MUX1	MUX0
ADCH	ADC9/-	ADC8/-	ADC7/-	ADC6/-	ADC5/-	ADC4/-	ADC3/9	ADC2/8
ADCL	ADC1/7	ADC0/6	-/ADC5	-/ADC4	-/ADC3	-/ADC2	-/ADC1	-/ADC0

Figure 26.2, ADC registers

ADCSRA and ADCSRB are the Analog-Digital Control and Status Registers. Here are their bit descriptions:

- ADEN: ENable. Set to one to enable to Analog-Digital system.
- ADSC : Start Conversion. Set to one to start a conversion. When the conversion is complete, the hardware will reset this bit to zero. Writing zero to this bit will have no effect.
- ADATE: Auto Trigger Enable. Set to one for auto trigger mode. A rising edge on the selected trigger signal will start a conversion. The trigger source is selected via the ADTS bits in the ADCSRB register (below).
- ADIF: Interrupt Flag. This is used with interrupt mode. This bit is set when a conversion is complete and the data registers are updated. This bit is cleared by the hardware when executing the interrupt handler routine. Use with ADIE.
- ADIE: Interrupt Enabled. Set to one to enable the ADC conversion interrupt. Use with ADIF.
- ADPS2-0: Prescaler bits. These bits are used to divide down the system clock to the ADC clock. 000 and 001 yield division by two and increasing values double the clock divide with 111 yielding divide by 128.
- ACME: Analog Comparator Multiplexer Enabled. Setting these bits allows any of the eight multiplexed ADC pins to replace the negative input to the analog comparator.
- ADTS2-0: Trigger Source. If ADATE is set to one, these bits determine the trigger source. 000 is free running mode, 001 triggers from the analog comparator, 010 uses External Interrupt Request 0, and the remaining values set various timer/counter sources.

`ADMUX` is the Analog-Digital MUltipleXer selection register. It is also used for reference selection and data justification. The bit descriptions follow:

- `REFS1-0`: REFerence voltages. 00 yields the `AREF` pin, 01 yields `AVCC`, 11 yields internal 1.1 volt reference and 10 is reserved.
- `ADLAR`: Left Adjust Result. Setting this bit will left justify the output bits in the `ADCH`/`ADCL` registers. Setting to zero right justifies.
- `MUX3-0`: MUltipleXer. These bits select the input pin. 000 selects `ADC0`, 001 selects `ADC1` and so on to 1000 selecting `ADC8` (the temperature sensor). 1110 selects the 1.1 volt reference and 1111 selects ground. All other values are reserved.

Finally we have `ADCH` and `ADCL`, the ADC High and Low output registers. There are two registers because this is a 10 bit converter but the registers only hold eight bits each. There are two modes of operation in conjunction with the `ADLAR` bit in the `ADMUX` register. If `ADLAR` is set then the top eight bits will be placed into `ADCH` and the bottom two bits will be placed at the top of `ADCL`. If `ADLAR` is cleared then the bottom eight bits will be found in `ADCL` and the top two bits will be found at the bottom of `ADCH`. Just think of the two registers as a single 16 bit register with the bits pushed to the left or right as shown in Figure 26.3.

ADLAR	ADCH								ADCL							
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	ADC 9	ADC 8	ADC 7	ADC 6	ADC 5	ADC 4	ADC 3	ADC 2	ADC 1	ADC 0	–	–	–	–	–	–
0	–	–	–	–	–	–	ADC 9	ADC 8	ADC 7	ADC 6	ADC 5	ADC 4	ADC 3	ADC 2	ADC 1	ADC 0

Figure 26.3, data justification

If only eight bits of resolution are needed, it is possible to left justify and read just `ADCH`, ignoring `ADCL`.

Single conversion operation is fairly straightforward. First, set up your initialization section. This begins with setting `ADEN` to enable the system. Then decide on the proper prescale value. The chip specifications indicate that the successive approximation circuitry will perform best with an ADC clock in the range of 50 kHz to 200 kHz. A typical conversion will take 13 ADC clock cycles (the first conversion will take 25 due to initialization overhead). Once this is determined, set `ADLAR` for data justification and `REFS` for the reference voltage. Then set the `ADMUX` bits to determine the input channel. Finally, set `ADSC` to start a conversion. You can wait on this bit or come back to it later. When the conversion is complete the hardware will reset it to zero. At that point read the value from `ADCL` and combine (shift and OR) with the value read from `ADCH` into a 16 bit integer. Note that you *must* read `ADCL` before reading `ADCH`. Reading `ADCL` locks the ADC registers preventing the hardware from overwriting the registers on a subsequent conversion. Reading `ADCH` unlocks them.

We will take a closer look at the software process in the next chapter.

26. Bits & Pieces: analogRead()

26.1 Introduction

Just as we would like to read the state of simple on/off switches, we also need to read continuously variable (i.e. analog) data. Usually this means the output voltage caused by some form of sensor such as a temperature sensor, force sensor, light sensor, etc. Very often simple passive devices such as CdS cells or FSRs are placed into a resistive bridge network, the output voltage of which will shift with changes in the environment. An even simpler application would be the use of a potentiometer hooked up to a fixed voltage supply. The position of the pot would be controlled by the user and could represent a setting of almost any conceivable parameter such as volume, brightness, time delay, frequency, etc. In order to read analog quantities, the ATmega 328P contains a single 10 bit analog-to-digital converter multiplexed across six input channels. On the Arduino Uno board, the inputs to these ADCs are found at the pins labeled A0 through A5. The Arduino development environment contains two useful functions to access theses, namely `analogRead()` and `analogReference()`. The on-line function descriptions are repeated below:

analogRead()[22]

Description

Reads the value from the specified analog pin. The Arduino board contains a 6 channel (8 channels on the Mini and Nano, 16 on the Mega), 10-bit analog to digital converter. This means that it will map input voltages between 0 and 5 volts into integer values between 0 and 1023. This yields a resolution between readings of: 5 volts / 1024 units or, 0.0049 volts (4.9 mV) per unit. The input range and resolution can be changed using [analogReference]().

It takes about 100 microseconds (0.0001 s) to read an analog input, so the maximum reading rate is about 10,000 times a second.

Syntax

analogRead(pin)

Parameters

pin: the number of the analog input pin to read from (0 to 5 on most boards, 0 to 7 on the Mini and Nano, 0 to 15 on the Mega)

Returns

int (0 to 1023)

Note

If the analog input pin is not connected to anything, the value returned by analogRead() will fluctuate based on a number of factors (e.g. the values of the other analog inputs, how close your hand is to the board, etc.).

---

[22] http://arduino.cc/en/Reference/AnalogRead

See also

- analogReference()

- Tutorial: Analog Input Pins

analogReference()[23]

Description

Configures the reference voltage used for analog input (i.e. the value used as the top of the input range). The options are:

- DEFAULT: the default analog reference of 5 volts (on 5V Arduino boards) or 3.3 volts (on 3.3V Arduino boards)

- INTERNAL: an built-in reference, equal to 1.1 volts on the ATmega168 or ATmega328 and 2.56 volts on the ATmega8 (*not available on the Arduino Mega*)

- INTERNAL1V1: a built-in 1.1V reference (*Arduino Mega only*)

- INTERNAL2V56: a built-in 2.56V reference (*Arduino Mega only*)

- EXTERNAL: the voltage applied to the AREF pin (**0 to 5V only**) is used as the reference.

Syntax

analogReference(type)

Parameters

type: which type of reference to use (DEFAULT, INTERNAL, INTERNAL1V1, INTERNAL2V56, or EXTERNAL).

Returns

None.

Note

After changing the analog reference, the first few readings from analogRead() may not be accurate.

Warning

Don't use anything less than 0V or more than 5V for external reference voltage on the AREF pin! If you're using an external reference on the AREF pin, you must set the analog reference to EXTERNAL before calling analogRead(). Otherwise, you will short together the active reference voltage (internally generated) and the AREF pin, possibly damaging the microcontroller on your Arduino board.

Alternatively, you can connect the external reference voltage to the AREF pin through a 5K resistor,

---

[23] http://arduino.cc/en/Reference/AnalogReference

allowing you to switch between external and internal reference voltages. Note that the resistor will alter the voltage that gets used as the reference because there is an internal 32K resistor on the AREF pin. The two act as a voltage divider, so, for example, 2.5V applied through the resistor will yield 2.5 * 32 / (32+5) = ~2.2V at the AREF pin.

See also

- [Description of the analog input pins](#)

- [analogRead](#)()

Figure 26.1, analogRead and analogReference docs

The `analogRead()` function is best used as a single conversion "snapshot". This is good for simple user interface and basic sensor applications. Generally speaking, `analogReference()` is called once during the setup and initialization phase of the program. Also, unless there is a compelling reason to do otherwise, the default mode is the best place to start. This will yield a 5 volt peak to peak input range with a bit resolution of just under 4.9 millivolts (5/1024). **It is important to note that this range runs from 0 volts to 5 volts, not −2.5 volts to +2.5 volts.** Depending on the amplitude and frequency range of the sensor signal, some input processing circuitry may be required to apply a DC offset, amplify or reduce signal strength, filter frequency extremes and so forth.

The code for `analogReference()` is about as simple as it gets. It just sets (and hides) a global variable which will be accessed by the `analogRead()` function:

```
uint8_t analog_reference = DEFAULT;

void analogReference(uint8_t mode)
{
        analog_reference = mode;
}
```

The `analogRead()` function is a bit more interesting (actually, 10 bits more interesting). From the preceding chapter we saw that there several registers associated with the ADC system. For single conversion operation the important registers include `ADCSRA` (ADC control and status register A) and `ADMUX` (ADC multiplexer selection register). The register bits are repeated below for convenience:

Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ADCSRA	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
ADMUX	REFS1	REFS0	ADLAR	–	MUX3	MUX2	MUX1	MUX0

Figure 26.2, ADC bits

Also recall that the `ADCH` and `ADCL` registers contain the high and low bytes of the result, respectively. For `ADCSRA`, setting `ADEN` enables the ADC. Setting `ADSC` starts a conversion. It will stay at one until the

conversion is complete, the hardware resetting it to zero. ADIF and ADIE are used with an interrupt-based mode not discussed here. ADATE stands for AD Auto Trigger Enable which allows triggering of the ADC from an external signal (again, not discussed here). ADPSx are pre-scaler bits which set the conversion speed. See the previous chapter for details.

For ADMUX, REFSx sets the reference voltage source where 00 = use AREF pin, 01 = use VCC, 10 = reserved, 11 = use internal 1.1 volt reference. Setting ADLAR left-adjusts the 10 bit word within the ADCH and ADCL result registers (i.e. it left-justifies, leaving the bottom six bits unused). Clearing this bit leaves the result right justified (i.e. top six bits are unused). The four MUXx bits select which input channel is used for the conversion. For example, to read from channel 5, set these bits to the pattern 0101.

Other registers are available such as ADSRB and ACSR which are useful for other modes of operation. They are not necessary for the current purpose, though.

The code follows, cleaned up for ease of reading. The original code contains a considerable number of #ifdefs so it works with different microcontrollers.

```
int analogRead(uint8_t pin)
{
    uint8_t low, high;

    if (pin >= 14) pin -= 14; // allow for channel or pin numbers

    // set the analog reference, input pin and clear left-justify (ADLAR)
    ADMUX = (analog_reference << 6) | (pin & 0x07);

    // start the conversion
    sbi(ADCSRA, ADSC);

    // ADSC is cleared when the conversion finishes
    while (bit_is_set(ADCSRA, ADSC));

    // read low and high bytes of result
    low  = ADCL;
    high = ADCH;

    // combine the two bytes
    return (high << 8) | low;
}
```

Figure 26.3, analogRead code

Let's take a closer look. First, two unsigned bytes are declared to hold the contents of the high and low result registers; then the pin argument is translated. Note the undocumented usage of either channel or pin numbers. Always be cautious about using undocumented features.

```
uint8_t low, high;

if (pin >= 14) pin -= 14; // allow for channel or pin numbers
```

At this point the reference is set (note how the `analog_reference` global is shifted up to the `REFSx` bits). The `pin` number is ANDed with 0x07 for safety and ORed into `ADMUX`. Note that the result will be right-justified as `ADLAR` is not set.

```
ADMUX = (analog_reference << 6) | (pin & 0x07);
```

The conversion is initiated by setting the `AD Start Conversion` bit. `sbi()` is a macro that reduces to a single "set bit" instruction in assembly. We then wait until the `ADSC` bit is cleared which signifies the conversion is complete. The `while` loop is referred to properly as a "busy-wait" loop; the code simply "sits" on that bit, checking it over and over until it is finally cleared. This causes the loop to exit. Again, `bit_is_set()` is a macro that returns `true` if the bit under test is set to one and `false` if it's clear.

```
// start the conversion
sbi(ADCSRA, ADSC);

// ADSC is cleared when the conversion finishes
while ( bit_is_set(ADCSRA, ADSC) );
```

The conversion is now complete. `ADCL` must be read first as doing so locks both the `ADCL` and `ADCH` registers until `ADCH` is read. Doing the reverse could result in spurious values. The two bytes are then combined into a single 16 bit integer result by ORing the low byte with the left-shifted high byte. As the data in the result registers were right-justified, the top six bits will be zeros in the returned integer, resulting in an output range of 0 through 1023.

```
low  = ADCL;
high = ADCH;

// combine the two bytes
return (high << 8) | low;
```

Given the default prescaler values and function call overhead, the maximum conversion rate is approximately 10 kHz which is perfectly acceptable for a wide variety of uses. The function is fairly bare-bones and straightforward to use as is. Use this function as a guide if you wish to produce left-justified data (set `ADLAR` bit) or other simple modifications. If time is of the essence and machine cycles cannot be wasted with the busy-wait loop seen above, an interrupt-based version or free-running version may be a better choice.

26.2 A Practical Example: Mapping a Sensor

As mentioned at the outset, the analog input system may be used to obtain data from a variety of sensors. One potential issue is that the range of sensor output voltages may not conveniently "line up" with the ADC input range, thus requiring some form of data scaling and offsetting. That is, the input voltages will

need to be mapped onto a range of desired numeric output values. Sometimes this can be done with hardware but other times a simple mathematical process will suffice.

For example, suppose we have a sensor that we'll use for measuring the temperature of liquid water at standard pressure. The specifications indicate that the unit will produce one volt at freezing (32 degrees Fahrenheit) and three volts at boiling (212 degrees Fahrenheit). Assuming that we're using the default reference which yields an input range of zero to five volts, we'll only be using 40% of the available dynamic range (i.e., two volts out of five). To make full use of the ten bit ADC, we'd need to run the sensor through a signal conditioning circuit to map the two volt output range onto the five volt input range. This would require an amplifier with a level shift of one volt and a voltage gain of 2.5. That is, we subtract one volt from the sensor output and then multiply the result by 2.5. By doing so, the 32 degree output of one volt will produce zero volts and the 212 degree output of three volts will produce five volts. A modest op amp or discrete transistor stage can be designed to achieve this. At the ADC input the total range of 180 degrees will be divided up into 1024 levels, yielding a resolution of better than 0.2 degrees per step. Note that the numeric values from the ADC are not the temperature values. A temperature of 32 degrees yields an ADC output of zero while 212 degrees produces 1023 from the ADC. If we need to display the Fahrenheit temperatures, perhaps on a multiplexed seven segment or LCD display, we'll need to map these. This can be done with a simple formula that we'll look at in a moment or with the Arduino **`map()`** function.

If a more modest resolution can be tolerated, a simple software mapping can be used. If we take the sensor output "as is", we have to accept that we'll only be getting 40% of the available resolution in this example. That will still give us around one-half degree resolution. Assuming this is sufficient, for the numeric readout just mentioned, all we need to do is map an input voltage of one volt to a numeric output of 32 and an input of three volts to a numeric output of 212. The one volt input represents one-fifth of the ADC's maximum input signal and would produce a value of approximately 205. The three volt input would produce approximately 614. 205 represents the offset of 32 degrees and the range of 614−205 or 409 needs to represent the temperature range of 180 degrees. We can do this with one small formula:

```
// av is the value returned from analogRead() and
// dv is the value to be displayed

dv = 32+180*(av-205)/409;
```

Note that this computation is done entirely with integer math rather than floating point for best possible speed and minimal compiled code. For very large values of the variable `av` the product could overflow an `unsigned int` so a `long` computation would be needed. Note that the multiply must be done before the integer divide or a horrendous loss of precision will occur due to truncation. `dv` itself could still be a `short` (or even an `unsigned char`), so a little creative casting might yield something like this:

```
dv = (short)(32+180*(long)(av-205)/409);
```

As we are using integers, the displayed temperature resolution will be one degree (truncating).

26.3 Exercise

Using a sensor that produces two volts at freezing and four volts at boiling, describe a circuit that will scale the sensor output to a zero-to-five volt input range along with the code required to display the associated values in Celsius (i.e., two volts displays 0 and four volts displays 100).

27. Bits & Pieces: analogWrite()

While some more advanced microcontrollers contain a digital to analog converter (DAC) to produce a continuously variable analog output signal, most do not. The ATmega 328P on the Uno is one of those that don't. How then can we get the controller to produce analog signals? One method is to use an entire bank of port pins, say all eight bits of port B, and feed these directly to an external parallel-input DAC. We then write each byte to the port at the desired rate. The DAC (along with a low-pass reconstruction filter) reconstructs these data into a smooth signal. The second method relies on pulse width modulation. This scheme is employed on the Uno and many other Arduino boards. Note that not all loads will operate properly with a simple PWM signal. Some loads may require further processing of the PWM signal (such as filtering).

The key to understanding pulse width modulation is to consider the "area under the curve". Suppose we have a one volt pulse signal that lasts for a period of five seconds, goes low for five seconds and then repeats. The same area would be achieved by a five volt pulse that stayed high for just one second out of ten. That is, the average value of either pulse over the course of ten seconds is one half volt. Similarly, if that five volt pulse stayed high for two seconds, then over the course of the ten second period the average value would be one volt. In other words; the greater the duty cycle, the higher the average voltage level. If we sped this up and the pulses were milliseconds or microseconds in width and repeated over and over, we could low pass filter the pulse train and achieve smoothly varying results. For some loads, we wouldn't even have to filter the result. Examples would include driving a resistive heating element to control temperature or driving an LED to control brightness.

The Uno achieves PWM through the use of its three internal counters. Each of these has an A and B component, so it's possible to generate six PWM signals. The Arduino system pre-configures these for you. You can tell which pins are PWM-capable just by looking at the board. PWM pins will have a tilde (~) next to their Arduino pin number. The `analogWrite()` function takes a great deal of work off your shoulders. Here is the on-line documentation repeated for your convenience:

> **analogWrite()**[24]
>
> ## Description
>
> Writes an analog value (PWM wave) to a pin. Can be used to light a LED at varying brightnesses or drive a motor at various speeds. After a call to **analogWrite()**, the pin will generate a steady square wave of the specified duty cycle until the next call to **analogWrite()** (or a call to **digitalRead()** or **digitalWrite()** on the same pin). The frequency of the PWM signal is approximately 490 Hz.
>
> On most Arduino boards (those with the ATmega168 or ATmega328), this function works on pins 3, 5, 6, 9, 10, and 11. On the Arduino Mega, it works on pins 2 through 13. Older Arduino boards with an ATmega8 only support analogWrite() on pins 9, 10, and 11.
>
> The Arduino Due supports analogWrite() on pins 2 through 13, plus pins DAC0 and DAC1. Unlike the PWM pins, DAC0 and DAC1 are Digital to Analog converters, and act as true analog outputs.
>
> You do not need to call pinMode() to set the pin as an output before calling analogWrite().

---

[24] http://arduino.cc/en/Reference/AnalogWrite

The *analogWrite* function has nothing to do with the analog pins or the *analogRead* function.

Syntax

analogWrite(pin, value)

Parameters

pin: the pin to write to.

value: the duty cycle: between 0 (always off) and 255 (always on).

Returns

nothing

Notes and Known Issues

The PWM outputs generated on pins 5 and 6 will have higher-than-expected duty cycles. This is because of interactions with the millis() and delay() functions, which share the same internal timer used to generate those PWM outputs. This will be noticed mostly on low duty-cycle settings (e.g 0 - 10) and may result in a value of 0 not fully turning off the output on pins 5 and 6.

See also

- analogRead()

- analogWriteResolution()

- Tutorial: PWM

Figure 27.1, analogWrite docs

The bottom line is that you call analogWrite() with just the pin of interest and a duty cycle value that ranges from 0 (off) to 255 (fully on). That's all there is to it. But why is the range 0 to 255 instead of the more obvious 0 to 100 percent? Let's take a look at the code, as usual slightly cleaned up for your convenience (the original code may be found in the file wiring_analog.c):

```
void analogWrite(uint8_t pin, int val)
{
      pinMode(pin, OUTPUT);

      if (val == 0)
      {
            digitalWrite(pin, LOW);
      }
      else
      {
            if (val == 255)
            {
                  digitalWrite(pin, HIGH);
            }
            else
            {
```

```
                switch( digitalPinToTimer(pin) )
                {
                case TIMER0A:
                        // connect pwm to pin on timer 0, channel A
                        sbi(TCCR0A, COM0A1);
                        OCR0A = val; // set pwm duty
                        break;

                case TIMER0B:
                        // connect pwm to pin on timer 0, channel B
                        sbi(TCCR0A, COM0B1);
                        OCR0B = val; // set pwm duty
                        break;

                // and so on for TIMER1A through TIMER2B

                case NOT_ON_TIMER:
                default:
                        if (val < 128)
                                digitalWrite(pin, LOW);
                        else
                                digitalWrite(pin, HIGH);
                }
        }
    }
}
```

Figure 27.2, analogWrite code

The first thing we see is a call to `pinMode()` to guarantee that the pin is set up for output. While it's possible to simply require programmers to make this function call themselves before each usage, it is safer to place it inside the function. Note that if you were only using this pin in this mode, you could make this call yourself just once as part of your setup routine and make your own version of `analogWrite()` having removed this part (and perhaps some other sections) to shave the execution time.

```
        pinMode(pin, OUTPUT);
```

The code then does a quick check to see if we want fully on or fully off, in which case it dispenses with the timer and just calls `digitalWrite()`.

```
        if (val == 0)
        {
                digitalWrite(pin, LOW);
        }
        else
        {
```

```
        if (val == 255)
        {
                digitalWrite(pin, HIGH);
        }
```

At this point, a value from 1 to 254 must have been entered. The pin is translated to a timer and, through a `switch/case` statement, the appropriate timer's control register is activated and the duty cycle value is loaded into the associated count register. These timers are eight bit units. Therefore they can hold values between 0 and 255. This is why the "duty cycle" value runs from 0 to 255 instead of 0 to 100 percent. (Technically, timer one is a 16 bit unit but is used here with eight.)

```
        switch( digitalPinToTimer(pin) )
        {
                case TIMER0A:
                        // connect pwm to pin on timer 0, channel A
                        sbi(TCCR0A, COM0A1);
                        OCR0A = val; // set pwm duty
                        break;

                case TIMER0B:
                        // connect pwm to pin on timer 0, channel B
                        sbi(TCCR0A, COM0B1);
                        OCR0B = val; // set pwm duty
                        break;

                // and so on for TIMER1A through TIMER2B
```

Finally, if `analogWrite()` is used on a pin that is not configured with a timer, the function does the best it can, namely calling `digitalWrite()`. Any value below half (128) will produce a low and any value at half or above will produce a high:

```
        case NOT_ON_TIMER:
        default:
                if (val < 128)
                        digitalWrite(pin, LOW);
                else
                        digitalWrite(pin, HIGH);
```

It is worth repeating that on Arduino boards that contain an internal DAC such as the Due, this function will produce true analog output signals (the code for that is not shown).

28. Bits & Pieces: Timer/Counters

28.1 Introduction

Virtually all microcontrollers contain one or more hardware timer/counter function blocks. These can be used for a variety of functions including generating time delays, counting input events, generating simple pulse or PWM waveforms, and triggering software interrupts. In general, there are multiple ways to configure these blocks. Often, they are set up as simple up counters that are clocked by the system clock. In some instances they may be run asynchronously from an external clock source.

The ATmega 328P contains three of these blocks: two eight bit units named TC0 and TC2 (Timer Counter 0 and 2), and one 16 bit unit, TC1. A functional block diagram of an eight bit unit is shown in Figure 28.1. The 16 bit unit is similar but offers extended abilities.
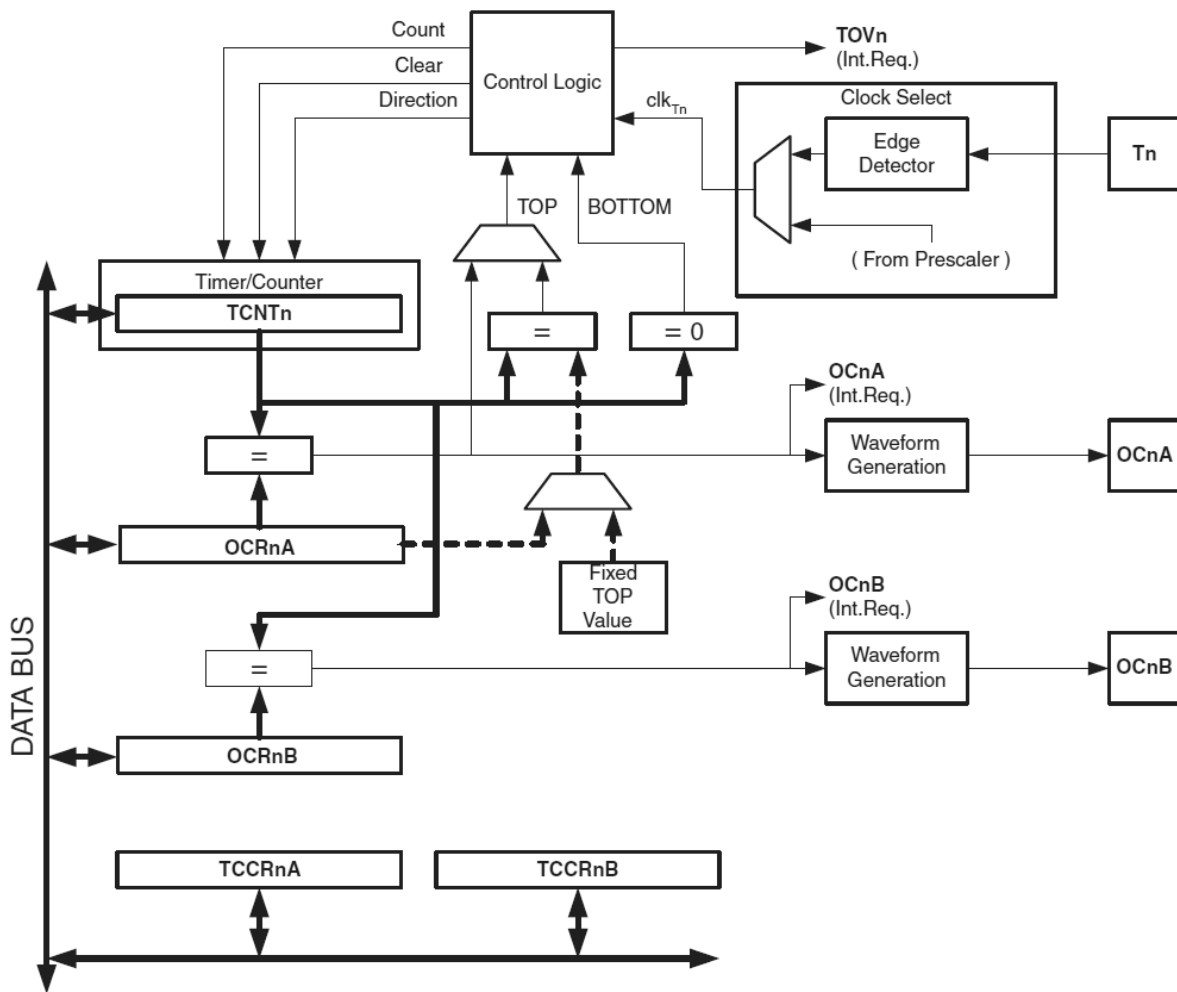

Figure 28.1, timer/counter functional block diagram (eight bit)

First, there are a couple of registers used to program the operation of the block. These are `TCCRnA` and `TCCRnB` (Timer Counter Control Registers A and B) where *n* is the timer/counter number (0, 1 or 2 here, although other microcontrollers in the series may have more). These bits are usually "set and forget", that is, they are set/cleared for a given use and not touched again unless the unit needs to be reprogrammed for a different use. More detail on these will be presented momentarily. There are also two registers that are used with software interrupts, `TIFRn` and `TIMSKn` (Timer Interrupt Flag Register and Timer Interrupt MaSK register) that aren't shown here. We will examine these in the section on software interrupts. The other key registers are `TCNTx` (Timer CouNT) along with `OCRnA` and `OCRnB` (Output Compare Registers A and B).

In its most basic operation the unit increments the `TCNTn` register with each tick of the system clock. Eventually, this register will reach its maximum value and overflow, resulting in zero, effectively resetting the register and the count continues up from there. This maximum value is 255 for an eight bit unit (i.e., 11111111 binary) and 65535 for a 16 bit unit. Optionally, the unit may be programmed to inspect the values contained in the `OCRn` registers and compare them to the current contents of `TCNTn` to see if they match. Both the overflow and the compare match can be used trigger some action, e.g., a waveform level change or software interrupt. The compare match section feeds a pulse waveform generator that in turn feeds an output pin. This section is shown in Figure 28.2.
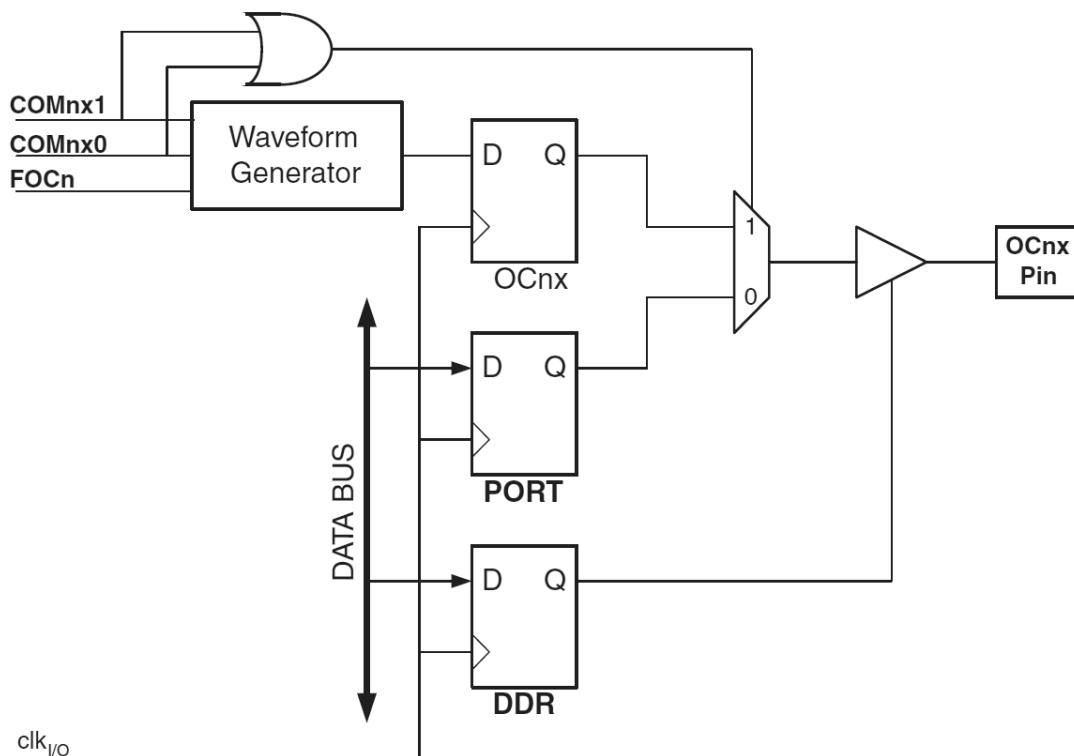

Figure 28.2, timer/counter compare match output circuitry

The box marked "OCnx Pin" is a physical pin on the microcontroller. As it would be impractical to have single purpose pins for every function in the microcontroller, the waveform function is mapped onto the general purpose IO pins. Compare this section to the GPIO block diagram discussed in chapter 19. Note the PORT and DDR flip-flops in particular. The waveform generator feeds a flip flop which then feeds a multiplexer along with the PORT flip flop. This signal is then fed to a tri-state buffer controlled by the DDR which leads to the physical output pin. As there are three timer/counters and each has two compare match registers, there are a total of six physical output pins available for waveform generation. The mapping assignments are shown in Figure 28.3.

OCnx	PORT.bit	Arduino Pin
OC0A	D.6	6
OC0B	D.5	5
OC1A	B.1	9
OC1B	B.2	10
OC2A	B.3	11
OC2B	D.3	3

Figure 28.3, waveform pin assignments

The following discussion will focus on timer/counter number two. This is because timer/counter units zero and one are already used for common Arduino system calls such as delay(). Reprogramming the timer/counters could alter the operation of those functions and procedures. Timer/counter number two is a little safer to experiment with in that regard although be aware that all three units are used to cover the six PWM capable output pins on the Uno (i.e., via analogWrite()).

The two control registers (TCCRnA/B) indicate the mode of operation. The registers are shown in Figures 28.4 and 28.5 for unit number two. The control registers for the other timer/counter units are similar and details may be found in the Appendix.

Bit	7	6	5	4	3	2	1	0
Function	COM2A1	COM2A0	COM2B1	COM2B0	-	-	WGM21	WGM20

Figure 28.4, TCCR2A register (Atmel 2014)

Bit	7	6	5	4	3	2	1	0
Function	FOC2A	FOC2B	-	-	WGM22	CS22	CS21	CS20

Figure 28.5, TCCR2B register (Atmel 2014)

The first items of interest are the `CS` bits. These are the Clock preScale bits and they allow for longer timer periods by dividing down the system clock before it feeds the counter. For example, if the controller is running at 16 MHz then each clock tick represents 1/16 MHz or 62.5 nanoseconds. Counting up from 0 to 255 would yield a maximum delay of only 62.5 nanoseconds times 256, or 16 microseconds. If the prescaler is set to eight, each tick is stretched by a factor of eight yielding a maximum delay of 128 microseconds in this example. The prescaler settings for TC2 are shown in Figure 28.6.

CS22	CS21	CS20	Divide By
0	0	0	TC Stopped
0	0	1	1 (no prescale)
0	1	0	8
0	1	1	32
1	0	0	64
1	0	1	128
1	1	0	256
1	1	1	1024

Figure 28.6, prescaler settings for TC2

Turning to the Waveform Generation Mode (`WGM`) bits, there are normal mode, a compare mode (`CTC` or Clear Timer on Compare) and two forms of pulse width modulation. We will examine Normal and Fast PWM modes here and CTC mode in the chapter covering interrupts. Note that the `WGM` bits are spread across both control registers rather than residing in a single register. The `WGM` bits are detailed in Figure 28.7.

WGM22	WGM21	WGM20	Mode
0	0	0	Normal
0	0	1	Phase Correct PWM, Top=0xff
0	1	0	CTC
0	1	1	Fast PWM, Top=0xff
1	0	0	Reserved
1	0	1	Phase Correct PWM, Top=OCRA
1	1	0	Reserved
1	1	1	Fast PWM, Top=OCRA

Figure 28.7, waveform generation mode bits for TC2

Finally let's consider the Compare Output Mode (`COM`) bits found in `TCCR2A`. There are two bits for `OC2A` and two for `OC2B`, and although the settings are similar they are not identical (for details, see the Atmel

documentation). The settings for OC2A are found in Figures 28.8 through 28.10 for three waveform generation modes and describe how the output pin signal responds. Note that in all three of these modes if both COM bits are clear then the output pin, OC2A, is disconnected from the timer/counter and operates in ordinary IO form. With other settings of the COM bits, OC2A is controlled by the waveform generator. It is important to note that the associated DDR must still be set for output mode in order for the waveform to reach the output pin. Examine the block diagram of Figure 28.2 for verification.

COM2A1	COM2A0	Description
0	0	Normal, OC2A disconnected
0	1	Toggle OC2A on Compare Match
1	0	Clear OC2A on Compare Match
1	1	Set OC2A on Compare Match

Figure 28.8, non-PWM, Compare Output Mode for TC2

COM2A1	COM2A0	Description
0	0	Normal, OC2A disconnected
0	1	WGM22=0: Normal, OC2A disconnected WGM22=1: Toggle OC2A on Compare Match (Reserved operation for OC2B)
1	0	Clear OC2A on Compare Match, Set OC2A on 0x00 (non-inverting mode)
1	1	Set OC2A on Compare Match, Clear OC2A on 0x00 (inverting mode)

Figure 28.9, Fast PWM, Compare Output Mode for TC2

COM2A1	COM2A0	Description
0	0	Normal, OC2A disconnected
0	1	WGM22=0: Normal, OC2A disconnected WGM22=1: Toggle OC2A on Compare Match (Reserved operation for OC2B)
1	0	Clear OC2A on Compare Match when up-counting, Set when down-counting
1	1	Set OC2A on Compare Match when up-counting, Set when down-counting

Figure 28.10, Phase Correct PWM, Compare Output Mode for TC2

Embedded Controllers

28.2 Normal Mode

Normal mode (WGM22:0 are clear) is the simplest mode of operation. The TC simply counts up until it overflows and wraps around back to zero. On overflow, the TOV2 flag (Timer OVerflow) is set in the TIFR2 flag register. More precisely, this bit will be set in the same clock cycle as when the timer/counter register TCNT2 becomes zero. There are no access restrictions on TCNT2. Your code may overwrite the contents of this register at any time in order to alter the counting behavior.

A short code example follows. The program uses TC2 to create a time delay similar to the delay() function. This delay is then used to blink an LED connected to PORTB.0.

```
/* Simple LED blinker using TC2 to create a time delay
   Active high LED connected to PORTB.0 */

#define LEDMASK 0x01
#define COUNTOFFSET 15

void setup()
{
  DDRB |= LEDMASK;

  TCCR2A=0;    // normal mode
  TCCR2B=0x07; // 1024x prescale
}

void mydelay(int x)
{
  while(x)
  {
    TCNT2=COUNTOFFSET;  //count up from here to 255 then overflow
    TIFR2=1<<TOV2;      // clear flag, bit is set on overflow
    while( !(TIFR2&(1<<TOV2)) ); // wait for overflow bit
    x--;
  }
}

void loop()
{
  PORTB |= LEDMASK;
  mydelay(120);
  PORTB &= ~LEDMASK;
  mydelay(30);
}
```

The setup() function sets the direction for the LED driver pin to output, places TC2 into normal mode and sets the clock prescaler to 1024. This means that each count will occur at a rate just slower than 16 kHz given the 16 MHz clock of the Uno. The mydelay() function initializes the counter's main register at a predefined value (COUNTOFFSET). It will count from there to 255 after which point it will overflow and set the TOV2 flag. We ensure that this bit is clear before continuing using the seemingly backward command TIFR2=1<<TOV2; It first appears that this command should set the flag bit but it does indeed clear it (as verified in the Atmel documentation). At this point we busy-wait on this bit waiting for it to be set. In this example it will take 241 counts to overflow or about 15 milliseconds. This

process is then iterated `x` times to achieve a longer and variable delay. Note that larger values of COUNTOFFSET will yield shorter blink times.

28.3 Fast PWM Mode

This is the "fast" pulse width modulation mode that can drive an output pin directly. Fast PWM is faster than phase correct PWM because fast PWM performs a single slope (i.e., up only) count. Phase correct PWM uses an up-then-down dual slope counting technique. We will not examine phase correct PWM here.

To use fast PWM mode to drive an output pin, first remember to set the corresponding DDR bit to output mode (see Figure 28.3). Following this, the WGM and COM bits need to be set according to Figures 28.7 and 28.9 as well as the desired prescaler CS bits from Figure 28.6. Finally a value for the output compare registers (OCR2A and OCR2B) needs to be set. The value of the OCR2x registers will determine the duty cycles of the resulting pulse waveform. Basically, TCNT2 will count up in the normal fashion, overflowing and recycling as usual. When the contents of TCNT2 match those of an associated output compare register, then the corresponding output pin will change to the appropriate state. The duty cycle is defined by the total count range (256) and the compare register value plus one. An example is seen below.

```
/* Waveform at external pins OC2A and OC2B (PORTB.3/PORTD.3)
   Fast PWM mode, non-inverting */

void setup()
{
  // enable output drivers for OC2A/B!
  DDRB |= 0x08;
  DDRD |= 0x08;

  // the following is done in two steps for clarity
  TCCR2A = (1<<WGM21) | (1<<WGM20);      // specify fast PWM
  TCCR2A|= ((1<<COM2A1) | (1<<COM2B1)); // add in non-inverting output

  TCCR2B = (1<<CS22);                    // prescale by 64x

  OCR2A = 128; // set duty cycles (0 through 255)
  OCR2B = 100;
}

void loop()
{
   // nothing to do here!
}
```

An interesting new twist here is that there is nothing in the main `loop()` function! Once the timer/counter is set up it runs without further attention, truly "set and forget" operation. In this case, the output frequency will be the main clock of 16 MHz divided by the 64x prescaler and then divided by the total count of 256. This yields approximately 977 Hz. The duty cycle for output A is (128+1)/256 or about 50.4%. For output B the result is (100+1)/256 or approximately 39.5%.

The preceding code was downloaded to an Arduino Uno and an oscilloscope was attached to the two output pins. The resulting waveforms are shown in Figure 28.11. Note the relative accuracy of the timing computations compared to those measured by the oscilloscope.

It is worthwhile to note that the values of the output compare registers could be changed programmatically, for example via a potentiometer connected to an analog input pin. This would effectively recreate an alternate form of the `analogWrite()` function.
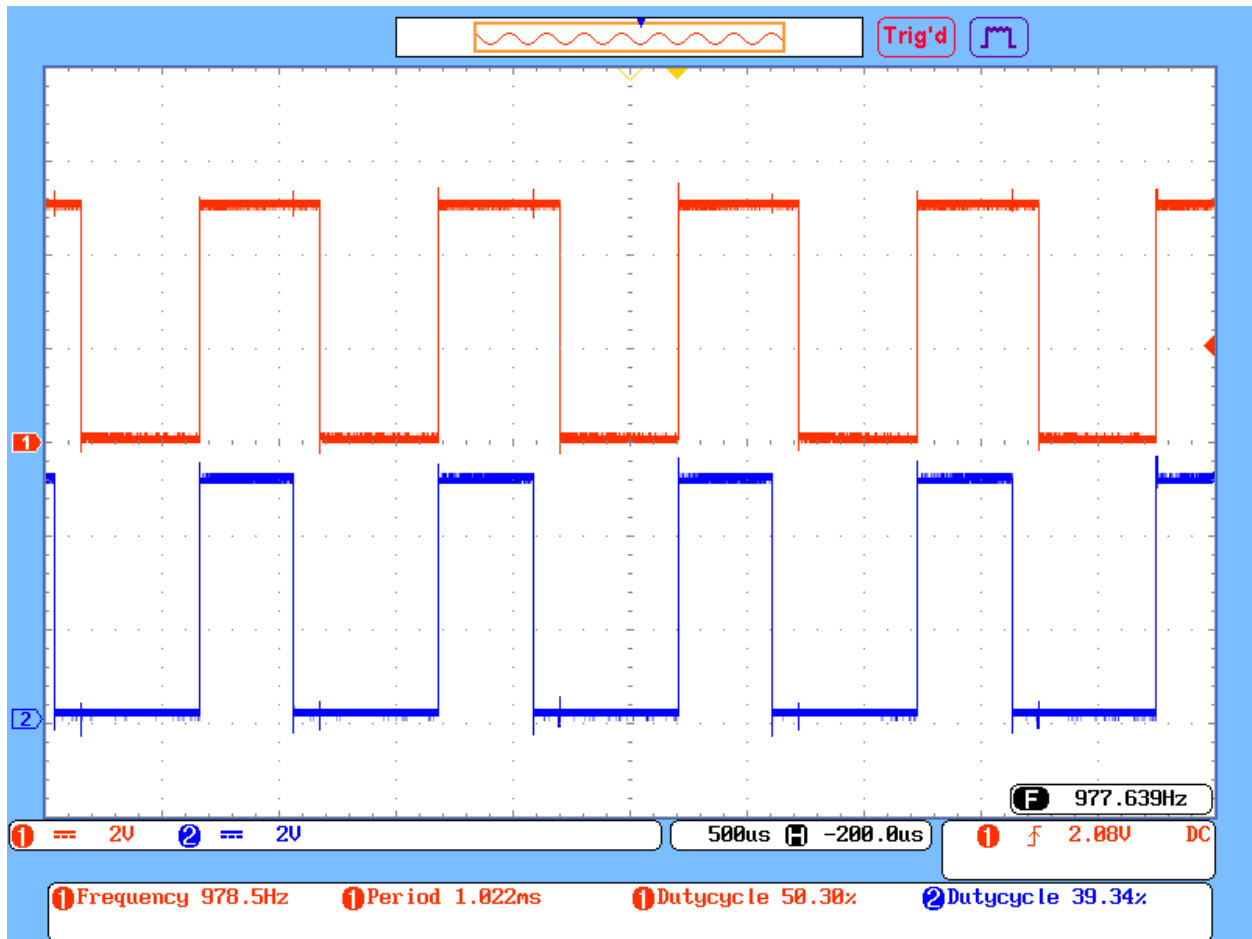

Figure 28.11, Fast PWM output waveforms

29. Bits & Pieces: Interrupts

29.1 Introduction

Interrupts may be thought of as small snippets of code that are of such high priority that the main program flow is momentarily halted so that the interrupt code can be run. Once the interrupt code (referred to as an *interrupt service routine*, or ISR) is finished, the main code picks up where it left off. When dealing with interrupts always keep in mind that they could happen at any time, even in the middle of a line of C code (this is due to the fact that a single line of C might generate several lines of native machine code and the interrupt could occur in between machine code instructions).

Interrupts usually come from two sources: an externally triggered event such as a physical pin changing state or via an internally triggered event. For example, an external switch could be used to force the program to reset itself from the start. Alternately, the overflow and compare match events of internal timer/counters are often used to create internal interrupts that can control the timing of other events.

When an interrupt occurs, the current state and location in the executing code is saved. The microcontroller then jumps to the starting address of the ISR. The ISR code is executed and upon completion program flow reverts back to where it left off in the main sequence. As the main program flow is being halted temporarily, ISRs are designed to be short and to execute quickly so as to have minimal timing impact on the main code. As most microcontrollers have numerous sources for interrupts, interrupts are usually prioritized, some being more important than others. This means it is possible for one interrupt to interrupt another interrupt (i.e., nested interrupts having different priorities). In this way, something like a fire alarm sensor could take precedence over a more mundane sensor input such as a volume control.

It is quite possible for a complicated application to have over a dozen possible interrupt sources. Each of these will have its own ISR. The starting locations of the ISRs are found in a *vector table*. A vector table is basically an array of pointers. In this case, these pointers point to the starting address of some code rather than to the addresses of variables. Some programmers will use the word "vector" as a verb as in "the code flow *vectors off* to the ISR".

To simplify programming, the names of the ISRs are predetermined. As a programmer you simply need to fill in the code for the ISR. This might be as simple as setting a global variable. When the program is compiled, the predetermined name is noted in your code and then expanded into the vector table for you automatically. Generally speaking, ISRs do not take arguments nor do you need to create function prototypes for them. A list of ISR names for the ATmega 328P series is shown in Figure 29.1. This list is taken directly from the hardware profile file `include/avr/iom328p.h`. Note that each ISR name ends in `_vect` while the first portion of the name indicates the hardware with which it associated. A quick scan of the list should reveal a number of hardware elements covered earlier such as the three timer/counters and the analog to digital converter.

Finally, before these interrupts can be used the desired interrupt must be enabled by setting the appropriate bit in the associated register (`EIMSK`, External Interrupt MaSK; `TIMSKx`, Timer Interrupt MaSKx; etc., see the Appendix and following examples for details). Of course, the global interrupt enable must also be set (which can be accomplished via the `sei()` call) although this is already the case in the Arduino system.

```
/* Interrupt Vector 0 is the reset vector. */
#define INT0_vect         _VECTOR(1)   /* External Interrupt Request 0 */
#define INT1_vect         _VECTOR(2)   /* External Interrupt Request 1 */
#define PCINT0_vect       _VECTOR(3)   /* Pin Change Interrupt Request 0 */
#define PCINT1_vect       _VECTOR(4)   /* Pin Change Interrupt Request 1 */
#define PCINT2_vect       _VECTOR(5)   /* Pin Change Interrupt Request 2 */
#define WDT_vect          _VECTOR(6)   /* Watchdog Time-out Interrupt */
#define TIMER2_COMPA_vect _VECTOR(7)   /* Timer/Counter2 Compare Match A */
#define TIMER2_COMPB_vect _VECTOR(8)   /* Timer/Counter2 Compare Match B */
#define TIMER2_OVF_vect   _VECTOR(9)   /* Timer/Counter2 Overflow */
#define TIMER1_CAPT_vect  _VECTOR(10)  /* Timer/Counter1 Capture Event */
#define TIMER1_COMPA_vect _VECTOR(11)  /* Timer/Counter1 Compare Match A */
#define TIMER1_COMPB_vect _VECTOR(12)  /* Timer/Counter1 Compare Match B */
#define TIMER1_OVF_vect   _VECTOR(13)  /* Timer/Counter1 Overflow */
#define TIMER0_COMPA_vect _VECTOR(14)  /* Timer/Counter0 Compare Match A */
#define TIMER0_COMPB_vect _VECTOR(15)  /* Timer/Counter0 Compare Match B */
#define TIMER0_OVF_vect   _VECTOR(16)  /* Timer/Counter0 Overflow */
#define SPI_STC_vect      _VECTOR(17)  /* SPI Serial Transfer Complete */
#define USART_RX_vect     _VECTOR(18)  /* USART, Rx Complete */
#define USART_UDRE_vect   _VECTOR(19)  /* USART, Data Register Empty */
#define USART_TX_vect     _VECTOR(20)  /* USART, Tx Complete */
#define ADC_vect          _VECTOR(21)  /* ADC Conversion Complete */
#define EE_READY_vect     _VECTOR(22)  /* EEPROM Ready */
#define ANALOG_COMP_vect  _VECTOR(23)  /* Analog Comparator */
#define TWI_vect          _VECTOR(24)  /* Two-wire Serial Interface */
#define SPM_READY_vect    _VECTOR(25)  /* Store Program Memory Read */
```

Figure 29.1, ISR vector table names

29.2 External Interrupts

Let's consider an external interrupt first. In this case, the state change seen at an external pin causes an interrupt. We will use External Interrupt Request 0, also known as the INT0 vector. This interrupt examines state changes at PORTD.2, Uno pin 2. Our example will be fairly simple: we attach a passive switch using the internal pull-up to pin 2. When the switch is activated it will toggle an LED connected to Uno pin 8 (PORTB.0). The code follows:

```
/* External Interrupt Example, INT0
   External interrupt pin lights an LED

   Active high LED attached to PORTB.0
   Switch on Uno pin 2, falling edge triggers external pin interrupt 0,
   ISR toggles LED at PORTB.0
*/

#define LEDMASK 0x01

void setup()
{
   DDRB |= LEDMASK;
```

```
        // set Uno pin 2 for input with pull-up
        DDRD &= ~(0x04);
        PORTD |= 0x04;

        EIMSK |= (1<<INT0); // enable external interrupt INT0 (Uno pin 2)
        EICRA |= (1<<ISC01); // trigger INT0 on falling edge
    }

    ISR(INT0_vect)
    {
        PORTB ^= LEDMASK; // toggle LED
    }

    void loop()
    {
    }
```

After setting the IO pins, the code sets the `INT0` bit in the `EIMSK` register to enable the interrupt. The External Interrupt Control Register (`EICRA`) is programmed to trigger on a falling (negative going) edge. When the switch is activated, the falling edge triggers the ISR to run. The ISR simply toggles the bit that the LED is tied to. Note that there is no port setting and pin examination in the main code. In fact, `loop()` is empty and does nothing.

Among other things, an external interrupt such as this is very useful for high importance items such as a panic switch.

29.3 Internal Interrupts: Blinking LED

In this section we shall examine a series of examples using interrupts trigger by the internal timer/counters. This technique makes use of either the overflow or compare match events. It is usually used when you wish to have events occur at predictable times but without the penalty and limitations of using a simple `delay()` style approach. That is, the triggers for these events will not need your attention in the main program loop but will generally be of a "set and forget" variety. There is a Timer Interrupt MaSK register for each of the counters (`TIMSK0`, `TIMSK1` and `TIMSK2`). The appropriate bit(s) must be set to enable the desired interrupt.

The first example shows a simple method of blinking an LED "hands off". The duty cycle of the blink will always be 50% although it is possible to alter this. The example also shows how to generate long time intervals from an eight bit timer/counter through the use of a global variable. The example is shown below.

```
/* Timer/Counter Interrupt Example
   Shows how to blink an LED "hands off" using overflow
   Also shows how to get a large timing value from an 8 bit counter
   normal count mode
*/
volatile int g_time = 0;

#define LEDMASK 0x01
```

```
#define ISR_COUNT 50

void setup()
{
  DDRB |= LEDMASK;

  TCCR2A = 0;    //  normal mode, OC2x pin disconnected
  TCCR2B = 0x07; // 1024x prescale

  TIMSK2 = (1<<TOIE2);  // enable overflow interrupt
}

ISR(TIMER2_OVF_vect)
{
  if(!g_time)
    PORTB ^= LEDMASK;

  g_time++;
  if( g_time >= ISR_COUNT )
    g_time = 0;
}

void loop()
{
}
```

Timer/counter 2 is used here in normal mode meaning that it will count from 0 to 255 and then overflow and repeat. The prescaler is set for 1024x, producing ticks at about 16 kHz. The overflow interrupt for this counter is then enabled. This triggers the TIMER2_OVF_vect.

A full count from 0 to 255 will create an overflow at a rate of slightly over 60 Hz. This means that the LED will have a single on/off cycle at a rate of about 30 per second which is a little too fast for the naked eye. Therefore we can't simply toggle the LED as it will always appear on. Instead, we make use of a global variable, g_time, that will keep track of how many times the interrupt has occurred. Once it hits our predetermined maximum, ISR_COUNT, it will be reset and the LED toggled. We now have a blinking LED that is completely "hands off" as far as the main code flow is concerned. Again, note that loop() is empty. This will continue to be the case in the remaining examples.

29.4 Internal Interrupts: Hand Wrought PWM

Our next example shows how a timer/counter overflow can be used to create a PWM signal at an arbitrary pin (i.e., not just at the OCnx pins). For this example we shall forego the use of an LED indicator and simply inspect the signal with an oscilloscope. The output frequency is slightly over 60 Hz and the duty cycle is set by a #define although it could just as easily be set by a variable. The duty cycle is (OVF_COUNT_START+1)/256. Thus, a value of 128 yields approximately 50% while a value of 50 yields nearly 20%. The output pin is set to Uno pin 8 or PORTB.0. The code follows:

```
/* Timer/Counter Interrupt Example
   Shows how to create hand wrought PWM at an arbitrary pin
*/
```

```
#define ARBPINMASK 0x01
#define OVF_COUNT_START 128

void setup()
{
  DDRB |= ARBPINMASK;

  TCCR2A = 0;    // normal mode, OC2x pin disconnected
  TCCR2B = 0x07; // 1024x prescale

  TIMSK2 = (1<<TOIE2);      // enable overflow interrupt
  TCNT2 = OVF_COUNT_START; // init counter, count up from here
}

ISR(TIMER2_OVF_vect)
{
  if( PORTB &= ARBPINMASK )
    TCNT2 = OVF_COUNT_START;
  else
    TCNT2 = 255-OVF_COUNT_START;

  PORTB ^= ARBPINMASK;
}

void loop()
{
}
```

The code is very similar to that of the prior example with the addition of the alteration of the `TCNT2` count register. Each interrupt causes the pin to toggle, however, unlike the previous example the count register is set to a new starting value which shortens the count time (i.e., the time to overflow). Further, the reset value is dependent on whether the current output level is high or low.

29.5 Internal Interrupts: CTC Mode

This example also involves toggling an arbitrary output pin although the `OCnx` pin could be piggybacked if desired. Instead of PWM, this example varies the frequency of a square wave using Clear Timer on Compare (`CTC`) mode. This mode simply counts up to the value stored in the output compare match register (`OCR2A`) and then resets back to zero.

```
/* Timer/Counter Interrupt Example
   Uses CTC mode (Clear Timer on Compare)
*/

#define ARBPINMASK 0x01
#define COMPARE_MATCH 128

void setup()
{
  DDRB |= ARBPINMASK;
```

```
  // enable output driver for OC2A for piggyback!
  DDRB |= 0x08;

  TCCR2A = (1<<COM2A0);  // non PWM, OC2A pin toggle on compare match
  TCCR2A |= (1<<WGM21);  // add CTC mode, Top=OCR2A

  TCCR2B = 0x07;         // 1024x prescale

  OCR2A = COMPARE_MATCH;

  TIMSK2 = (1<<OCIE2A); // enable compare match interrupt
}

ISR(TIMER2_COMPA_vect)
{
  PORTB ^= ARBPINMASK;
}

void loop()
{
}
```

Note that a new bit must be set in the timer/counter interrupt mask register, namely OCIE2A (Output Compare Interrupt Enable for unit 2A). The Output Compare Register (OCR2A) is set with a fixed value but again, this could be set by a variable. A COMPARE_MATCH value of 128 yields a square wave frequency of slightly more than 60 Hz. Halving the value to 64 will roughly double the frequency to about 120 Hz.

Appendix A

ATmega 328P Register Map

Derived from the October, 2014 version of the Atmel 328P documentation which may
be found at http://www.atmel.com/devices/ATMEGA328P.aspx

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x23	PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0
0x24	DDRB	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0
0x25	PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0
0x26	PINC	-	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0
0x27	DDRC	-	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0
0x28	PORTC	-	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0
0x29	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0
0x2A	DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0
0x2B	PORTD	PORT7	PORT6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0
0x2C–34	Reserved	-	-	-	-	-	-	-	-
0x35	TIFR0	-	-	-	-	-	OCF0B	OCF0A	TOV0
0x36	TIFR1	-	-	ICF1	-	-	OCF1B	OCF1A	TOV1
0x37	TIFR2	-	-	-	-	-	OCF2B	OCF2A	TOV2
0x38–3A	Reserved	-	-	-	-	-	-	-	-
0x3B	PCIFR	-	-	-	-	-	PCIF2	PCIF1	PCIF0
0x3C	EIFR	-	-	-	-	-	-	INT1	INT0
0x3D	EIMSK	-	-	-	-	-	-	INTF1	INTF0
0x3E	GPIOR0	General Purpose I/O Register 0							
0x3F	EECR	-	-	EEPM1	EEPM0	EERIE	EEMPE	EEPE	EERE
0x40	EEDR	EEPROM Data Register							
0x41	EEARL	EEPROM Address Register Low Byte							
0x42	EEARH	EEPROM Address Register High Byte							
0x43	GTCCR	TSM	-	-	-	-	-	PSRASY	PSRSYNC
0x44	TCCR0A	COM0A1	COM0A0	COM0B1	COM0B0	-	-	WGM01	WGM00
0x45	TCCR0B	FOC0A	FOC0B	-	-	WGM02	CS02	CS01	CS00
0x46	TCNT0	Timer/Counter0 (8-bit)							
0x47	OCR0A	Timer/Counter0 Output Compare Register A							
0x48	OCR0B	Timer/Counter0 Output Compare Register B							
0x49	Reserved	-	-	-	-	-	-	-	-
0x4A	GPIOR1	General Purpose I/O Register 1							
0x4B	GPIOR2	General Purpose I/O Register 1							
0x4C	SPCR	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0
0x4D	SPSR	SPIF	WCOL	-	-	-	-	-	SPI2X
0x4E	SPDR	SPI Data Register							
0x4F	Reserved	-	-	-	-	-	-	-	-

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x50	ACSR	ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0
0x51–52	Reserved	-	-	-	-	-	-	-	-
0x53	SMCR	-	-	-	-	SM2	SM1	SM0	SE
0x54	MCUSR	-	-	-	-	WDRF	BORF	EXTRF	PORF
0x55	MCUCR	-	BODS	BODSE	PUD	-	-	IVCEL	IVCE
0x56	Reserved	-	-	-	-	-	-	-	-
0x57	SPMCSR	SPMIE	RWWSB	SIGRD	RWWSRE	BLBSET	PGWRT	PGERS	SPMEN
0x58–5C	Reserved	-	-	-	-	-	-	-	-
0x5D	SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0
0x5E	SPH	-	-	-	-	-	SP10	SP9	SP8
0x5F	SREG	I	T	H	S	V	N	Z	C
0x60	WDTCSR	WDIF	WDIE	WDP3	WDCE	WDE	WDP2	WDP1	WDP0
0x61	CLKPR	CLKPCE	-	-	-	CLKPS3	CLKPS2	CLKPS1	CLKPS0
0x62–63	Reserved	-	-	-	-	-	-	-	-
0x64	PRR	PRTWI	PRTM2	PRTM0	-	PRTM1	PRSPI	PRUSART0	PRADC
0x65	Reserved	-	-	-	-	-	-	-	-
0x66	OSCCAL	Oscillator Calibration Register							
0x67	Reserved	-	-	-	-	-	-	-	-
0x68	PCICR	-	-	-	-	-	PCIE2	PCIE1	PCIE0
0x69	EICRA	-	-	-	-	ISC11	ISC10	ISC01	ISC00
0x6A	Reserved	-	-	-	-	-	-	-	-
0x6B	PCMSK0	PCINT7	PCINT6	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0
0x6C	PCMSK1	-	PCINT14	PCINT13	PCINT12	PCINT11	PCINT10	PCINT9	PCINT8
0x6D	PCMSK2	PCINT23	PCINT22	PCINT21	PCINT20	PCINT19	PCINT18	PCINT17	PCINT16
0x6E	TIMSK0	-	-	-	-	-	OCIE0B	OCIE0A	TOIE0
0x6F	TIMSK1	-	-	ICIE1	-	-	OCIE1B	OCIE1A	TOIE1
0x70	TIMSK2	-	-	-	-	-	OCIE2B	OCIE2A	TOIE2
0x71–77	Reserved	-	-	-	-	-	-	-	-
0x78	ADCL	ADC Data Register Low byte							
0x79	ADCH	ADC Data Register High byte							
0x7A	ADCSRA	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
0x7B	ADCSRB	-	ACME	-	-	-	ADPT2	ADPT1	ADPT0
0x7C	ADMUX	REFS1	REFS0	ADLAR	-	MUX3	MUX2	MUX1	MUX0
0x7D	Reserved	-	-	-	-	-	-	-	-
0x7E	DIDR0	-	-	ADC5D	ADC4D	ADC3D	ADC2D	ADC1D	ADC0D
0x7F	DIDR1	-	-	-	-	-	-	AIN1D	AIN0D
0x80	TCCR1A	COM1A1	COM1A0	COM1B1	COM1B0	-	-	WGM11	WGM10
0x81	TCCR1B	ICNC1	ICES1	-	WGM13	WGM12	CS12	CS11	CS10
0x82	TCCR1C	FOC1A	FOC1B	-	-	-	-	-	-
0x83	Reserved	-	-	-	-	-	-	-	-
0x84	TCNT1L	Timer/Counter1 - Counter Register Low Byte							
0x85	TCNT1H	Timer/Counter1 - Counter Register High Byte							
0x86	ICR1L	Timer/Counter1 - Input Capture Register Low Byte							
0x87	ICR1H	Timer/Counter1 - Input Capture Register High Byte							

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x88	OCR1AL	Timer/Counter1 - Output Compare Register A Low Byte							
0x89	OCR1AH	Timer/Counter1 - Output Compare Register A High Byte							
0x8A	OCR1BL	Timer/Counter1 - Output Compare Register B Low Byte							
0x8B	OCR1BH	Timer/Counter1 - Output Compare Register B High Byte							
8C–AF	Reserved	-	-	-	-	-	-	-	-
0xB0	TCCR2A	COM2A1	COM2A0	COM2B1	COM2B0	-	-	WGM21	WGM20
0xB1	TCCR2B	FOC2A	FOC2B	-	-	WGM22	CS22	CS21	CS20
0xB2	TCNT2	Timer/Counter2 (8-bit)							
0xB3	OCR2A	Timer/Counter2 Output Compare Register A							
0xB4	OCR2B	Timer/Counter2 Output Compare Register B							
0xB5	Reserved	-	-	-	-	-	-	-	-
0xB6	ASSR	-	EXCLK	AS2	TCN2UB	OCR2AUB	OCR2BUB	TCR2AUB	TCR2BUB
0xB7	Reserved	-	-	-	-	-	-	-	-
0xB8	TWBR	2-wire Serial Interface Bit Rate Register							
0xB9	TWSR	TWS7	TWS6	TWS5	TWS4	TWS3	-	TWPS1	TWPS0
0xBA	TWAR	TWA6	TWA5	TWA4	TWA3	TWA2	TWA1	TWA0	TWGCE
0xBB	TWDR	2-wire Serial Interface Data Register							
0xBC	TWCR	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE
0xBD	TWAMR	TWAM6	TWAM5	TWAM4	TWAM3	TWAM2	TWAM1	TWAM0	-
0xBE	Reserved	-	-	-	-	-	-	-	-
0xBF	Reserved	-	-	-	-	-	-	-	-
0xC0	UCSR0A	RXC0	TXC0	UDRE0	FE0	DOR0	UPE0	U2X0	MPCM0
0xC1	UCSR0B	RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	UCSZ02	RXB80	TXB80
0xC2	UCSR0C	UMSEL01	UMSEL00	UPM01	UPM00	USBS0	UCSZ01 / UDORD0	UCSZ00 / UCPHA0	UCPOL0
0xC3	Reserved	-	-	-	-	-	-	-	-
0xC4	UBRR0L	USART I/O Data Register							
0xC5	UBBR0H	-	-	-	-	USART Baud Rate Register High			
0xC6	UDR0	USART Baud Rate Register Low							
0xC7–FF	Reserved	-	-	-	-	-	-	-	-

Appendix B

Answers to Selected Problems

Chapter 3
1.
```
/* Ralph Johnson, June 31, 2112 */
// Ralph Johnson, June 31, 2112
```

3.
```
#include <stdio.h>

int main(void)
{
   printf("Ralph Johnson");
}
```

Chapter 4
1.
```
printf("The result is %f volts", output_voltage);
```

3.
```
int x;

x = sizeof(POWER_SUPPLY);
```

5.
```
Y = Y|0x01;
```

7.
```
W = ~W;
```

9. a) 0xf1, b) 0x10, c) 0x0f, d) 0x11

Chapter 7
1.
```
if(X<0)
   printf("negative value");
```

3.
```
// This assumes the variables are integers
if(X>Y)
   printf("%d",X);
else
   printf("%d",Y);
```

5.
```
for(i=0;i<6;i++)
   printf("Error!");
```

7.
```
for(R=100;R<=200;R+=5)
```


Chapter 8
1.
```
float *fptr;
```

3. `c` is an unsigned character, an eight bit (one byte) variable. `p` is a pointer to an unsigned character meaning that it contains the address of a variable which is an unsigned character (such as `c`). `p` could be two, four or eight bytes in size, depending on the operating system.

5. `&` is the "address of" operator. It returns the address of the associated variable. `*` is the dereferencing operator. It returns the value which is at the memory address referenced by the associated variable.

7. This multiplies `b` by the value referenced by `c` (i.e., `c` is a pointer). It may be more clear to show the pointer dereference explicitly:
```
a = b * (*c);
```


Chapter 10
1.
```
// structure definition
struct Quest {
   float X;
   long Y;
   unsigned char Z;
};

// declaration of an instance
struct Quest Grail;
```


Chapter 12
1.
```
char *x;
x = malloc(1000);
```

3.
```
free(x);
free(f);
```

Index

74HC14 127

ADC, 54, 75, 78, 94, 107, 132–140
Arduino home, 4, 8
ARM Coretex, 112
ASCII, 14, 36, 39, 72, 112
auto class variable, 12, 32

Biscotti, 85
Bitwise operators, 27, 29, 41, 92
Boolean, 41, 91
Bus, 12, 80, 100, 104
Byte, 10, 15

C Compiler (free), 8
Cast, 26, 95, 96
`char` keyword, 11, 14, 15, 32, 36
CISC, 78, 80
`const` keyword, 30, 33, 110
CTC mode, 158

DAC, 53, 112, 132, 134, 142, 145
DDR (Data Direction Register), 27–29, 92, 100, 104, 104–112
debounce, 127
`double` (float) keyword, 11, 14, 15, 25, 26, 48, 49, 52
DRAM (Dynamic RAM), 81

EEPROM, 81, 82, 85
`extern` keyword, 30, 34

Flip-Flop, 81, 100, 104
`float` keyword, 15, 25, 26, 30

Global declaration, 11, 18, 21, 33, 34
Global Interrupt Enable, 80, 115, 121
GPIO, 78, 98, 99, 102

Harvard architecture, 12, 78, 79, 85, 110
Header files, 19, 20, 28, 34, 39, 43, 57, 64, 90–96
Hex (hexadecimal), 24, 28, 69, 70

Include files, see *Header files*
`int` keyword (long vs. short), 14, 25, 26
ISR (interrupt service routine), 154–159

LED, 29, 74, 100, 112, 113, 117, 124, 142, 156

Embedded Controllers
Using C and Arduino / 2E
Laboratory Manual
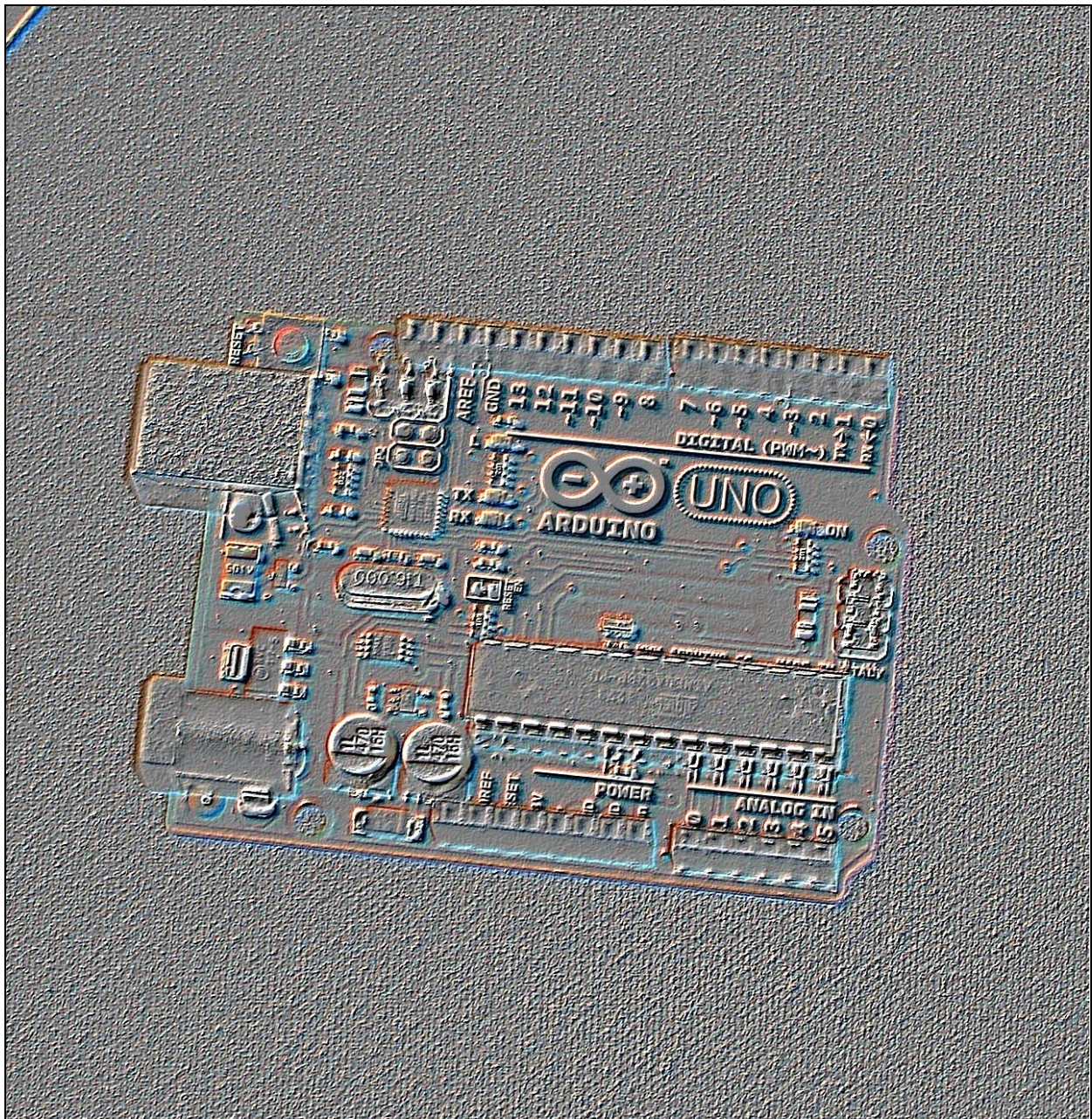
James M. Fiore

Laboratory Manual

for

Embedded Controllers

Using C and Arduino

by

James M. Fiore

Version 2.3.5, 18 April 2020

This **Laboratory Manual for Embedded Controllers Using C and Arduino, by James M. Fiore** is
copyrighted under the terms of a Creative Commons license:

For more information or feedback, contact:

James Fiore, Professor
Electrical Engineering Technology
Mohawk Valley Community College
1101 Sherman Drive
Utica, NY 13501
jfiore@mvcc.edu

For the latest revisions, related titles, and links to low cost print versions, go to:

www.mvcc.edu/jfiore or my mirror sites www.dissidents.com and www.jimfiore.org

YouTube Channel: Electronics with Professor Fiore

Cover art by the author

Introduction

This laboratory manual is intended for use in an introductory microprocessor or embedded controller course and is appropriate for two and four year electrical engineering technology curriculums. It utilizes the C programming language and the inexpensive, open-source Arduino hardware platform, specifically, the Arduino Uno which uses an Atmel ATmega 328P processor. The manual contains sufficient exercises for a typical 15 week course using a two to three hour practicum period. Some exercises may require more than one period (in particular, the arbitrary waveform generator). The first portion deals strictly with an introduction to the C language using standard desktop tools. Any reasonable compiler will do, and many are available free of charge. The second portion (roughly 2/3rds of the total) addresses the Arduino hardware. The Arduino system was chosen because the software is free, open-source, and multi-platform (Windows, Mac and Linux). There are several choices for hardware, most of which are quite inexpensive and open source. Although this manual focuses on the Uno board, other boards may be used with some modifications to the lab text. Interface components are fairly common such as LEDs and seven segment displays, switches, capacitors, diodes, low power resistors and switching transistors such as the 2N3904 or 2N2222. One unique element is an FSR (force sensing resistor), although the circuit may be implemented with an ordinary momentary contact push-button switch. Another item of interest is a small DC hobby motor or boxer fan (and optionally, the ZVN4206A power FET).

Each exercise starts with an overview of the topics to be discussed. This usually includes some code snippets for illustration. A programming application of interest is then developed with a pseudo-code. The pseudo-code is then broken into appropriate C language code chunks. Where appropriate, hardware interface issues are discussed. Finally, the entire package is fit together.

There is a companion OER (Open Educational Resource) text to accompany this laboratory manual. There are also OER texts on Operational Amplifiers and Linear Integrated Circuits, Semiconductor Devices, and DC and AC Electrical Circuits. Each of these has its own laboratory manual. Other laboratory manuals in this series include Computer Programming with Python, and Science of Sound. Please check my web sites for the latest versions.

A Note from the Author

This manual is used at Mohawk Valley Community College in Utica, NY, for our ABET accredited AAS program in Electrical Engineering Technology. A key goal was to keep the student costs low so that students could buy their own development system. Having full "any time" access to the development hardware and software can be very motivating. I am indebted to my students, co-workers and the MVCC family for their support and encouragement of this project. While it would have been possible to seek a traditional publisher for this work, as a long-time supporter and contributor to freeware and shareware computer software, I have decided instead to release this using a Creative Commons non-commercial, share-alike license. I encourage others to make use of this manual for their own work and to build upon it. If you do add to this effort, I would appreciate a notification.

"Resist corporate power"

- jmf

Table of Contents

1

Introduction to C Programming

The main objective of this initial programming exercise is to become familiar with using the programming language tools. The programs in this exercise will be fairly trivial, but serve as a springboard to later work. We will be using programs similar to the ones examined in lecture.

The precise C language package in use is not of extreme importance. Various companies will create different programming tools and although the features and fine points may differ, the basics remain the same. All C language tools need to compile C code and assemble it. Further, they need to link this assembled code with other assembled modules and libraries in order to create a finished executable program. In the simplest case, these tools will be in the form of command line utilities, i.e.; they run from a DOS prompt or shell. Ordinarily, the tools are part of a graphical integrated development environment, or IDE. IDEs normally include a text editor. C compilers expect to work on raw text. Do not attempt to "feed" them the output of a word processor, such as a .doc file. If you are using simple command line tools instead of an IDE, you will create your source files with a basic text editor such as Notepad.

C source code files utilize a ".c" extension. The output of the compiler is called an *object file*. It will normally have a ".o" or ".obj" extension. In the Windows world, finished executables usually have a ".exe" extension. Many IDEs require that you create a *project* before you start entering your code. The project includes many attributes such as the file names used for source code (there may be several in larger projects), the appropriate libraries to link, the name of the finished executable, and so on. For simple programming chores involving small amounts of source code, this can be a bit of a pain, however, it is wonderful for larger endeavors. All the C source code of all of our exercises during this course can easily fit on a single very modestly-sized (64 MB!) USB drive. This includes the project files that can become much larger (many megabytes) than the C source. In networked labs on campus, project files and source files can be saved to the student's network storage area. For those performing labs off-campus, it will probably be easiest to simply create new projects on your hard drive as needed.

This lab will use the Pelles C application. There is nothing magical about Pelles C though and other systems are perfectly acceptable. We shall only be using Pelles C for the introductory exercises anyway. Once we get rolling we shall shift our emphasis to the Arduino development board.

Our first exercise focuses on creating a project, editing source code, compiling and linking it, and testing it. We shall then edit it and repeat the process. We shall also look at error reporting. If you're using command line utilities, see the note at the end of this exercise before continuing.

To begin, open the C language IDE. In Pelles C select "Start a new project" from the start pane. The new project can be one of many things. We will not be creating Windows-GUI programs, but rather DOS shell utilities, so select Win32 or Win64 Console Application (depending on your operating system) and give the project a name. To create C source code, you will need to create a new text file. Select *New>>Source code* under the *File* menu. A blank text edit window will pop open.

Type the following code into the editor:

```
#include <stdio.h>

/* It all begins here */

int main( void )
{
      printf("Hello world!\n");
}
```

Save this as `hello.c`. using *File>>Save as*. A dialog box will pop up asking if you want to add this file to the current project. Select "Yes". Note that in some IDEs you will have to manually insert the file into the project (look for the appropriate menu item in such a case).

While it is possible to separately compile and link modules, most developers use the *Build* shortcut. This will compile and link all files as needed. In Pelles C you can select *Build* from either the *Project* menu or from the toolbar. As the project is built, you will notice messages in the status area at the bottom. If everything works out properly, it will say *Building <name of project> Done*. You can now check the program. Select either the *Execute* toolbar button or *Project>>Execute* from the menu bar. A small DOS shell should pop open with the message "Hello World!". Press any key to clear this shell window and return to the IDE. You have successfully completed your first program!

Depending on the settings for your IDE, you may notice different colors on the text. In many IDEs you can specify different colors for different items such as keywords, constants, math operators, comments, and so forth. This makes reading the code a little easier.

Let's edit the source code and have it do something else. Alter the text so that it looks like this:

```
#include <stdio.h>

/* Program two */

int main( void )
{
      int x, y;

      x = 10;
      y = x + 20;
      printf("The result is %d\n",y);
}
```

Rebuild and test the resulting code. You should get a message that says "The result is 30". Most IDE editors have the usual functionality of cut/copy/paste along with search and replace. Some also have automatic indenting, matching brace creation, and other advanced features.

OK, what happens if you make an error in your code? We shall insert a few on purpose and see what happens. Alter the program above so that it looks like this:

```
#include <stdio.h>

/* Program three, with errors */

int main( void )
{
        int x, y;

        x = 10
        y = x + 20;
        printf(The result is %d\n",y);
}
```

Note that we have left off the trailing semi-colon on `x=10;` as well as the leading quote on the `printf()` function. Rebuild the project. This time you will receive a bunch of errors and warnings. They may differ in wording from development system to development system, but you should see something about a missing semi-colon before the `y`. You'll probably also see an error concerning "The" being an undeclared identifier. You may see many warnings as well. Usually, double clicking on the error message will highlight the corresponding line in the code. Sometimes a single omission can cause the compiler to emit dozens of error messages. This is because the compiler sort of "loses track" of where it is and starts flagging perfectly good code as having errors. For this reason, if you get errors (and you will), always look at the **first** reported error and fix it. Do not look at the last reported error as it may lead you on a wild goose chase.

Finally, you may wish to save your code for backup. Simply select *File>>Save as* and choose an appropriate name. Again, C source files should use a ".c" extension. Note that you can create, read, or edit C source files without the IDE. All you need is a simple text editor. You won't be able to compile or build it, but you can at least get some work done on an assignment without a compiler handy.

For those using a command line system (no IDE), the process is similar to what has been described, although less automatic. You will need to create your source file in a text editor. You then invoke the compiler from a DOS shell window, usually with a command something like:

```
cc hello.c
```

This will create the object file. You then invoke the linker, usually with a command like:

```
ln hello.exe hello.obj stdio.lib
```

You will have to consult your documentation from the proper commands and syntax. Once the executable is created, you test it from the shell by typing its name:

```
hello.exe
```

The appropriate output will be sent to the shell window. To edit the program, reopen the C source file in the text editor, make the changes, save the file, and then repeat compile/link commands. If errors occur, the error messages will be printed in shell window.

2

Using Standard I/O (Input/Output)

In this exercise we will be taking a closer look at the Standard I/O functions `printf()` and `scanf()`. We shall also investigate the use of multiple user functions and prototypes. Further, we shall investigate the importance of proper program specification as well as choosing proper variable types.

This exercise involves creating a program that will assist with a simple series DC circuit analysis given component tolerances. Suppose that we have a circuit consisting of a DC voltage source and a resistor. We are interested in determining the resultant current through the circuit and the power dissipated by the resistor. This is a fairly straightforward exercise involving just Ohm's Law and Power Law. To make this a little more interesting, we will include the effects of resistor tolerance. The program will create a table of currents and powers for the nominal, maximum and minimum allowed resistance values. This would be a very useful program for someone just starting their electrical studies. Let us keep this in mind while we design the program.

When designing a program, try to keep the user interaction as simple and as clear as possible. Also, try to structure the code to facilitate maintenance. Studies have shown that majority of programming time spent on non-trivial applications is in the area of maintenance (adding features, fixing bugs, etc.). Strive to make your code as clear as glass and include appropriate comments. Do not, however, add comments to code that even a novice would understand as that just creates clutter. Here is an example of bad commenting:

```
a = b + c;   /* add b to c */
```

Duh! This comment adds nothing to the quality of the code. Similarly, use mnemonic variable names where possible. This helps to create *self-commenting* code. Here is an example:

```
x = y + z * 60;
total_seconds = seconds + minutes * 60;
```

These two lines of code perform the same mathematical operations, but the second line gives you a hint as to what is intended. The first line would probably need a comment to make sense of it while the second line stands by itself.

The Program

Here is the (first try) specification for the program:

The program will prompt the user for a DC voltage source value, a nominal resistor value and a resistor tolerance. It will then print out the values for current and power dissipation based on the nominal, minimum and maximum acceptable values of the resistor.

Not bad, but we need to refine it. First, command line programs usually need some form of start-up message or print out of directions. Remember these are not GUI-driven programs with Help menus. Second, always prompt for input values *indicating expected units*. If the program expects ohms but the user types in kilo ohms, there's going to be trouble. Unless there is a compelling reason not to, always use base units (ohms versus kilo ohms for example).

Here's our refined specification:

The program will first give appropriate directions/explanations of use to the user. The program will prompt the user for a DC voltage source value in volts, a nominal resistor value in ohms and a resistor tolerance in percent. It will then print out the values for current in amps and power dissipation in watts based on the nominal, minimum and maximum acceptable values of the resistor.

Note that we have specified tolerance as a percentage rather than as a factor. This is because the typical user would be prepared to enter `10` for 10%, not `0.1`. You can use this specification to create a pseudo code or flow chart. Here is a possible pseudo code:

1. Print out directions for user.
2. Prompt user for voltage (in volts) and obtain value.
3. Prompt user for resistance (in ohms) and obtain value.
4. Prompt user for tolerance (in percent) and obtain value.
5. Determine maximum and minimum resistance values.
6. Calculate currents based on the three resistances.
7. Calculate powers based on the three resistances.
8. Print a heading for the values.
9. Print out the values.

You could of course choose an alternate algorithm or method of solution. For example, you might prefer to print the heading before the calculations and then print values following each calculation. You might prefer to change the format so that you get rows for each resistor rather than for the current and power. You might even choose an entirely different approach using loops and/or arrays. There will be upsides and downsides to each approach. Often, the question is not "Can I solve this problem?" but rather "What is the most effective way of solving this problem?" Extend a little forethought before you begin coding.

Based on the above pseudo code, the following program should fit the bill. We will refine it later. Note the use of `double` as we will most likely have fractional values with which to deal.

```c
#include <stdio.h>

int main( void )
{
      double v, tol;
      double rnom, rlow, rhigh;
      double inom, ilow, ihigh;
      double pnom, plow, phigh;

      printf("This program determines current and power.\n");

      printf("Please enter the voltage source in volts.\n");
      scanf("%lf", &v);
      printf("Please enter the nominal resistance in ohms.\n");
      scanf("%lf", &rnom);
      printf("Please enter the resistor tolerance in percent.\n");
      scanf("%lf", &tol);

      tol = tol/100.0;  /* turn tolerance into a factor */
      rlow = rnom - rnom*tol;
      rhigh = rnom + rnom*tol;
      inom = v/rnom;
      ihigh = v/rlow;
      ilow = v/rhigh;

      pnom = v * inom;
      plow = v * ilow;
      phigh = v * ihigh;

      printf("Resistance (ohms)  Current (amps)  Power (watts)\n");
      printf("%lf     %lf      %lf\n", rnom, inom, pnom );
      printf("%lf     %lf      %lf\n", rhigh, ilow, plow );
      printf("%lf     %lf      %lf\n", rlow, ihigh, phigh );
}
```

A word of caution here: Note that the variable `ihigh` is the highest current, not the current associated with the highest resistor. This can make the print out code seem incorrect. This is a good place for some comments! Also, the initial "directions" are skimpy at best. In any case, enter and build the code above and verify that it works.

You may have noticed that there is a bit of repetition in this code in the form of calculations and printouts. It may be more convenient if we created functions to handle these. For example, we could create a function to calculate the current:

```c
double find_current( double voltage, double resistance )
{
      double current;

      current = voltage/resistance;
      return( current );
}
```

You could also do this in one step:

```
double find_current( double voltage, double resistance )
{
        return( voltage/resistance );
}
```

Updating the program produces the following:

```
#include <stdio.h>

double find_current( double voltage, double resistance )
{
        return( voltage/resistance );
}

int main( void )
{
        double v, tol;
        double rnom, rlow, rhigh;
        double inom, ilow, ihigh;
        double pnom, plow, phigh;

        printf("This program determines current and power.\n");

        printf("Please enter the voltage source in volts.\n");
        scanf("%lf", &v);
        printf("Please enter the nominal resistance in ohms.\n");
        scanf("%lf", &rnom);
        printf("Please enter the resistor tolerance in percent.\n");
        scanf("%lf", &tol);

        tol = tol/100.0;  /* turn tolerance into a factor */
        rlow = rnom - rnom*tol;
        rhigh = rnom + rnom*tol;

        inom = find_current( v, rnom );
        ihigh = find_current( v, rlow );
        ilow = find_current( v, rhigh );

        pnom = v * inom;
        plow = v * ilow;
        phigh = v * ihigh;

        printf("Resistance (ohms)  Current (amps)  Power (watts)\n");
        printf("%lf      %lf       %lf\n", rnom, inom, pnom );
        printf("%lf      %lf       %lf\n", rhigh, ilow, plow );
        printf("%lf      %lf       %lf\n", rlow, ihigh, phigh );
}
```

This doesn't seem to be much of an improvement. In fact, it just seems longer! This is true, but extend the idea a moment. What if the calculation for current involved a dozen lines of code instead of just one? This new format would save considerable code space. Note that this is not just a matter of saving some typing, but rather in saving memory used by the executable. This is particularly important when using constrained embedded systems with only a small amount of available memory.

Note that the new function was added before `main()`. This is not required. We could also have added it after `main()`, but in that case we'd have to add a function prototype so that the compiler would know what to expect when it saw the function call in `main()`. It would look something like this:

```
#include <stdio.h>

/* this is the prototype so the compiler can do type checking */

double find_current( double voltage, double resistance );

int main( void )
{
        ....
}

double find_current( double voltage, double resistance )
{
        return( voltage/resistance );
}
```

Alter the program to use this new current calculation function and test it. Once this is complete, alter the program one more time to use a function to calculate the power and another to print out the three values. Use the current calculation function as a guide. Test this with the following values: 12 volt source with a 100 ohm, 5% resistor. Finally, consider what might go wrong with the program. What would happen if we the user entered `0` for the resistor value? How could you get around that problem?

3
Using Conditionals

In this exercise we will be taking a look at using conditionals. These include the `if/else` construct and the `switch/case` construct. Conditionals are used to make a choice, that is, to split the program flow into various paths. The `if/else` works best with simple either/or choices while the `switch/case` is designed to deal with one (or possibly multiple) choice(s) from a list of fixed possibilities.

The simplest conditional is the straight `if()`. Depending on whether or not the item(s) to be tested evaluate to true determines if subsequent action is taken. By grouping clauses with the logical operators `||` and `&&`, complex tests may be created. `if()` statements may be nested as well as include processing for the test failure by using the `else` clause. If you need to choose a single item from a list, for example when processing a menu selection, this may be achieved through nesting in the following fashion:

```
if( choice == 1 )
{
      /* do stuff for 1 */
}
else
{
      if( choice == 2 )
      {
            /* do stuff for 2 */
      }
      else
      {
            if( choice == 3 )
            {
                  /* do stuff for 3 */
            }
            /* and so on */
      }
}
```

This arrangement is a bit cumbersome when choosing from a large list. Also, it is difficult to deal with a plural choice (e.g., picking items 1 and 3). For these situations the C language offers the `switch/case` construct. There is nothing a `switch/case` can do that you can't recreate with nested `if/else`'s and additional code, but the former offers greater convenience as well as clearer and more compact code. The `switch/case` is used frequently, but ultimately, the `if/else` is more flexible because it is not limited to choosing from a list of numeric values. Ordinarily, numeric constants are not used in production code, as in the example above. Instead, `#define`'s are used for the constants in order to make the code more readable. A real-world `switch/case` version of the previous example might look like:

```
#define WALK_DOG        1
#define LET_OUT_CAT     2
#define COMB_WOMBAT     3

switch( choice )
{
      case WALK_DOG:
            /* c'mon poochie... */
            break;

      case LET_OUT_CAT:
            /* there's the door... */
            break;

      case COMB_WOMBAT:
            /* first the shampoo... */
            break;

      /* and so on */
}
```

In this exercise we're going to make use of both constructs. The program will involve the calculation of DC bias parameters for simple transistor circuits. We shall give the user a choice of three different biasing arrangements (voltage divider, two-supply emitter, and collector feedback). The program will then ask for the appropriate component values and determine the quiescent collector current and collector-emitter voltage. It will also determine whether or not the circuit is in saturation. These values will be displayed to the user.

One approach to this problem is to consider it as three little problems joined together. That is, consider what you need to do for one bias and then replicate it with appropriate changes for the other two. The three are then tied together with some simple menu processing. Here is a pseudo code:

1. Give the user appropriate directions and a list of bias choices.
2. Ask the user for their bias choice.
3. Branch to the appropriate routine for the chosen bias. For each bias,
        a. Ask for the needed component values (resistors, power supply, beta).
        b. Compute Ic and Vce and determine if the circuit is in saturation.
        c. Display values to the user.

The appropriate equations for each bias follow. All biases use the following: Vcc is the positive supply. Re is the emitter resistor while Rc is the collector resistor. beta is the current gain (hfe). The base-emitter (Vbe) may be assumed to be 0.7 volts. Note that if Ic-saturation is greater than Ic, then the actual Ic is equal to Ic-saturation and Vce will be 0.

Voltage Divider: also requires R1, R2 (upper and lower divider resistors).

Vth = Vcc*R2/(R1+R2)
Rth = R1*R2/(R1+R2)
Ic = (Vth-Vbe)/(Re+Rth/beta)
Vce = Vcc-Ic*(Re+Rc)
Ic-saturation = Vcc/(Rc+Re)

Collector Feedback: also requires Rb (base resistor).

Ic = (Vcc-Vbe)/(Re+Rc+Rb/beta)
Vce = Vcc-Ic*(Re+Rc)
Ic-saturation = Vcc/(Rc+Re)

Two-supply Emitter: also requires Vee (negative emitter supply) and Rb (base resistor).

Ic = (Vee-Vbe)/(Re+Rb/beta)
Vce = Vee+Vcc-Ic*(Re+Rc)
Ic-saturation = (Vee+Vcc)/(Rc+Re)

where Vee is an absolute value in all cases.

The Program

The program is presented in chunks, below, in the sequence you might write it. First comes the main skeleton.

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

#define VOLTAGE_DIVIDER       1
#define EMITTER               2
#define COLLECTOR_FEEDBACK    3
#define VBE .7

int main( void )
{
      int choice;

      give_directions();
      choice = get_choice();

      switch( choice )
      {
            case VOLTAGE_DIVIDER:
                  voltage_divider();
                  break;

            case EMITTER:
                  emitter();
                  break;

            case COLLECTOR_FEEDBACK:
                  collector_feedback();
                  break;

            default:    /* tell user they're not so bright... */
                  printf("No such choice!\n");
                  break;
      }
}
```

The first two functions might look something like this (don't forget to add their prototypes later):

```c
void give_directions( void )
{
      printf("DC Bias Q Point calculator\n\n");
      printf("These are your bias choices:\n");
      printf("1. Voltage Divider\n");
      printf("2. Two Supply Emitter\n");
      printf("3. Collector Feedback\n");
}

int get_choice( void )
{
      int ch;

      printf("Enter your choice number:");
      scanf("%d", &ch);
      return( ch );
}
```

Now it's time to write the bias functions. Here is how the `voltage_divider()` function might look:

```c
void voltage_divider( void )
{
      double vcc, vth, r1, r2, rth, re, rc, beta, ic, icsat, vce;

      printf("Enter collector supply in volts");
      scanf("%lf", &vcc);
      printf("Enter current gain (beta or hfe)");
      scanf("%lf", &beta);
      printf("Please enter all resistors in ohms\n");
      printf("Enter upper divider resistor");
      scanf("%lf", &r1);
      printf("Enter lower divider resistor");
      scanf("%lf", &r2);
      printf("Enter collector resistor");
      scanf("%lf", &rc);
      printf("Enter emitter resistor");
      scanf("%lf", &re);

      vth = vcc*r2/(r1+r2);
      rth = r1*r2/(r1+r2);
      ic = (vth-VBE)/(re+rth/beta);
      icsat = vcc/(rc+re);

      if( ic >= icsat )
      {
            printf("Circuit is in saturation!\n");
            printf("Ic = %lf amps and Vce = 0 volts\n", icsat );
      }
      else
      {
            vce = vcc-ic*(re+rc);
            printf("Ic = %lf amps and Vce = %lf volts\n", ic, vce );
      }
}
```

The other two bias functions would be similar to this. A few points to note: In order to obtain the absolute value, consider using the `fabs()` function (floating point absolute). Another approach to the program would be make the `vce`, `icsat`, and `ic` variables globals and move the printout section to `main()` because the final comparison and printout will be the identical in all three functions. (It is also possible to have the functions return the values via pointers, avoiding the need for globals.)

Complete the other two bias functions, build, and test the program. Use the following values: Trial #1, Two supply emitter bias, Vcc=20 V, Vee=−10 V, Rb=2 kΩ, Re=1 kΩ, Rc=1.5 kΩ, beta=100. Trial #2, Collector feedback bias, Vcc=30 V, Rc=10 kΩ, Re=1 kΩ, Rb=100 kΩ, beta=100. To avoid compiler warnings, you will need to place function prototypes before `main()`. You could place these in a separate header file, but there are too few to bother with. To create the prototypes, simply copy and paste the function declarations and add a trailing semi-colon. For example:

```
#define VBE .7

void give_directions( void );
int get_choice( void );
/* and so forth */

void main( void )
{
        ...
```

Without the prototypes, the compiler won't "know" what the functions take as arguments nor what sort of variables they return (if any). Thus, the compiler can't do type checking and will warn you about this. Also, the compiler will assume default argument and return types of `int`, so when the compiler sees the function code, it will complain that the types don't match the default. This might seem like a pain at first, but it is a cross-checking mechanism that can prevent many software bugs.

Alterations

This program might be useful to a student studying transistors, but it may be a little cumbersome to use. After all, the program needs to be restarted for every circuit. How would you alter the program so that it would ask whether or not the user wanted to try another circuit? In other words, the user could start the program and run it continually for 100 circuits if they so wished.

Perhaps more interestingly, what would be required so that the program could be used for homework generation? That is, the program would create appropriate circuits using new component values each time, and inform the user. It would then ask user to calculate the current and voltage by hand and enter them. The program would then determine if the user's answers were correct (within a certain percentage to compensate for round off error). Another possibility would be to present the user with a multiple choice list of possible answers instead of having them enter precise values.

Lastly, how might you attempt to "draw" the circuits so that the components are visually identified?

4

Using Loops

In this exercise we will be taking a look at using loops, or iteration. We shall also investigate a few functions in the standard library. Computers are of course ideal for repetitive calculations such as those needed to create data tables.

If a specific number of iterations are required, usually the `for()` loop construct is the best choice. For situations where the precise number of iterations is not known, or where the termination condition can be expressed in an easier fashion, the `while()` loop construct is preferred. This exercise shall focus on using the `while()` construct, although it is recommended that upon completion you attempt to alter the code to utilize the `for()` construct.

This program will be used to compute a table of values for a series RC circuit. It will compute and display the values that might be used for *Bode plot*. This plot shows the gain or attenuation of a network with respect to frequency. In this example we shall use the voltage across the capacitor as the output quantity. The gain is normally displayed in *decibel* form (or *dB* for short) rather than as an ordinary factor. This program won't draw the curve, but rather list the values so that you could draw the curve yourself.

First, the program will need to obtain the resistor and capacitor values from the user. From these it will compute and display the critical frequency, f_c. The user will then be asked to specify a range of frequencies over which to compute the gain values. This will consist of a start frequency, an end frequency and the number of frequency points to calculate per decade (a decade being a factor of 10). Normally the frequencies would not be equally spaced in hertz, but rather, equally spaced by a factor. In other words, the ratio of any two adjacent frequencies would be the same rather than the number of hertz between them. For example, if we started at 100 Hz, finished at 1000 Hz and only wanted 2 points per decade, we wouldn't use 100, 550, and 1000 (1000 starts a new decade, so only count the 100 and 550 as the "two points per decade"). 550 is equidistant between 100 and 1000, but the ratios are way off: It's 5.5 times larger than the starting point but not even a factor of 2 smaller than the ending point. In fact, we want the sequence 100, 316, 1000 because the ratio of 100 to 316 is 3.16 and the ratio of 316 to 1000 is also 3.16. How do we know to use this ratio? Simple! Think of the specification "two points per decade". You want a number that when multiplied by itself yields 10, i.e. the square root of 10 (3.16). If you wanted 5 points per decade you'd use the 5th root of 10 and so on. Here's a pseudo-code to start with:

1. Give user appropriate directions
2. Obtain resistance value in ohms and capacitance value in farads from user.
3. Compute and print critical frequency.
4. Obtain start and stop frequencies (in Hz) along with number of points per decade from user.
5. Print out table heading consisting of Frequency (Hz) and Gain (dB).
6. Determine frequency factor based on points per decade.
7. Initialize frequency to start value.
8. Start a loop that continues as long as frequency is less than or equal to the stop frequency.
9. Compute gain in dB.
10. Print out frequency and dB gain.
11. Multiply frequency by frequency factor to obtain next frequency of interest.
12. End of loop, and of program.

The only thing that might remain a bit fuzzy is the computation of gain and its conversion to dB. The conversion to dB is defined by the following equation:

$$dB = 20 * \log_{10}(\text{ gain })$$

The gain is basically a voltage divider between the resistor and the capacitive reactance, so let's refine step 9:

9a. Determine capacitive reactance Xc.
9b. Calculate gain based on the vector voltage divider: gain = $-j$Xc/(R$-j$Xc).
    Reminder: The magnitude of this is $Xc/\sqrt{(R^2 + Xc^2)}$.
9c. Calculate dB based on dB $= 20 * \log_{10}(\text{ gain })$

The math library (math.h) will come in handy for common log, roots, and so forth. The functions of interest are `pow()`, `log10()`, and `sqrt()`.

The Program

First, notice the use of `#define` for pi and function prototypes. The functions, though small, make the main section much more readable.

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


#define M_PI 3.141592653


void give_directions( void );
double find_fc( double res, double cap );
double find_xc( double freq, double cap );
double find_dB( double gain );
```

```c
int main( void )
{
        double r, c, xc, gain, dB, steps;
        double fc, f, fstart, fstop, ffactor;

        give_directions();

        printf("Enter the resistance in ohms:");
        scanf("%lf", &r);
        printf("Enter the capacitance in farads:");
        scanf("%lf", &c);

        fc = find_fc( r, c );

        printf("\nThe critical frequency is %lf hertz.\n\n", fc);

        printf("Enter the start frequency in hertz:");
        scanf("%lf", &fstart);
        printf("Enter the stop frequency in hertz:");
        scanf("%lf", &fstop);
        printf("Enter the number of steps per decade to display:");
        scanf("%lf", &steps);

        printf("Frequency (Hz)\t\t\tGain (dB)\n"); /* \t is a tab */

        ffactor = pow( 10.0, 1.0/steps );

        f = fstart;

        while( f <= fstop )
        {
                xc = find_xc( f, c );
                gain = xc/sqrt(r*r + xc*xc); /* could use pow() for square here,
                                              but mult by self executes faster */

                dB = find_dB( gain );
                printf("%10.1lf\t\t%10.1lf\n", f, dB ); /* %10.1lf is 10 spaces
                                                      with 1 digit after decimal */

                f *= ffactor; /* shortcut for f=f*ffactor; */
        }
}


void give_directions( void )
{
        printf("Bode Table Generator\n\n");
        printf("This program will display dB gains for a simple RC circuit\n");
}


double find_fc( double res, double cap )
{
        return( 1.0/(2.0*M_PI*res*cap) );
}


double find_xc( double freq, double cap )
{
        return( 1.0/(2.0*M_PI*freq*cap) );
}
```

```
double find_dB( double gain )
{
        return( 20.0 * log10( gain ) );
}
```

Enter this program and test it using R=1 kΩ, C=100 nF, start frequency=100 Hz, stop frequency=20 kHz, points per decade=8. Consider what might go wrong with this program and how you might circumvent those problems. For example, consider what might happen if the user entered `0` for the resistor value, or a stop frequency that was less than the start frequency.

Alterations and Extensions

There are two useful extensions once you have completed this exercise. The first, as mentioned earlier, is to re-code the loop using the `for()` construct. The second alteration is to add a third column to the table that shows the phase shift at each frequency. A true Bode plot is in fact two plots: One of gain magnitude versus frequency and one of gain phase versus frequency.

5

Introduction to Addresses, Pointers and Handles

All variables and functions have an address (memory location) associated with them. This is where they "live". For multiple byte entities, this is the starting address of the memory block the item occupies. The address of any variable (with the exception of `register` class) can be found with the `&` operator. Depending on the operating system, the address will generally be contained within either 2 bytes or 4 bytes in length (8 bytes for a 64 bit OS). That is, all possible addresses in the system can be so described. If an address is placed into another variable, this other variable is said to be a pointer (i.e., it stores the address of, or "points to", the first variable). In order to take advantage of type checking and pointer math, pointers are declared by the type of item they point to, **even though all pointers are themselves the same size.** Thus, a pointer is properly declared as a pointer to a type `char` or a pointer to a type `float`, for example. The `*` operator is used to indicate that the declaration is of a pointer. For example:

```
int *pc;
```

declares a pointer to an `int`.

```
int c, *pc;

c = 12;
pc = &c;
```

This chunk of code declares an `int` and a pointer to an `int`, assigns 12 to the `int` `(c)`, and then places the address of the `int` `(c)` in the pointer `(pc)`. The value of the item pointed to can be retrieved through the indirection operator `*`. Consider the following addition to the above:

```
printf("The value of c = %d, the address of c = %d\n", c, &c );
printf("The value of pc = %d, the value pc points to = %d\n", pc, *pc );
```

Note that `c` and `*pc` are the same value (12). Also, note that if the value of `c` is changed, `*pc` reflects this change. Thus, adding

```
c = 36;
printf("New value of c = %d, the address of c = %d\n", c, &c );
printf("The value of pc = %d, the value pc points to = %d\n", pc, *pc );
```

will show 36 for `*pc` as well as `c`, and the addresses will not have changed.

Exercise one: Create a small program based on the above code and run it. Compare your results to those of your fellow lab workers. What do you notice?

What if the address of the pointer is stored in another pointer? This is called a *handle*. Consider the following code fragment:

```
int c, *pc, **ppc;

c = 12;
pc = &c;
ppc = &pc;

printf("pc = %d, the value pc points to = %d\n", pc, *pc );
printf("ppc = %d, the value ppc points to = %d\n", ppc, *ppc );
```

Exercise Two: Alter your program to reflect the above. Run it and compare your results. What do you think the value `**ppc` is? (Try it).

Addresses can be sent as arguments to functions. *This is how a function can return more than one value and is an important concept to remember.* Consider the following:

```
int main( void )
{
      int a, b, c, *pc;


      pc = &c;

      assign_it( &a, &b, pc );

      printf("The values in main are: %d %d %d\n", a, b, c );
}


void  assign_it( int *x, int *y, int *z )
{
      *x = 1;
      *y = 20;
      *z = 300;
}
```

Note that `assign_it()` can be called using either the address of operator `(&)` on an existing variable, or by passing a pointer to a variable (as in the case of `&c` or pc). Also, `assign_it()` s declaration states that it is accepting *pointers to* `int`, not plain old `int`.

Exercise Three: Using the above as a guide, create a program to test returning multiple values from a function.

<u>Exercise Four:</u> Modify the above to print out the addresses of the variables as received by `assign_it()`, as well as `&a`, `&b`, and `pc` back in `main()`. What does this show?

Note: If you are unsure of the size of the pointers in a given operating system, simply use the `sizeof()` operator. This will return the size of the given item in bytes. For example, the code fragment,

```
int x, *pc;

x = sizeof( pc );
printf("Pointers are %d bytes\n", x );
```

may print 2 (Windows 3.1, Eeek!), 4 (Windows 95/XP, some UNIX systems) or 8 (true 64 bit operating systems). Note that `pc` could be a pointer to a `float`, `double`, `char` or anything, and it will still be the same size. **This is a very important point to understand**. Try using `sizeof()` to verify this.

6

Hello Arduino

Installation

One reason the Arduino system was chosen for this course is because of its low cost. The software is free and development boards are available for under $30. If you have already installed the Arduino software on your computer or do not intend to, skip forward to the next section entitled "IDE".

To install, first simply go to http://arduino.cc/en/Main/Software and download the current version of the software for your operating system. It is available for Windows, Mac and Linux. On Windows, the software is packed into a zip file. This will need to be unzipped with a program such as WinZip. Once unzipped, simply run the setup executable file. This will install almost all of the files needed in short order. There is only one glitch, and that involves installing the driver for your board. Once the software is loaded, plug your Arduino board into your PC via a USB A to USB B cable. In a moment Windows will inform you that it found new hardware and is looking for a driver for it. If you select to see "details", Windows will pop up something like this (Windows 7):
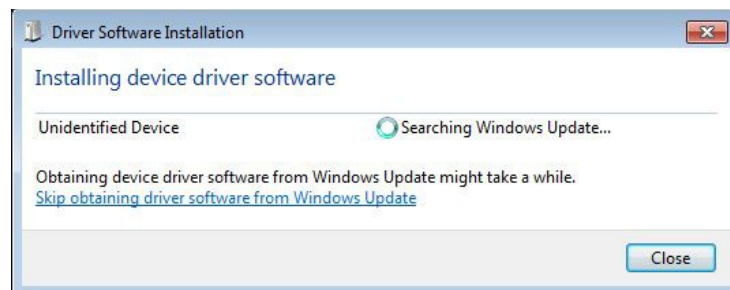

The glitch is that Windows won't be able to find the driver and you'll be greeted with something like this:
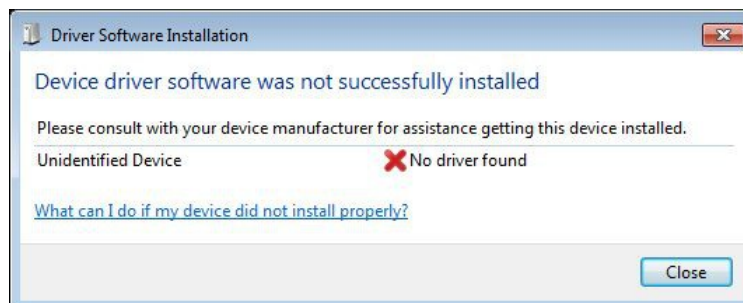
Not to worry. You can install the driver manually. The precise process varies a little from XP to Vista to 7 but goes generally as follows. First, open the Device Manager (via Control Panel>>System>>Hardware). Scroll down to "Ports". You should see entry like "Arduino Uno R3 (COM1)". (You might instead find it under "Unknown Devices" instead of "Ports".) Select this and go to the Driver tab. Click on Update/Install Driver. Browse to find your driver. It will be located in the "drivers" directory of the Arduino software install directory. For the Uno, it will be named "Arduino Uno R3.inf". Select it and let Windows install it. If you have difficulty, a step-by-step guide is available here: http://arduino.cc/en/Guide/Windows Also, a step-by-step installation guide is available at http://www.robotc.net/wikiarchive/ARDUINO complete with Windows pics.

IDE

Once the board is installed, it's time to open the Arduino IDE. It is fairly simple when compared to larger desktop C language development systems.
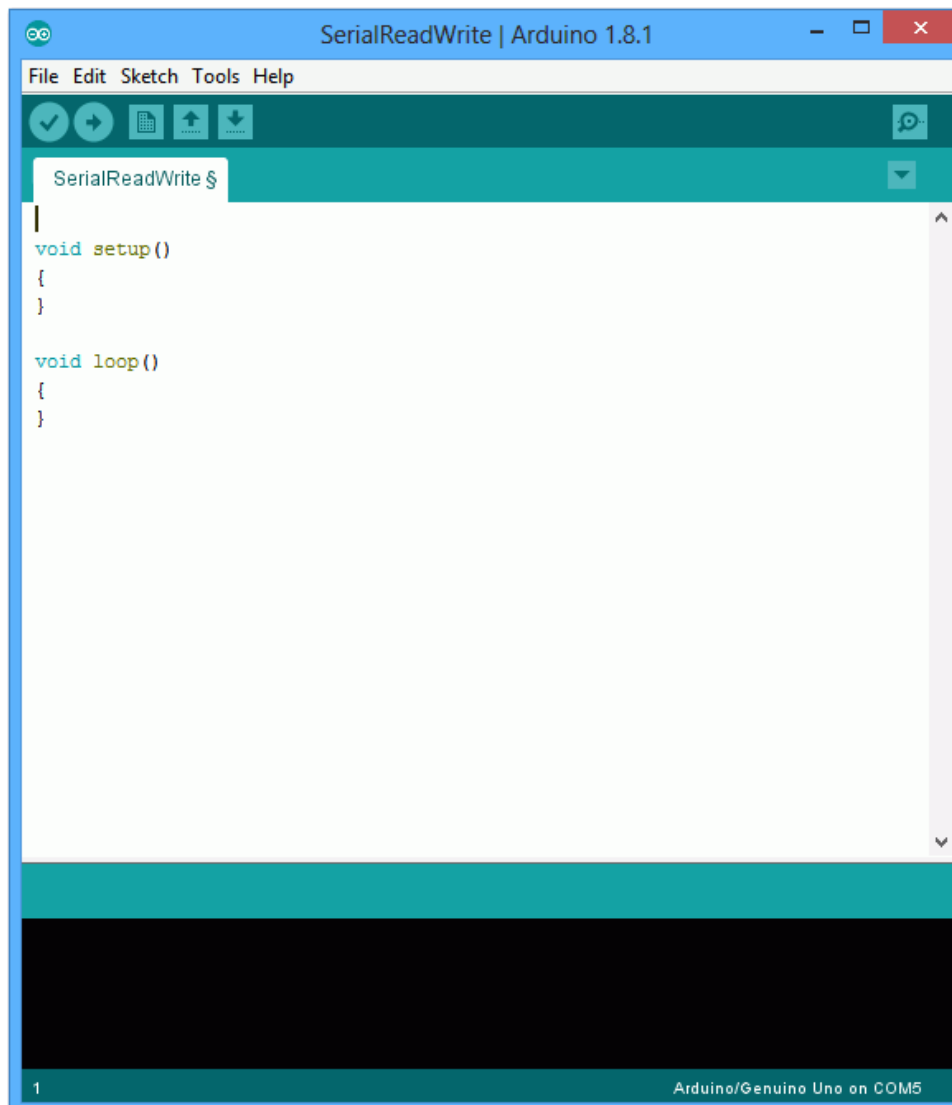
You will see a single edit space with the usual cut-paste-copy facilities. The black area below the edit space is for messages from the compiler/downloader. That is, this is where compiler errors will show up along with status messages. The File and Edit menu content are fairly typical. The Tools menu is particularly important.
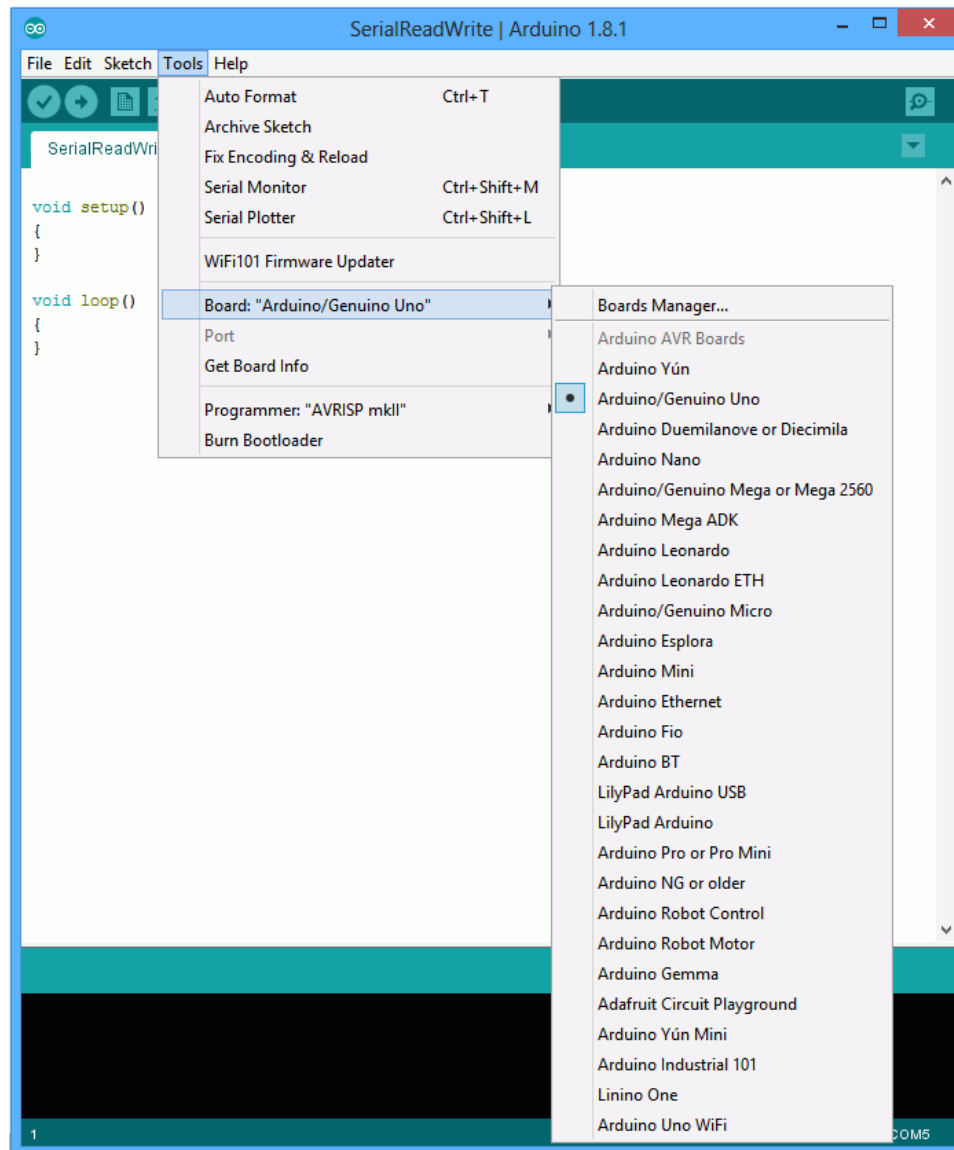
The two most important items here are Board (from which you select your particular board) and Serial Port (from which you select the COM port to which your board is connected). Failure to set these correctly will result in unsuccessful programming of your board. Note that the current board and COM port selections are printed at the lower right corner of the IDE window.

There is a simple toolbar below the menus. The first item (checkmark) is called "Verify" and compiles your code. The second item (right arrow) downloads the code to the target board. The other items open and save files.

Coding

Ordinarily, all code will be contained within a single file. When you create a new project, or "sketch" in Arduino, a new directory will be created for it. Unlike ordinary C language source files, Arduino sketches utilize a ".ino" extension instead of ".c". Further, the normal `main()` entry point is not used. In essence, the Arduino system has already written `main()` for you and it looks something like this:

```
void main()
{
      init();

      setup();

      while(1)
            loop();
}
```

The first function call is where various system initializations occur such as allocating and presetting timers, the ADC system, etc. This has already been written for you. The other two calls, `setup()` and `loop()`, are for you to write. `setup()` is your own initialization function (i.e., things you need to do just once at start-up) and `loop()` is the portion that will be called repeatedly. So you usually start coding with something that looks like the first figure, above.

One of the issues with embedded programming is that you don't have a console for input and output via `scanf()` and `printf()`. Normally, an embedded program won't need these but they are very useful when it comes to debugging a program. Without `printf()`, how can you insert test points in your code to check progress, values, etc? One technique is to simply flash an LED at specific points in the program. The Arduino boards include a small LED on one of the ports for this purpose. While this is useful, it is limited, so a second and more powerful technique involves two-way communication with the host computer. On the Arduino, this is done via the Serial library. The IDE includes a *serial monitor* under the tools menu. Selecting this will open a small window. You can use it to send data to the board or send data from the board back to the host computer.

Let's look at an example of communicating via the Serial Monitor. First, let's consider write data from the board back to the host. Here's a "Hello World" program.

```
void setup()
{
      Serial.begin(9600);
}

void loop()
{
      Serial.print("Hello World\n");
      delay( 1000 );
}
```

In the setup function we open the serial library and set the communication speed at 9600 baud (more or less standard speed for the Arduino) using the `Serial.begin()` function. There is a complimentary `Serial.end()` function should you decide that you need the particular port pins that are used for serial communication for other purposes later on. Note that all of the calls to functions in this library will start with the library's name. The `loop()` functions writes our message using `Serial.print()` and then

waits for about one second using the `delay()` function, the argument of which is measured in milliseconds. Once that's done program flow returns back to `main()`. Of course, `main()` just repeats on the `loop()` function so it calls it again. Our message is printed a second time, a third time and so on.

Enter the bit of code above, compile and download it to the target. Once that's done, open the Serial Monitor and look at the output. Each second you should see the message "Hello World" printed again. That's all the controller program does. Not too useful by itself here, but the power to print items back to the host will be very useful once we get going.

Get used to using the on-line documentation. Two good places to start are the Reference page at http://arduino.cc/en/Reference/HomePage and the Tutorial page at http://arduino.cc/en/Tutorial/HomePage The Reference page includes information on all of the operators and functions built into the Arduino system. The Tutorial gives information on programming concepts and ideas. For example, here is a copy of the on-line documentation of the `Serial.print()` function:

Serial.print()

Description

Prints data to the serial port as human-readable ASCII text. This command can take many forms. Numbers are printed using an ASCII character for each digit. Floats are similarly printed as ASCII digits, defaulting to two decimal places. Bytes are sent as a single character. Characters and strings are sent as is. For example:

- Serial.print(78) gives "78"

- Serial.print(1.23456) gives "1.23"

- Serial.print('N') gives "N"

- Serial.print("Hello world.") gives "Hello world."

An optional second parameter specifies the base (format) to use; permitted values are BIN (binary, or base 2), OCT (octal, or base 8), DEC (decimal, or base 10), HEX (hexadecimal, or base 16). For floating point numbers, this parameter specifies the number of decimal places to use. For example:

- Serial.print(78, BIN) gives "1001110"

- Serial.print(78, OCT) gives "116"

- Serial.print(78, DEC) gives "78"

- Serial.print(78, HEX) gives "4E"

- Serial.println(1.23456, 0) gives "1"

- Serial.println(1.23456, 2) gives "1.23"

- Serial.println(1.23456, 4) gives "1.2346"

You can pass flash-memory based strings to Serial.print() by wrapping them with F(). For example :

- Serial.print(F("Hello World"))

To send a single byte, use Serial.write().

Syntax

Serial.print(val)
Serial.print(val, format)

Parameters

val: the value to print - any data type

format: specifies the number base (for integral data types) or number of decimal places (for floating point types)

Returns

size_t (long): print() returns the number of bytes written, though reading that number is optional

/**** end of copied reference material ****/

The cool thing about this function is the optional second argument that allows you to specify format information. The ability to print out values in hex and binary are particularly useful, for example, when examining bit fields. Modify your program so that it appears as follows, compile, download and run it:

```
void setup()
{
      int i = 27; // try different values for this

      Serial.begin(9600);

      // println is just print with an added newline character
      Serial.println(i, DEC);
      Serial.println(i, BIN);
      Serial.println(i, HEX);
}

void loop()
{
      // nothing to do here
}
```

Less common but still useful from time to time when debugging is the ability to pass values into the running program. The Serial Monitor window includes a small data entry field at the top with an associated Send button. The Serial functions of greatest interest are `Serial.available()`, `Serial.parseFloat()` and `Serial.parseInt()`:

Serial.available()

Description

Get the number of bytes (characters) available for reading from the serial port. This is data that's already arrived and stored in the serial receive buffer (which holds 64 bytes). available() inherits from the Stream utility class.

Syntax

Serial.available()

Parameters

Returns

the number of bytes available to read

Serial.parseFloat()

Description

Serial.parseFloat() returns the first valid floating point number from the Serial buffer. Characters that are not digits (or the minus sign) are skipped. parseFloat() is terminated by the first character that is not a floating point number.

Serial.parseFloat() inherits from the Stream utility class.

Syntax

Serial.parseFloat()

Parameters

Returns

float

/**** end of copied reference material ****/

`Serial.parseInt()` is similar to `Serial.parseFloat()` but returns an integer rather than a floating point value. The two parsing functions examine the input string and begin by discarding non-numeral characters. Once a numeral is found, it continues to look at characters in the string until another non-numeral is found. It then translates the ASCII numeral bytes into a single integer value (or float, as the case may be). Note that it is possible to send multiple values at the same time.

Here's an example of how they might be used to create a decimal to hex converter. Enter the following chunk of code, compile and download it.

```
int i1, i2; // globals to hold the data

void setup()
{
     Serial.begin(9600);
}

void loop()
{

     // Anything in the buffer?
     if (Serial.available() > 0)
     {
          i1 = Serial.parseInt();
          i2 = Serial.parseInt();

          // Echo back what you got, but in hex
          Serial.println("In hex, that is: ");
          Serial.println(i1, HEX);
          Serial.println(i2, HEX);
       }
}
```

Once the program has been downloaded, open the Serial Monitor and enter 10 27 into the text box at the top and then hit the Send button next to it (you could also hit the Enter key on your keyboard). The result should be:

```
In hex, that is:
A
1B
```

You can try other values as well. If you enter only one value, i2 will come back as 0. If you enter several values, multiple returns will follow. Try them both before proceeding.

If you enter a negative value, something interesting will happen. Try typing in -1 -2. What do you see and why?

Now that you have some basic knowledge of the IDE and how to perform simple text debugging, we can turn to examining how to read from and write to external circuitry.

7

Arduino Digital Output

In this exercise we shall examine basic digital output using both the Arduino code libraries and more direct "bit twiddling". The target of this effort will be controlling an LED, including appropriate driver circuits.

First, let's use a modified version of the example Blink program. This blinks the on board test LED connected to Arduino pin 13 at a rate of two seconds on, one second off. We will add serial text output to verify functionality.

Type the following code into the editor:

```
/*
      MyBlink
      Turns on an LED then off, repeatedly.
      With serial output
*/

// Pin 13 has an LED connected on most Arduino boards. Give it a name:
#define LEDPIN 13

void setup()
{
      // initialize the digital pin as an output.
      pinMode( LEDPIN, OUTPUT );
      // initialize serial communication at 9600 bits per second:
      Serial.begin( 9600 );
}

void loop()
{
      Serial.println("On");
      digitalWrite( LEDPIN, HIGH ); // turn the LED on
      delay(2000);                  // wait for 2 secs

      Serial.println("Off");
      digitalWrite( LEDPIN, LOW );  // turn the LED off
      delay(1000);                  // wait for 1 sec
}
```

Compile the code and transfer it to the board. Open the Serial Monitor. You should see messages on the Serial Monitor reflecting the state of the on-board LED.

The code is more or less self-explanatory but a few details may be in order. First, pin 13 on the Arduino Uno's digital header is connected to a small surface mount LED. The pin is connected to a current limiting resistor which is connected to the LED's anode, the cathode of which returns to ground. Thus, a high voltage turns on the LED. The `setup` routine sets this pin for output and establishes the serial communication system. The `loop` routine uses `digitalWrite()` to alternately send this pin high or low, thus turning the LED on and off. The `delay` function is used to hold the levels for a moment so that they can be seen easily with the naked eye. The `delay` function works well enough here but it is a simple busy-wait loop meaning that no other useful work can be done by the controller during the delay time. This is a serious limitation for more intensive programming tasks but for now it is sufficient.

Let's move off of the internal LED to an external LED. While the AVR 328P can deliver in excess of 20 milliamps per pin, if we choose to drive several LEDs it may be better to use transistor drivers for them in order to reduce the current and power demands on the controller. The basic style follows:

NPN Driver   *PNP Driver*
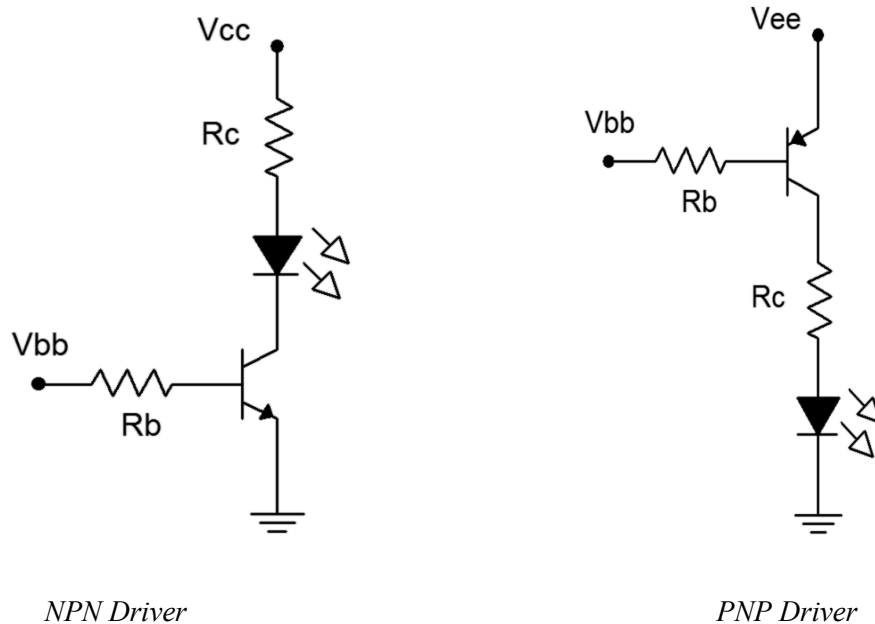
The NPN driver will saturate and turn on the LED with a high base voltage while the PNP version saturates with a low base voltage. In both cases the LED current is the saturation current which can be approximated as the supply voltage minus the LED voltage, divided by the collector resistance. For example, to achieve 10 milliamps with a 5 volt supply and a red LED (about 2 volts forward bias), the collector resistor would need to be about 300 ohms. The base resistor would be about twenty times this value resulting in a base current of about 700 microamps. This would guarantee saturation for any reasonable transistor beta. The 2N3904 or 2N3906 are common choices but if a large number of devices are to be driven it might be wise to consider a driver IC such as the ULN2003.

Let's modify the example above to use an external NPN driver on Arduino digital pin 8. For future reference, this corresponds to Port B, bit 0. First, wire up the driver on a protoboard using standard resistor values (a 330 and a 6.8k should do). Jump the Vcc connection to the +5V header pin on the Uno and jump the ground connection to the Uno's GND header pin (both on the POWER side of the board). Wire the input logic connection (Vbb) to Arduino digital pin 8. Modify the code as follows:

```
/*    External LED Blink   */

#define LEDPIN 8

void setup()
{

      pinMode( LEDPIN, OUTPUT );
}

void loop()
{
      digitalWrite( LEDPIN, HIGH );
      delay(2000);                      // wait for 2 secs

      digitalWrite( LEDPIN, LOW );
      delay(1000);                      // wait for 1 sec
}
```

Compile the code, transfer it and run it. You should have a nice blinking LED. Note the timing. Build the PNP driver and connect it to pin 8 in place of the NPN version (do not disassemble the NPN driver, just unhook it from the controller). Do not change the code. You should notice a difference in the on/off timing. If don't notice a change, reconnect the NPN and look again.

Now let's consider direct access to the hardware without using the Arduino libraries. There are two ways to do this; the first is by looking up the actual addresses for the hardware ports and registers and assigning pointer variables to them. The second method (somewhat easier) involves using the pre-defined variables for the ports and registers.

The following is a variation of the blink program using direct port access via pointers. Note that the variables for the port and data direction registers are declared and then assigned hard numerical addresses. These would vary from processor to processor so the code is not portable. That's not desirable; however, the process does illustrate precisely what is going on with the controller and does so with a reduction in resulting assembly code. Note the usage of LEDmask. Changing the value of this allows you to access different bits of that port. Also, note the use of the bitwise OR, AND and COMPLEMENT operators (|&~) to set the appropriate bit(s) in the registers.

```
/* Direct Blink: Using direct bit fiddling via pointers */

// This is Port B bit 0, Arduino digital output number 8

#define LEDMASK 0x01

void setup()
{
      unsigned char *portDDRB;

      portDDRB = (unsigned char *)0x24;

      // initialize the digital pin as an output.
      *portDDRB |= LEDMASK;
}
```

```
void loop()
{
      unsigned char *portB;

      portB = (unsigned char *)0x25;

      // turn LED on
      *portB |= LEDMASK;
      delay(2000);

      // turn LED off
      *portB &= (~LEDMASK);
      delay(1000);
}
```

Type in the code above and connect the NPN driver to pin 8. Compile and transfer the code, and test it. It should run as before.

A better version of Direct Blink uses pre-defined variables such as PORTB. These can be defined within a header file and as such, can be adjusted across devices. This makes the code portable across a family of controllers rather than tied to a specific unit. As you might notice, the code also tends to be somewhat easier to read.

```
/* Direct Blink 2: Using direct bit fiddling via pre-defined variables */

// This is Port B bit 0, Arduino digital output number 8

#define LEDMASK 0x01

void setup()
{
      // initialize the digital pin as an output.
      DDRB |= LEDMASK;
}


void loop()
{
      // turn LED on
      PORTB |= LEDMASK;
      delay(2000);

      // turn LED off
      PORTB &= (~LEDMASK);
      delay(1000);
}
```

Type in the code above, compile and transfer the code, and test it. It should run as before.

All versions of this program can be extended to control several output bits, each connected to something different. For example, one bit could control an LED while another bit controls a relay or power transistor that controls a motor. Using Direct Blink 2, multiple bits can be controlled with a single operation simply

by creating an appropriate bit mask. For a quick example, simply set `LEDMASK` to `0x21` in Direct Blink 2 to control both the external and on-board LEDs in tandem. The limitation with this approach is that each item cannot be controlled with completely different timing; all items generally will have the same timing. It is possible to break up the delay calls into smaller chunks and insert calls to turn on and off individual bits between them. This is a rather clunky method though and is neither particularly accurate nor flexible if a great many items need to be controlled. We shall examine ways around this in future work.

Assignment

Create a program that will blink two LEDs; one red, one green, alternately. That is, when the red LED comes on, the green should turn off and when the green comes on, the red should turn off. The red should be on for two seconds and the green should be on for one second. At no time should both LEDs be on simultaneously or off simultaneously (at least not within ordinary human perception). This blinking pattern should repeat over and over: red on for two seconds, green on for one second, red on for two, green on for one, and so on.

Include your code and a proper schematic diagram with component values. It is suggested that Multisim or another schematic capture program be used to generate the schematic.

8
Arduino Digital Input

This exercise examines basic digital input code and hardware for the microcontroller. Typically, the controller will need to react to external true/false state changes from either a user or some other circuit. For example, a user may push a button indicating that they want to start or stop a process such as muting an audio signal or engaging an electric motor. There is no "half way" with these sorts of things; you don't "sort of start a motor a little bit", you either do it or don't. The controller's digital input pins are designed for precisely this sort of signal.

Input signals must conform to proper logic levels in order to properly trigger the controller. In the case of the Arduino Uno that means the signals must adhere to the standard 0V and 5V. The signal sources can be thought of in terms of two broad categories: active and passive. An active source would be a circuit such as a logic gate or comparator that drives the input pin to 5V or 0V. For some applications this is excessive and we can get by with something much simpler, namely the passive type. For these, an internal pull-up resistor is enabled on the input pin of the controller. The pull-up is a large value resistor tied between the power supply and the input pin. The pull-up will create a logic high at the input (i.e., the input pin is "pulled high"). To achieve a low, we connect a simple switch from the pin to ground. No other external circuitry or power is needed. Obviously, a pull-up should be disabled when using active sources.

To read the logic level at an input, two things must be done. First, the corresponding data direction register bit must be set for input or read mode (and optionally, we may also wish to engage the pull-up). Once this is done we then read the appropriate port pin bit. It is worth noting that on some microcontrollers the same register is used for both input and output (e.g., the PORTB register as used on the digital write exercise). On the Atmel AVR series (such as the ATmega 328P used on the Uno), however, two different addresses are used: PORTx for writing and PINx for reading. A common error for the new AVR programmer is to try to read from PORTB when they really want to read from PINB.

In our first example we'll toggle an input pin with a simple SPST switch using an internal pull-up. The state will be observed using the Serial Monitor. Initially we'll do this using the Arduino libraries but we'll also look at more generic forms.

Solder a couple wires to an SPST switch and connect them to pin 8 and ground. If you prefer, you can simply insert a wire into Arduino pin 8 and then manually insert the free end into one of the ground contacts when needed. Type the following code into the editor:

```
/* Read Digital Input V1. Monitors Arduino pin 8 and sends value to the
Serial Monitor */

void setup()
{
      Serial.begin(9600);
      pinMode(8, INPUT_PULLUP);
}

void loop()
{
      int i;

      i = digitalRead(8);
      Serial.println(i);
}
```

This is about as basic as it gets. First we open the serial monitor and then set the data direction register for input mode with pull-up via the `pinMode()` function (setting the mode to just `INPUT` disables the pull-up). In the looping section we read the value from pin 8 and then report it back to the Serial Monitor. Compile the code and transfer it to the Arduino board. Open the Serial Monitor. You should see a series of logic highs (1) scrolling by. Now insert the free end of the wire into one of the ground headers. The Serial Monitor should now show a series of logic lows (0) scrolling by. As you connect and disconnect the wire (or throw the switch) the values should flip back and forth.

```
/* Read Digital Input V2. Monitors Arduino pin 8 and sends value to the
Serial Monitor, generic form */

// Arduino pin 8 is bit 0 of port B, also known as port bit B.0
#define BITMASK 0x01

void setup()
{
      Serial.begin(9600);

      DDRB &= (~BITMASK);      // initialize the pin as an input.
      PORTB |= BITMASK;        // enable pull-up
}

void loop()
{
      int i;

      i = PINB & BITMASK;
      Serial.println(i);
}
```

Compared to the original, the two calls to set the input mode are a little more work but result in noticeably less machine language code (under V1.0.4 the first version used 2726 bytes while the second used 2340 bytes as noted in the status bar of the IDE). Also note that the printed variable is the actual bit value not the idealized 1 for true and 0 for false (in this instance a 1 at bit 0 is the value 1 so it's not immediately apparent). This is generally not a problem as any non-zero value is treated as true in the C language. As a reminder, don't check to see if a variable is true by testing to see if it's the same as 1. Instead, check to see if it "exists":

```
if(v==1)      // don't do this to check for true
if(v)         // do this instead
if(!v)        // and do this to check for false
```

Enter the second version of the Read Digital Input program, compile it, transfer it and test it. When watching the Serial Monitor the constant scrolling can be a bit of a pain (pun intended). Currently, it's as if the Uno is saying "switch is down, switch is still down, switch is still down" over and over. It would be more convenient if it only reported a change of state. To do this, the code will need to remember the current state for future comparison. Consider the following alteration:

```
/* Read Digital Input V3. Monitors Arduino pin 8 and sends value to the
Serial Monitor, generic form, only reports state changes */

// Arduino pin 8 is port bit B.0
#define BITMASK 0x01

int priorstate=0;

void setup()
{
      Serial.begin(9600);

      DDRB &= (~BITMASK);     // initialize the pin as an input.
      PORTB |= BITMASK;       // enable pull-up
}

void loop()
{
      int state;

      state = PINB & BITMASK;

      if( state != priorstate )
      {
            Serial.println(state);
            priorstate = state;
      }
}
```

We initialize the `priorstate` variable to 0 as that's what the input pin is prior to power up and on reset. The `state` variable is only sent to the Serial Monitor if there has been a change of state. Edit, compile, transfer and test the new version. Note that the flurry of scrolled values is replaced by a more useful output. If you throw the switch back and forth several times while watching the Serial Monitor you might see something odd. You'll probably notice that you get more state changes than throws of the switch. Where are these random extras coming from?

Debouncing

Real world mechanical switches do not behave ideally. When a switch is thrown, the mechanical contacts bounce after the initial closure resulting in a series of make-break contacts. This might last for 10 milliseconds or more depending on the design of the switch. Examples of switch bounce are shown in Figure 8-1. The problem is that the microcontroller can read the input pin in a fraction of a millisecond,

and therefore, it appears as if the switch is being thrown on and off very rapidly during this multi-millisecond stretch. In some applications this is not a problem, for example, when an LED is activated according to switch position (i.e. "up" or "depressed" lights the LED). In this instance, the bounce or chatter of the signal and the LED will not be noticed by the human eye. In contrast, we might program a button switch so that multiple pushes increase a level such as brightness or volume. In this case the chatter might cause a series of levels to be skipped over yielding an inconsistent and quirky response to user input.
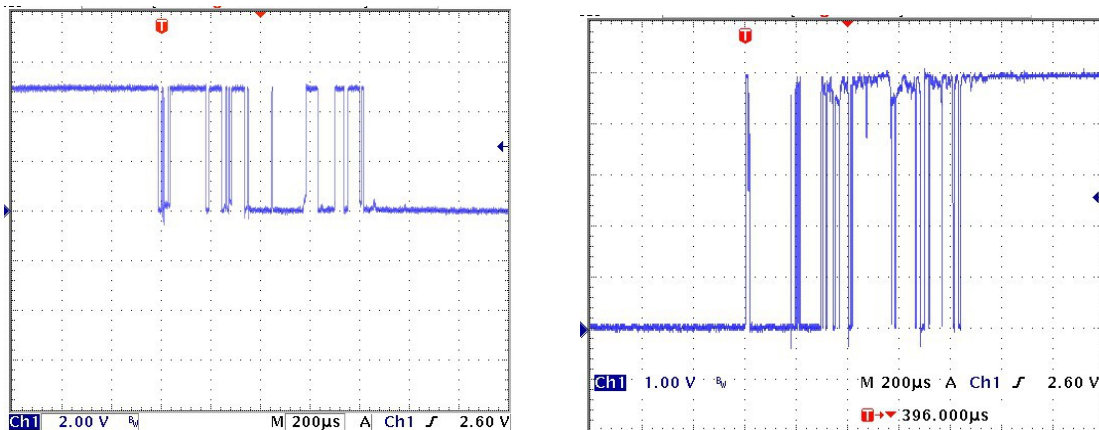
Switch bounce

Figure 8-1

There are two basic ways of "debouncing" switches. The first involves software and the second involves hardware. Let's look at a software solution first.

The key to a software debounce is to stretch out the time the switch is being examined. In other words, we want to examine the state over a long enough period of time that the bouncing settles out. For example, if we note that the state has not changed over the course of, say, ten milliseconds, we can safely assume we're not trying to read the input during a bounce. Consider the code variant below. It looks a lot like version three but there is a new function added, `get_state()`. This is where all the action is.

```
/* Read Digital Input V4. Monitors Arduino pin 8 and sends value to the
Serial Monitor, generic form, only reports state changes, with software
debounce */

// new function prototype
int get_state(void);

// Arduino pin 8 is port bit B.0
#define BITMASK 0x01

int priorstate=0;

void setup()
{
      Serial.begin(9600);

      DDRB &= (~BITMASK);      // initialize the pin as an input.
      PORTB |= BITMASK;        // enable pull-up
}
```

```
void loop()
{
      int state;

      state = get_state();

      if( state != priorstate )
      {
            Serial.println(state);
            priorstate = state;
      }
}

int get_state(void)
{
      int priorstate, state, count=0, i=0;

      priorstate = PINB & BITMASK;

      while( i < 30 )
      {
            state = PINB & BITMASK;

            if( state == priorstate )
                  count++;
            else
            {
                  count = 0;
                  priorstate = state;
            }

            if( count >= 10 )
                  break;

            delay(1);
            i++;
      }
      return( state );
}
```

The idea behind this new function is to continually check the pin and not return until we have seen ten milliseconds worth of consistent values. Every time the current state is the same as the prior state we increment `count`. When it hits 10 we know we have achieved at least ten milliseconds of unchanging signal (due to the one millisecond `delay()` call at the bottom of the loop), we break out of the loop and return the pin state. If levels shift due to bounce, `state` and `priorstate` will not be the same so `count` is reset. If a switch is faulty, the bounce period could last a very long time and to prevent the code from "hanging up" and staying in this function forever, a maximum number of 30 iterations (at least 30 milliseconds) is instituted via the variable `i`.

Enter version four and save it under a new name as we'll be coming back to version three a little later. Compile and transfer the code, and then test it using the Serial Monitor. This version should produce a nice, consistent output on the Serial Monitor without the extraneous transitions of version three. It's also

worth noting that the resultant machine code is still smaller than the original code made with the Arduino libraries in spite of greater functionality (2654 bytes versus 2726 bytes for version one).

The second method to debounce a switch is via hardware. Generally, this takes the form of the switch being connected to an RC network which feeds a Schmitt Trigger (74HC14). The RC network will effectively filter the chattering signal and the hysteresis of the Schmitt will take care of any lingering noise or fluctuation. Figure 8-2 is a typical example (note the switch logic is reversed due to the inverter):
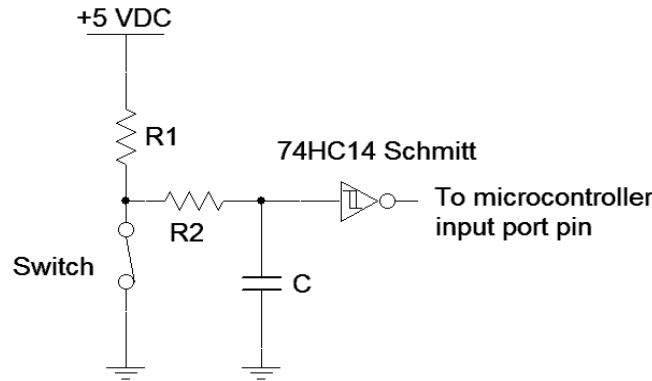
RC-Schmitt Trigger debounce circuit

Figure 8-2

Typical values for this circuit would be 4.7 kΩ for R1, 470 kΩ for R2 and 100 nF for C. The RC time constant for this circuit is roughly 50 milliseconds so the Schmitt thresholds will be reached in around 200 milliseconds (less than 5 τ). This should take care of even the crankiest mechanical switch. Also, note that this is an *active* input signal so the input pull-up will not be needed. Let's go back to version three (without software debounce) and make a modification to accommodate the hardware debounce:

```
/* Read Digital Input V5. Monitors Arduino pin 8 and sends value to the
Serial Monitor, generic form, only reports state changes, requires hardware
debounce to drive the input */

// Arduino pin 8 is port bit B.0
#define BITMASK 0x01

int priorstate=0;


void setup()
{
      Serial.begin(9600);

      DDRB &= (~BITMASK);      // initialize the pin as an input.
      // pull-up code has been removed
}
```

```
void loop()
{
      int state;

      state = PINB & BITMASK;

      if( state != priorstate )
      {
            Serial.println(state);
            priorstate = state;
      }
}
```

Remove the hardware switch from the board. Build the debounce circuit from Figure 8-2 and connect its output to Arduino pin 8. Compile and transfer the code. Open the Serial Monitor and test the results. It should be similar to that of version four with software debounce. Also note that the code is about 300 bytes smaller. We have effectively traded memory for hardware.

There are some nice advantages to software debounce compared to hardware debounce. First, it is very flexible in that the code can be changed to fine tune the performance. Second, there's no associated cost for the additional hardware components or expanded board space to accommodate them. There are some disadvantages, though, including increased code memory requirements for the debouncing functions (remember, microcontrollers, unlike PCs, tend to have very modest memory capacity). Another issue is the time wasted waiting for the switch signal to settle. Ten or twenty milliseconds may not seem like a lot but in some applications it can be an eternity. Whether software or hardware debounce will be used for a particular design; indeed, whether debounce will be used at all, will depend on other system factors.

Obviously, monitoring the pin state with the Serial Monitor tool is very useful for debugging and learning how the device works but it is not a typical application so let's move to something a little more useful. Instead of sending output to the serial monitor, let's reflect the state using an external LED off of Arduino pin 12, i.e., bit 4 of port B. We'll continue to use the hardware debounce circuit with the new code below:

```
/* Read Digital Input V6. Monitors Arduino pin 8 and reflects value to an
external LED, generic form, requires hardware debounce to drive the input */

// Arduino pin 8 is port bit B.0
#define BITMASK 0x01

// Place LED driver on Arduino pin 12 or port bit B.4
#define LEDMASK 0x10

int priorstate=0;


void setup()
{
      DDRB &= (~BITMASK);     // initialize the pin as an input from 74HC14
      DDRB |= LEDMASK;        // initialize the pin as an output to the LED
}
```

```
void loop()
{
    int state;

    state = PINB & BITMASK;

    if( state != priorstate ) // don't bother if no change
    {
        priorstate = state;

        if( state )
            PORTB |= LEDMASK;        // light LED
        else
            PORTB &= (~LEDMASK);     // extinguish LED
    }
}
```
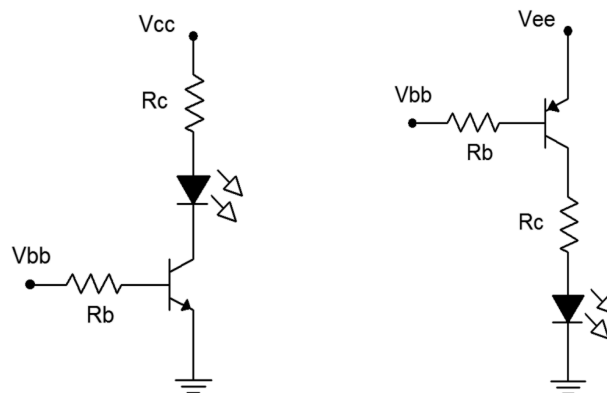
Build a standard LED driver circuit (NPN and PNP versions shown in Figure 8-3) and connect it to Arduino pin 12. Determine a value for Rc to achieve about 10 milliamps of LED current assuming the LED voltage is 2 volts. Set Rb to achieve a base current of around 500 microamps.

LED Driver Circuits

Figure 8-3

Once the hardware is in place; enter, compile and transfer the code. The LED should reflect your switch position. Also, if you haven't already noticed, look at the status bar on the editor. Because we are no longer using the Serial Monitor and its associated library, the code size has dropped to a mere 510 bytes!

Assignment

It is not always the case that an LED would directly reflect the state of a switch. As mentioned earlier, a push-button might be used to toggle a device. For example, push to turn something on, push again to turn it off. Modify the existing code so that the switch toggles the state of the LED rather than having the LED indicate the position of the switch. That is, the LED should change state only on a low-to-high transition of the switch (or alternately, only on a high-to-low transition). Include your code and the accompanying schematic diagram drawn using Multisim or another schematic capture tool.

9

Arduino Analog Input

In this exercise we shall examine a method of obtaining analog input signals such as those generated by various continuous sensors. Continuously variable sensors might be used to measure temperature, force, acceleration, and so forth. This ability shall be combined with the digital input and output techniques already examined to produce a system that indicates an applied force via a string of LEDs; The greater the applied force to the sensor, the greater the LED indication.

Microcontrollers rely on analog to digital converters (ADCs) to obtain continuously variable input data. The ATmega 328P contains a 10 bit ADC along with a six channel multiplexer. These six lines are brought out to the Arduino Uno's analog input header; pins A0 through A5. By default, the reference voltage for the converter is the on-board five volt DC source. This represents the maximum input level that may be digitized. The minimum level is ground or zero volts (if negative signals are needed, some form of offset bias will be required). A 10 bit converter yields 2^{10} or 1024 discrete levels. This means that each step will be slightly less than five millivolts.

To use the ADC, a reference voltage must be set during the initialization routine. After this, whenever a value is needed, simply call the `analogRead()` function. The only argument to the function is the input channel desired (zero through five for the Uno). The function returns the digitized value as an integer in the range of 0 to 1023. While it is possible to go direct to the hardware registers to access the ADC, the Arduino library is fairly efficient and we will save only a few bytes, if any, by avoiding it[1].

We shall start with something straightforward to illustrate how to read an analog input pin. Note that unlike the digital ports, initially setting `pinMode()` or a data direction register (`DDRx`) is not required when using the `analogRead()` function.

Connect the outer pins of a 10 kΩ potentiometer to the +5V and ground terminals on the Uno's header. Connect the wiper to analog pin 0 (A0). Enter the code below, compile and transfer it to the board.

```
/* Read analog V1. Reads voltage off of a pot and prints the digitized value
to the Serial Monitor. Range = 0->1024 */

#define ANALOG_IN_PIN 0

int prior = 0; // remembers prior value of analog input

void setup()
{
  Serial.begin(9600);
  analogReference( DEFAULT );
}
```

---

[1] See "Bits & Pieces: analogRead" in the text for details.

```
void loop()
{
      int a;

      a = analogRead( ANALOG_IN_PIN );

      if( a != prior )
      {
            Serial.println(a);
            prior = a;
      }
}
```

The code is self-explanatory. The interesting twist is that instead of simply printing the analog value, the value is compared to the previous value and only if they are different is the result printed. This saves some processing effort and speeds things up.

Open the Serial Monitor. Turn the pot shaft clockwise and counterclockwise. The values on the Serial Monitor should change as the shaft is adjusted. The values should track from 0 up to 1023 (you might be off by a count due to noise). Quite literally, you could determine the actual potentiometer wiper voltage by multiplying the printed value by the step size (5V/1024 or roughly 4.883 millivolts per step), thus achieving a very rudimentary DC voltmeter.

Force Sensor

Let's replace the potentiometer with a sensor. For this exercise we shall use an FSR or force sensing resistor. An FSR is a two terminal device usually in the form of a flat film, the resistance of which depends on the applied force (FSR 400 series data sheet and integration guide). With no applied force the resistance is very high, usually much greater than one megohm. With full force applied (such as a hard squeeze of the fingers) the resistance may drop to as little as 100 Ω. Remove the pot and wire one lead of the FSR to the +5V header pin and the other end to a 10 kΩ resistor. The connection point between the resistor and FSR should be wired to analog pin A0 while the free end of the resistor should be connected to ground. This is nothing more than a voltage divider, one leg of which is a force-dependent resistance. As force is applied, the resistance drops and therefore the connection point voltage will rise. Do not change the code. Simply note how the values in the Serial Monitor change as force is applied to the FSR: the greater the force, the greater the value. Finally, swap the positions of the resistor and FSR. Increasing force should now result in decreasing values in the Serial Monitor. **Leave the FSR in this new position for the remainder of this exercise.**

LED Output

Now that the sensor and analog input function have been verified, let's combine this with an LED display. Instead of printing values to the Serial Monitor, we will light up a string of LEDs. This will require digital output (write) code as explored in a prior exercise. To keep the effort simple and just show "proof of concept", we'll light up to four LEDs. Each LED will require an appropriate transistor driver circuit (or consider a Darlington driver array such as the ULN2003 that contains seven drivers in a single IC). We shall use the bottom four bits of port B for the output signals, Arduino pins 8 through 11. Consider the code below:

```
/* Read analog V2. Uses FSR (lower) and 10k (to +5V) on pin A0. Lights 4 LEDs
off of port B.0:3 to indicate force. Active high drivers. */

#define ANALOG_IN_PIN 0

// These are the bits to be used on port B for the LEDs
#define LEDMASK0        0x01
#define LEDMASK1        0x02
#define LEDMASK2        0x04
#define LEDMASK3        0x08
#define LEDMASK         (LEDMASK0 | LEDMASK1 | LEDMASK2 | LEDMASK3)

void  lightLEDsBar( int a );

int prior=0;

void setup()
{
  Serial.begin(9600);
  analogReference( DEFAULT );
  // set LED driver bits to output mode
  DDRB |= LEDMASK;
}

void loop()
{
  int a;

  a = analogRead( ANALOG_IN_PIN );
  if( a != prior )
  {
    Serial.println(a);
    lightLEDsBar( a );
    prior = a;
  }
}

void  lightLEDsBar( int a )
{

  if( a > 820 )
    PORTB |= LEDMASK; // light everything
  else
  {
    if( a > 615 )
    {
      PORTB &= (~LEDMASK3); // extinguish top
      PORTB |= (LEDMASK2 | LEDMASK1 | LEDMASK0); // light everything below
    }
    else
    {
      if( a > 410 )
      {
        PORTB &= (~(LEDMASK3 | LEDMASK2));
        PORTB |= (LEDMASK1 | LEDMASK0);
      }
      else
```

```
      {
        if( a > 205 )
        {
          PORTB &= (~(LEDMASK3 | LEDMASK2 | LEDMASK1));
          PORTB |= LEDMASK0;
        }
        else
          PORTB &= (~LEDMASK);
      }
    }
  }
}
```

Of primary interest here is the requirement to initialize `DDRB` for output on bits 0:3 which will be used to drive the LEDs. Separate bit masks are defined for each output pin for ease of code maintenance. A new function has been created to handle the lighting chores, `lightLEDsBar()`. This function will create a bar graph based on the value of the argument passed to it. As there are four LEDs, we can slice the 1024 range into five segments. If the value is below 20% (about 205) no LEDs are lit. If it's between 20% and 40% only the lowest LED is lit. As the value increases more LEDs are lit until, at 80% or higher, all LEDs are lit. The function is little more than a big cascading if/else. In any given region first the top LEDs are unlit and then the remaining LEDs below are lit. Note that the technique above is more efficient than using the `digitalWrite()` function because we can set the state of several bits at once by directly accessing `PORTB`. In contrast, `digitalWrite()` would require one call for each bit/LED. The Serial Monitor code has been left in so that you can monitor the results to make sure everything is working properly.

While leaving the FSR in place, wire four LED drivers to Arduino pins 8 through 11. An LED current of 10 milliamps should be sufficient. Enter the code above, compile and transfer it. To test, simply open the Serial Monitor and press on the FSR. As the force increases, the values on the Serial Monitor should decrease and fewer LEDs should light. You can think of this as an *"absence of force"* meter. While this might sound backwards, it can be quite useful. There are times when we want something to be under pressure and a lack of force serves as a warning (i.e. more LEDs lit means a bigger problem).

Sometimes a string of LEDs are arranged to mimic an analog meter. In such an instance we might wish to only light a single LED rather than all of the LEDs below it. We might call this "dot mode" versus "bar mode". Version three of the program shows dot mode with an interesting twist. Also, the Serial Monitor code has been removed. Note the decrease in program size.

```
/* Read analog V3. Uses FSR (lower) and 10k (to +5V) on pin A0. Lights 4 LEDs
off of port B.0:3 to indicate force. Dot display mode. Active high drivers. */

#define ANALOG_IN_PIN 0

// These are the bits to be used on port B for the LEDs
#define LEDMASK0        0x01
#define LEDMASK1        0x02
#define LEDMASK2        0x04
#define LEDMASK3        0x08
#define LEDMASK         (LEDMASK0 | LEDMASK1 | LEDMASK2 | LEDMASK3)

void  lightLEDsDot( int a );

int prior=0;

int thresh[] = {820, 615, 410, 205};
```

```
void setup()
{
  analogReference( DEFAULT );
  DDRB |= LEDMASK;   // set LED driver bits to output mode
}

void loop()
{
  int a;

  a = analogRead( ANALOG_IN_PIN );
  if( a != prior )
  {
    lightLEDsDot( a );
    prior = a;
  }
}

void  lightLEDsDot( int a )
{
  int *p;
  p = thresh;

  if( a > *p++ )
  {
    PORTB &= (~(LEDMASK2 | LEDMASK1 | LEDMASK0)); // turn off bottom 3
    PORTB |= (LEDMASK3);                          // light only topmost
  }
  else
  {
    if( a > *p++ )
    {
      PORTB &= (~(LEDMASK3 | LEDMASK1 | LEDMASK0));
      PORTB |= (LEDMASK2);
    }
    else
    {
      if( a > *p++ )
      {
        PORTB &= (~(LEDMASK3 | LEDMASK2 | LEDMASK0));
        PORTB |= LEDMASK1;
      }
      else
      {
        if( a > *p )
        {
          PORTB &= (~(LEDMASK3 | LEDMASK2 | LEDMASK1));
          PORTB |= LEDMASK0;
        }
        else
          PORTB &= (~LEDMASK);
      }
    }
  }
}
```

The Dot variant is similar to the Bar version except that all LEDs are extinguished with the exception of the single "top most" LED. You might wonder why we go to trouble of extinguishing the individual effected bits instead of just setting them all to off (ANDing with 0xf0) and then setting the desired bit on. The reason is because this might create a quick toggle on/off of the desired "top most" bit. In this application you'd probably never see the difference but there might be applications where this toggling can be a problem. In any case, the total amount of machine code generated is roughly the same either way.

The more interesting alteration of the function is that an array of values is used for the thresholds rather than fixed, hard-coded values. This makes the program much more flexible should other levels be required in the future. To use a different set of thresholds, all that needs to be done is a pointer assignment to the base of the appropriate threshold array at the start of the function. Also, note in particular the use of the post-increment operator on the pointer p. This is generally more efficient than the typical array indexing method.

Assignment

Alter the program so that the lighting logic is inverted, that is, increasing force should light a greater number of LEDs in bar mode and reverses the motion of dot mode. Do **not** alter the FSR circuit. This must be achieved via code only. Second, add a switch to Arduino pin 7 (port D.7). The position of this switch should allow the user to choose between a dot mode and bar mode display. Debouncing of this switch should not be necessary. For extra credit, add a second switch (Arduino pin 6, Port D.6) which allows the user to choose between the existing linear scale and a logarithmic scale corresponding to 6 dB per LED (i.e. halving of levels, the top most LED lighting above 512). Include your code and a proper schematic diagram with component values.

10

Arduino Reaction Timer

In this exercise we shall combine digital input and output processing along with timing functions and random number generation to create a simple game. The point of the game will be to test the player's reaction time (technically, we'll be measuring the *response time* which is the sum of the reaction time plus the resulting movement time). Instead of just looking at code, this exercise also will allow us to take a step back and look at system design and simple user interface issues.

The game will give the player five tries to achieve their best reaction time. Each try will consist of activating a red warning light as a means of telling the player to get ready. Then, a second green light (i.e., "go") will be activated between one and ten seconds later, the precise time being random between tries. Once the green light comes on, the player needs to hit a switch. The amount of time between the green light turning on and the response switch activating is the player's reaction time. The player will be told this value and at the end of the five tries, the best result will be displayed. The game also has to guard against cheating. That is, it has to disallow a hit that simply anticipates the green light firing, in other words, "jumping the gun".

Before we consider hardware and software, we need to have some idea of what we're measuring. In general, how fast is human response time? It turns out that this is partly a function of the stimulus (audible cues are processed more quickly by the human brain than visual cues). Typical response times are around 160 milliseconds for an aural stimulus and 190 milliseconds for a visual stimulus, although professional athletes such as Olympic sprinters might on occasion achieve a value only 2/3[rds] of this[2]. Obviously then, our hardware and software need to be much faster if we want to achieve a decent level of accuracy.

The user interface consists of four things: a red light, a green light, the player's response switch and some means of text output. For simplicity and ease of prototyping, we shall use the Serial Monitor for text output. If this were to become a production item, the monitor could be replaced with a dedicated LCD display. Regarding the lights, LEDs seem to be an obvious choice perhaps because of their familiarity and the one-on-one personal immediacy of the game (if several people were to respond in sync, we might need a much larger or brighter light source than an ordinary LED). There is one other item to be considered and that is the turn-on speed of the light. LEDs turn on in a fraction of a millisecond whereas incandescent lights might not reach full brightness for tens or even hundreds of milliseconds. As we might be expecting response times in the 200 millisecond region, an incandescent may not be fast enough. That is, the time delay required for it to reach full brightness might skew the results, slowing the player's apparent (measured) response time. So, it seems that simple LEDs would be a good choice here.

The final item is the player's switch. The player might hit the switch fairly hard because they'll want to move their hand as fast as possible to minimize the movement time. Consequently, the switch needs to be fairly robust. It should also represent a decently sized target for the player, that is, one that is large enough that the player doesn't also have to worry about making a very precise movement. While a simple

---

[2] See http://en.wikipedia.org/wiki/Reaction_time

momentary contact push-button might seem like a decent choice, an FSR might prove better. The FSR contains no moving parts and provides a large target area. It also responds nearly instantly and does not suffer from switch bounce. Interconnection is easy too: We can simply connect it between an input pin and ground, and enable the internal pull-up. Normally the FSR is a very high resistance but when pressed the value drops to maybe 100 or so ohms. This would be in series with the pull-up that is connected to the power supply. With the FSR untouched, the voltage divider rule indicates the pin voltage will float high. When the FSR is depressed, the voltage will dive to a logic low. The only downside to the FSR is the lack of physical feedback. Many switches make a clicking sound when they are engaged. This provides auditory feedback to player so that they know that their strike has registered. We can accomplish some player feedback by lighting one or both of the LEDs at the moment the strike is sensed.

We now have an idea of the potential hardware requirements. What about the software? Obviously, previously examined techniques for digital input and output can be used to read and control the player switch and LEDs. The `delay()` function can be used for timing the LED flashes and the main "go" delay. Two new items are needed though: some way of generating random values (in our case, between 1000 milliseconds and 10,000 milliseconds) and some method of measuring the time between two events. Let's consider the random number first.

Generally, digital computing devices are not good at generating truly random values because they are deterministic machines. In fact, we try to "design out" all potential randomness because we need highly predictable operation. Truly random values are in no way correlated to each other. That is, there is nothing about a string of random values that will allow you to determine the next value in the sequence. It turns out, though, that we can create pseudo-random sequences without a lot of effort. A pseudo-random sequence appears to be random but if you run the sequence long enough, it will repeat. Pseudo-random sequences may not be sufficiently random for proper scientific research efforts but they're plenty sufficient for something like our game. While the process of deriving pseudo-random numbers is beyond the scope of this exercise, we can say that the process involves some fairly rudimentary processing, such as bit shifting, on a starting value referred to as a *seed*. The result of this computation is then used as input to compute the next value and so on.

The Arduino library includes a function, `random()`, which will return a pseudo-random long integer value. The function may also be called with upper and lower bounds to constrain the returned values as in `random(30,200)`. The sequence itself is initiated via the `randomSeed()` function which takes a long integer argument. A sometimes useful effect of this is that if the seed value is the same for subsequent runs of the program, the resulting pseudo-random sequence will repeat exactly. This allows for a somewhat systematic debugging of a supposedly random feature. But this raises a problem, namely, how do we get a random number for the seed to begin with? While it's possible to ask the user for a seed value, that opens the door to possible cheating. A better way is to use a noise voltage. By its very nature, noise is random. If we look at an analog input pin which is not connected to anything, we will see a small noise voltage. We can digitize this truly random value with the `analogRead()` function and use it for the seed. This voltage won't change over a very large range, but it will change a sufficient amount to make our pseudo-random sequence to appear very random indeed.

 The following code shows how the random number system works. Enter the code, compile, and transfer it, and then run it. First it will print out the noise voltage and then print out random values between 50 and 1000. After you see a sufficient number of values, hit the reset button on the board to restart the code. Do this several times. These restarts should yield different sequences. From time to time you might get the same noise voltage. If you do see this, notice that the pseudo-random number sequence will be repeated exactly.

```
void setup()
{
      long i;

      Serial.begin(9600);

      i=analogRead(0); // any analog channel will do
      randomSeed(i);

      Serial.print("Noise voltage: ");
      Serial.println(i);
}

void loop()
{
      long a;

      a = random(50,1000);
      Serial.println(a);
      delay(2000);
}
```

This technique should work perfectly for our application. All we need to do is constrain the random function to 1000 to 10,000 so we can use the values directly as the millisecond argument for the `delay()` call used to randomize the lighting of the green "go" LED.

The second part that we need involves measuring the time delay between the lighting of the "go" LED and the player striking the FSR. Normally, if you were doing this from scratch you'd have to initialize one of the ATmega 328P's timers and check its contents when needed. Timing events is a common need in the embedded world so the Arduino initialization routine sets up a timer for you. The instant you turn on or reset the board, this internal timer starts incrementing a register that is initialized at zero. The register is incremented by one each millisecond. This timer keeps running regardless of and independent of the main code that is running (unless the main code reprograms the timer, of course). Consequently, the register value indicates how many milliseconds have passed since the board was turned on or reset. The contents of this register may be read via the `millis()` function. That is, `millis()` returns the number of milliseconds that have transpired since the last reset/power up. After about 50 days, this register will overflow and the count will begin again. It is doubtful that any individual will want to play this game that long continuously so this should not be issue. The following code snippet will do what we need:

```
starttime = millis();

// Flash "go" LED

// wait for response from player pressing FSR, looking at input port pin

finishtime = millis();
responsetime = finishtime - starttime;
```

So, we can now come up with a pseudo code for the looping portion of the game:

```
1. Initialize best response time as a very high value (long int)
2. Tell user to start game by hitting FSR pad
3. Wait for the player's start response of pressing the FSR
4. Start a loop for five tries, for each try:
```

```
       a. Generate a random number between 1000 and 10,000
       b. Wait a second or two to give the user a moment to get ready
       c. Flash red LED one or more times quickly to indicate the start
            of this try
       d. Delay the random amount of milliseconds from 4.a
       e. Record start time millis
       f. Flash green "go" LED quickly
       g. Wait for player's response of pressing FSR
       h. Record finish time and calculate response time millis
       j. Flash both red and green LEDs for visual feedback
       k. If response time is less than 100 msec, declare a cheat and
            reset response time to a high penalty value
       l. Print out the response time
       m. Compare response time to best time so far, resetting best to
            this value if this value is smaller (i.e., better)
     (Loop complete)
5. Print out the best value
6. Wait a moment before starting next game
```

A few of these steps might need further clarification or adjustment. In step 4.k, a "cheat" is declared if the response time is under 100 milliseconds. This is probably generous given the statistics noted earlier and could be adjusted to a higher value. The penalty value here could be the same as the initialization value, perhaps 10,000 or more milliseconds. After all, if it takes an otherwise normal individual more than 10 seconds to respond to the stimulus, one might surmise that they are inebriated, easily confused or need to stop texting so much.

Another item of importance is the code for steps 3 and 4.g. In essence, the code needs to "wait on" the player hitting the FSR. If the FSR is wired to an input pin utilizing an internal pull-up resistor, pressing the FSR will cause the digital input at that pin to go from high to low. (Remember, pressing the FSR creates a low resistance and the FSR is basically in series with the pull-up resistor. Therefore the voltage divider rule shows that the voltage between the two items drops to a very low value, or a logic low.) Consequently, the code can "busy wait" on that pin, breaking out when the pin goes low. We might wire up the FSR to Arduino pin 8 which is bit 0 of port B. Further, we might avoid hard coding this by using a #define:

```
#define FSRMASK  0x01
```

So the requisite bit test code looks like so:

```
while( PINB & FSRMASK );        // DON'T use PORTB!!
```

Remember, for input we look at PINx and for output we use PORTx. So this loop keeps running as long as the FSR input bit is high. As soon as the player pushes on the FSR with reasonable force, the pin goes low and the loop terminates.

We can streamline our code a little more regarding steps 4.c, 4.f and 4.h which flash an LED. We might hook up the green and red LEDs to Arduino pins 6 and 7 (port D.6 and D.7) and use #defines along with a function that turns an LED on and off quickly:

```
// At D.6
#define GREENLED 0x40
// At D.7
#define REDLED   0x80
// on/off time in msec
#define LEDTIME  30

void FlashLED( int mask )
{
   PORTD |= mask;     // turn on LED
   delay(LEDTIME);
   PORTD &= (~mask); // turn off LED
   delay(LEDTIME);
}
```

This function would be called like so for a single quick flash:

```
FlashLED(GREENLED);
```

It could also be placed in a loop for a series of quick flashes and you could bitwise OR the mask values to flash the two LEDs simultaneously.

The last item of concern is step 6, "Wait a moment before starting next game". Why would we include this? The reason is not obvious. Remember that the microcontroller will process everything in very short order, perhaps just a few milliseconds will transpire between the time the player hits the FSR, the finals values are printed and the `loop()` function terminates, only to be called again by the invisible `main()`. Upon re-entry, into `loop()`, the code tells the player to press the FSR to start the next game and then waits on the FSR bit. In that short amount of time the player will not have had a chance to lift their finger. In other words, the FSR bit is still low so the loop terminates and trial one executes immediately! To prevent this, we can either enter a small delay as the average person will not keep pressing the FSR once the response time is recorded, or we could use a separate FSR to start the game. The former seems much more straightforward.

Based on the earlier code snippets, the setup code might look something like this:

```
// Green LED at D.6, red at D.7, FSR at B.0, flash time in msec
#define GREENLED 0x40
#define REDLED   0x80
#define FSRMASK  0x01
#define LEDTIME  30

void setup()
{
   Serial.begin(9600);

   // set Arduino pins 6 & 7 for output to LEDs
   DDRD |= (GREENLED | REDLED);
   // Arduino pin 8 for input from FSR
   DDRB &= (~FSRMASK);   // initialize the pin as an input.
   PORTB |= FSRMASK;     // enable pull-up

   // get seed value for random, noise voltage at pin A0
   randomSeed(analogRead( 0 ));
}
```

Assignment

Based on the pseudo code presented prior, create the required `loop()` function for the game and test it on yourself several times. Turn in your completed code with an appropriate schematic showing all interconnections to the microcontroller. LEDs should be fed by an appropriate driver circuit to minimize controller output current. A nice modification for extra credit would be to have the program also determine and print out the total response time for each game of five tries along with the average response time. It should also indicate the number of "cheats", if any. Note that cheats will cause the total response time and average to suffer considerably!

11

Arduino Reaction Timer Redux

Sometimes it's nice to revisit a past project in order to investigate a different approach to the design. Here we're going to rework the Reaction Timer exercise. While it was certainly useful as an instructional exercise, the obvious limitation as far as the game itself is concerned is the fact that the development board needed to be connected to a host PC running the Arduino software because it relied on the Serial Monitor for numeric output. Instead, we're going to make a response timer that is self-contained. All it will need is a power source. There are a few different ways to achieve numeric output, perhaps the most direct is through the use of seven-segment LED displays. These devices will allow us to display all ten numerals plus a few alphabetic characters (but not the entire English alphabet unambiguously).

For ease of coding we shall dispense with the game "extras", namely determining the best-of-five attempts, the average, and so forth; and instead focus on interfacing the displays to a basic response timer. The overall operation will be similar: The displays will flash quickly to tell the user to "get ready", there will be a random time delay between one and ten seconds, then the displays will flash quickly as the trigger stimulus at which point the player will hit the response button (FSR) as quickly as possible. The response time will be shown on the displays for a few seconds and after a short pause, the process will repeat. The functional outline, or pseudo-code, is very similar to that of the original game. The major issue is the change of output device from Serial Monitor to seven-segment displays, both in hardware and software.

As far as the rest of the hardware is concerned, we will no longer need the signaling LEDs although we will still need the player's switch (FSR). We do not expect reaction times faster than 100 milliseconds and it is doubtful that healthy individuals would exhibit response times slower than one second, assuming they are not distracted or operating within the constraints of some manner of trendy chemical amusement aid that performs a (hopefully) temporary self-inflicted misfiring of their neural network. Consequently, three displays should be sufficient, allowing us to render from 0 through 999 milliseconds.

Three displays will require 21 port bits if we drive them independently. This is about all we have on the Arduino Uno so it's not a practical solution. Instead, we can time-multiplex the displays. In essence, we'll render the hundreds digit on the left-most display for a few milliseconds, then render the tens digit on the middle display for a few milliseconds, and finally render the units digit on the right-most display for a few milliseconds. Then we'll repeat the process. As long as we keep the overall loop time to less than about 25 milliseconds, our eye-brain processing will integrate the display over time so that it appears to be a solid and unwavering set of three digits. For this we'll only need ten port bits; seven for the segments and three more to multiplex the individual displays. If we use common anode displays (e.g., FND507) we could configure them as shown in Figure 11-1.
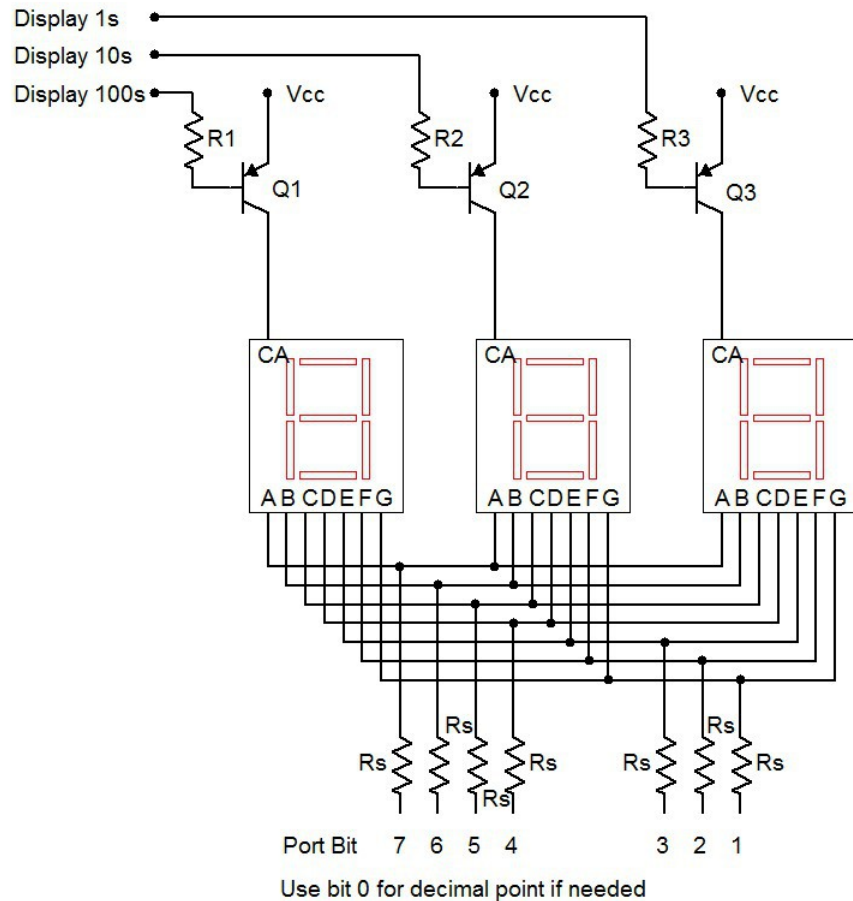
Seven-segment display configuration

Figure 11-1

The logic for this circuit will be "active low" meaning that a logic low lights a segment/display and a logic high turns it off. Note that LED drivers are not shown here, rather, the segments are connected directly to the port pins through current limiting resistors, Rs. The Arduino will have sufficient current capacity for this if we keep the segment current to a modest level. Assuming a 5 volt supply and about 2 volts for the LEDs, a 470 Ω would keep the currents at around 6 or 7 milliamps per segment. This would be sufficient for indoor use. The individual displays are enabled through the use of the PNP drivers. A logic low on their base resistors will place them into saturation, enabling current flow to the display. For this application, a small signal PNP such as the 2N3906 will suffice. Having all segments lit will draw about 50 milliamps through the collector. Therefore, a base resistor value in the low single digit kΩ range should work well, yielding a base current of a few milliamps.

Practically speaking, it would be easiest to dedicate port D to the segments (D.1:7, Arduino pins 1–7). This does mean that we'll lose the ability to use the Serial Monitor (it uses D.0:1) but that shouldn't be a problem. We can use B.0:2 (Arduino pins 8–10) for the digit multiplexers and B.3 (pin 11) for the FSR. Some #defines would be good for these. We'll also need an array of bit patterns for the displays:

```
// Port B.0:2 for 7 segment mux
#define DIGIT1   0x01
#define DIGIT10  0x02
#define DIGIT100 0x04
#define DIGIT111 0x07

#define FSRMASK  0x08

unsigned char numeral[]={
   //ABCDEFG,dp
   0b00000011,   // 0
   0b10011111,   // 1
   0b00100101,   // 2
   0b00001101,   // 3
   0b10011001,
   0b01001001,
   0b01000001,
   0b00011111,
   0b00000001,
   0b00011001,   // 9
   0b11111111,   // blank
   0b01100001,   // E
   0b01110011,   // r
   0b00001001,   // g
   0b00111001    // o
};

#define LETTER_BLANK  10
#define LETTER_E      11
#define LETTER_R      12
#define LETTER_G      13
#define LETTER_O      14
#define MSG_GO        -1
#define MSG_ERR       -2
```

The variable numeral is an array defined using binary values. Each bit signifies a segment with the MSB being segment A (top horizontal) and the LSB being the decimal point (unused here). For example, the numeral "0" requires every segment except G (the middle horizontal bar) and the decimal point to be lit. Because our circuit is active low, we put a 0 at each bit that will be lit. We repeat this process for the remaining nine numerals. We also add a few special characters, namely blank and the letters E, r, g and o. We'll use the first two to spell "Err" for error (what we used to call a "cheat") and the second two to spell "go" which will take the place of the former green "go" LED. We have also defined a couple of MSG variables to be examined in a moment.

Given these definitions, let's take a look at how the setup() function has changed. We need to change which pins are being used but we still need the seed value for random(). Note that since we are setting all bit of port D to output mode, we can just blast in 0xff instead of bitwise ORing the specific bits. It is also important that we set the mux bits (DIGIT111) of port B high. As the port D pins are also low by default, all of the displays will light up when the board is reset. This will cause a considerable current draw through the microcontroller. Setting the mux bits high ensures that everything will be off (not enabled).

```
void setup()
{
   // set Arduino pins 0 through 7 (port D.0:7) for output displays
   DDRD = 0xff;

   // set Arduino pins 8 through 10 (port B.0:2) for output to mux LEDs
   DDRB |= DIGIT111;

   // Displays are active low so turn them off initially
   PORTB |= DIGIT111;

   // Arduino pin 11, port B.3 for input from FSR
   DDRB &= (~FSRMASK);   // initialize the pin as an input.
   PORTB |= FSRMASK;     // enable pull-up

   // get seed value for random- noise voltage at pin A0
   randomSeed(analogRead( 0 ));
}
```

Here is our main loop, slightly altered. Compare it line-by-line to the original version and note any similarities and alterations. The comments alone should provide sufficient detail. The most obvious changes are the removal of the Serial Monitor code and the inclusion of a new function called DisplayValue() which will be discussed shortly.

```
void loop()
{
   unsigned long starttime, finishtime;
   int i, j, nettime;
   long a;


   for(i=0;i<5;i++)
   {
      a = random(1000, 10000);

      // wait a couple of seconds to start this round
      delay(2000);

      // flash display a few times to tell player to get ready
      for(j=0;j<5;j++)
      {
         DisplayValue( 888, 100 );
         delay(100);
      }

      // delay random amount
      delay(a);

      // flash LED for real
      starttime = millis();
      DisplayValue( MSG_GO, 50 );

      // wait for response, pressing FSR makes a low
      while( PINB & FSRMASK );
```

```
        finishtime = millis();
        nettime = finishtime - starttime;

        // check for cheat
        if( nettime < 100 )
            nettime = MSG_ERR;

        DisplayValue( nettime, 3000 );
    }
}
```

Functionally, `DisplayValue()` is fairly straight-forward to use. The first argument is the number to be displayed, from 0 through 999. The second value is the duration of the display in milliseconds. So `DisplayValue( 888, 100 )` means "display the number 888 for 100 milliseconds". This particular line makes the "get ready" flash. `DisplayValue()` also allows for a few special values. These are encoded as negative values. A negative response time is impossible, of course, without some form of time travel ability. If the value is set to `MSG_ERR` then the display shows "Err". If the value is set to `MSG_GO` then the display shows "go". Thus the former flashing of the green "go" LED is replaced with `DisplayValue( MSG_GO, 50 )`, producing the "go" message for 50 milliseconds.

The first thing the function should do is validate the value passed to it. It needs to check for the special message codes as well as values out of range. Whatever value is passed, we must derive three new values, namely the indices for the `numeral` array. These will be stored in the variables `h`, `t` and `u` which stand for hundreds place, tens place and units place. The special cases should be checked first. Beyond that the value needs to be broken down into its individual digits through the use of integer divides and mods. Successive integer divisions by 10 will effectively shift the number down a place while mod by 10 will leave the remainder. Think of a three digit number at random and verify that the logic works.

```
void DisplayValue( int v, int msec )
{
    unsigned char i, h, t, u;


    if( (v <= MSG_ERR) || (v > 999) ) // error code
    {
        h = LETTER_E;
        t = u = LETTER_R;
    }
    else
    {
        if( v == MSG_GO )
        {
            h = LETTER_G;
            t = LETTER_O;
            u = LETTER_BLANK;
        }
        else
        {
            u = v%10;
            v = v/10;
            t = v%10;
            h = v/10;
        }
    }
```

At this point we need to turn the requested duration into loop iterations. As coded, each iteration lasts about 15 milliseconds given that each digit is illuminated for 5 milliseconds. A simple division will suffice although we may wish to guard against someone accidentally requesting just a few milliseconds (which would yield zero).

```
// turn millisecs into loop iterations
msec = msec/15;
if( msec < 1 )
    msec = 1;
```

We now create a loop that will roughly yield the requested duration with 15 millisecond resolution. Inside the loop, first we clear all of the displays. Then we enable the desired display and blast in the code for the numeral. Note that by entering the binary values into the array in ascending order we can just use the computed digit value as the index into the array. We don't need to translate the desired numeral into the appropriate index. A little forethought can save coding effort and memory space. Each digit is held for 5 milliseconds. At the close of the function we turn off the display.

```
// display the value for the specified time
for( i=0; i<msec; i++ )
{
    // clear all displays then activate the desired digit
    PORTB |= DIGIT111;
    PORTD = numeral[h];
    PORTB &= ~DIGIT100;
    delay(5);

    PORTB |= DIGIT111;
    PORTD = numeral[t];
    PORTB &= ~DIGIT10;
    delay(5);

    PORTB |= DIGIT111;
    PORTD = numeral[u];
    PORTB &= ~DIGIT1;
    delay(5);
}

// clear display
PORTB |= DIGIT111;
```

Assignment

This exercise involves a considerable amount of wiring and a non-trivial amount of code. There are a lot of things that can go wrong if you just dive in and try to do everything at once. For example, if a segment or an entire digit fails to light, how do know whether the problem is in the hardware or in the software? Debugging both simultaneously is no minor task. It is better to make sure that chunks of the code work as expected first, then build on that. For example, you might just code the chunk that lights one display then expand to a multiplexed display. Consider something like this:

```
// Port B.0:2 for 7 segment mux
#define DIGIT1   0x01
#define DIGIT10  0x02
#define DIGIT100 0x04
#define DIGIT111 0x07

#define FSRMASK  0x08

unsigned char numeral[]={
   //ABCDEFG,dp
   0b00000011,   // 0
   0b10011111,   // 1
   0b00100101,   // 2
   0b00001101,   // 3
   0b10011001,
   0b01001001,
   0b01000001,
   0b00011111,
   0b00000001,
   0b00011001,   // 9
   0b11111111,   // blank
   0b01100001,   // E
   0b01110011,   // r
   0b00001001,   // g
   0b00111001    // o
};

#define LETTER_BLANK  10
#define LETTER_E      11
#define LETTER_R      12
#define LETTER_G      13
#define LETTER_O      14
#define MSG_ERR       -2


void loop()
{
   // try a bunch of different things...
   DisplayValue(123);
   DisplayValue(456);
   DisplayValue(12);
   DisplayValue(3);
   DisplayValue(100);
   DisplayValue(-2);
   DisplayValue(50);
}
```

```
void DisplayValue( int v )
{
   unsigned char i, h, t, u; // hundreds, tens, units

   if( (v <= MSG_ERR) || (v > 999) ) // error code
   {
      h = LETTER_E;
      t = u = LETTER_R;
   }
   else
   {
      u = v%10;
      v = v/10;
      t = v%10;
      h = v/10;
   }

   // display the value for approx 1 sec (66x15msec)
   for( i=0; i<66; i++ )
   {
      // clear all displays then activate the desired digit
      PORTB |= DIGIT111;
      PORTD = numeral[h];
      PORTB &= ~DIGIT100;
      delay(5);

      PORTB |= DIGIT111;
      PORTD = numeral[t];
      PORTB &= ~DIGIT10;
      delay(5);

      PORTB |= DIGIT111;
      PORTD = numeral[u];
      PORTB &= ~DIGIT1;
      delay(5);
   }

   // clear display
   PORTB |= DIGIT111;
}
```

Once you have the circuit working, try it for real! Turn in your code and a completed schematic. Also, comment on what code and hardware alterations would be needed (if any) if common cathode displays were used instead of common anode.

12
Arduino Analog Output via PWM

In this exercise we shall examine analog output through the use of PWM or pulse width modulation. PWM is a method to produce analog signals using a digital output rather than a true continuously variable output. While some microcontrollers and boards have true DACs built-in (the Arduino Due being one example), most do not. Therefore, a digital pulse output is used with a variable duty cycle. The "analog" signal basically is the "area under the curve". If you thought of averaging the pulse train with a low pass filter, a small duty cycle would achieve a low value and a high duty cycle would achieve a high value. Some loads may need proper filtering of the pulse train but some do not. In this exercise we shall begin by examining a simple LED.

It is easy enough to flash an LED on and off. Consider the code below:

```
/* PWM V1 Manual LED brightness test */

#define LEDBIT    0x40

// on/off time in msec
#define LEDONTIME   300
#define LEDOFFTIME  300

void setup()
{
      // set Arduino pin 6 for output to an LED
      DDRD |= LEDBIT;
}

void loop()
{
      PORTD |= LEDBIT;     // turn on LED
      delay(LEDONTIME);

      PORTD &= (~LEDBIT); // turn off LED
      delay(LEDOFFTIME);
}
```

In this example the LED flashes on and off 300 milliseconds each for a total cycle time of 600 milliseconds. Attach a standard NPN LED driver circuit to power, ground and Arduino pin 6. Enter the code, compile, transfer and run it to verify. Reduce the on/off times to 30 milliseconds each and repeat. The LED should flash much faster. Reduce the on/off times to 5 milliseconds each and repeat. A curious thing happens. The LED doesn't seem to be flashing anymore; rather it just seems dimmer. Once the timing drops below about 20 milliseconds, the human eye integrates the visual input over time and we no

longer see discrete flashes[3]. Instead, we see the average illumination. Since the LED is only on for half the total time, it appears dimmer than a single long flash. You can verify this by changing the on time to 1 millisecond and the off time to 9 milliseconds, still achieving a 10 millisecond cycle time. The LED should appear quite dim as the average level is now only 10 percent of the maximum.

Instead of adjusting the timing manually, we can use the library function `analogWrite()` on one of the PWM capable output pins (these are marked with a "~" next to the header). Fortunately, pin 6 is PWM capable so all we need is a little change to the code:

```
/* PWM V2 LED brightness using analogWrite */

#define LEDPIN 6

// brightness 0 to 255
#define LEDBRIGHTNESS 230

void setup()
{
}

void loop()
{
      analogWrite(LEDPIN,LEDBRIGHTNESS);
}
```

Note that pin 6 does not need to be set for output mode as this is done automatically by the `analogWrite()` function (see "Bits & Pieces: analogWrite" in the text for details). If you placed an oscilloscope at pin 6 you'd see a square wave at about 490 Hz with 90 percent duty cycle. Note that the LEDBRIGHTESS parameter can be replaced with the results from an `analogRead()` call. So, we can connect a potentiometer to an analog input pin and read its voltage which will range from 0 to 1023. We then scale this range onto the 0 to 255 (unsigned char) input of `analogWrite()` to create a dimmer. The scaling can be performed using the Arduino library function `map()`. The template is:

```
result = map(value, fromLow, fromHigh, toLow, toHigh);
```

In our case we'd use:

```
result = map(value, 0, 1023, 0, 255);
```

To continue, wire a 10 kΩ potentiometer between power and ground, connecting the wiper arm to Arduino pin A0. Enter the code below, compile and transfer it.

```
/* PWM V3 LED dimmer using analogWrite and potentiometer */

#define ANALOG_IN_PIN 0
#define LEDPIN 6
```

---

[3] If this was not the case we would not get the impression of continuous motion while watching TV or movies which use frame rates of 30 and 24 frames per second respectively. The frames "blur together" in our brains creating the illusion of continuous motion.

```
void setup()
{
      analogReference( DEFAULT );
}

void loop()
{
      int a;

      a = analogRead( ANALOG_IN_PIN );

      a = map(a,0,1023,0,255);
      analogWrite(LEDPIN,a);
}
```
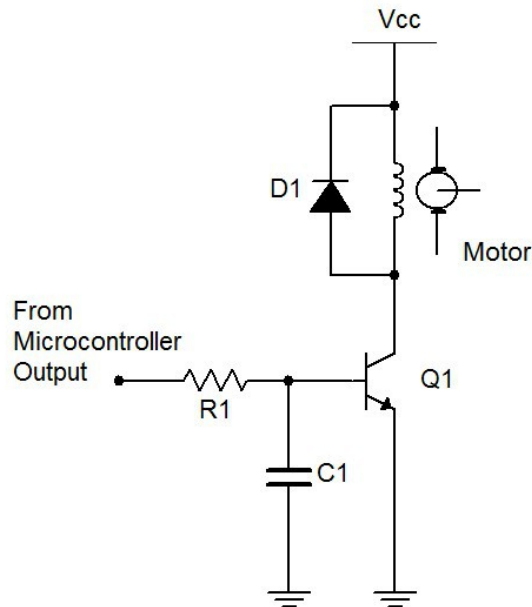
LEDs are all well and good but we might wish to control something a little more interesting such as a motor. Obviously, our LED driver circuit will be insufficient for any decently sized motor but it will do just fine for a small 10–15 milliamp (unloaded) DC hobby motor given some minor modifications. An example is seen in Figure 12-1.


Motor controller with BJT

Figure 12-1

First note that a higher supply is required. In this case, 15 VDC would be appropriate. R1 would need to be about 6.8 kΩ to guarantee transistor turn-on and C1 needs to be about 100 nF. D1 should be a 1N4002 rectifier. It is used to prevent, or snub, a sudden voltage spike caused by the motor's inductance when the transistor switches state between saturation and cutoff[4]. For a larger load, a power FET can be used as shown in Figure 12-2. A nice load for this variation would be a common 12 volt boxer fan (of one or two

---

[4] Remember, the current through an inductor cannot change instantaneously. The sudden transistor switching would result in a large inductor reverse voltage which would in turn cause a large voltage spike across the transistor due to KVL. So the diode is placed there to limit said voltage and protect the devices. See: http://en.wikipedia.org/wiki/Flyback_diode

watts dissipation). The ZVN4206A EMOSFET would be a decent choice for this larger load as it handles 600 milliamps continuous and 8 amps pulsed (set Vdd to 12 VDC).
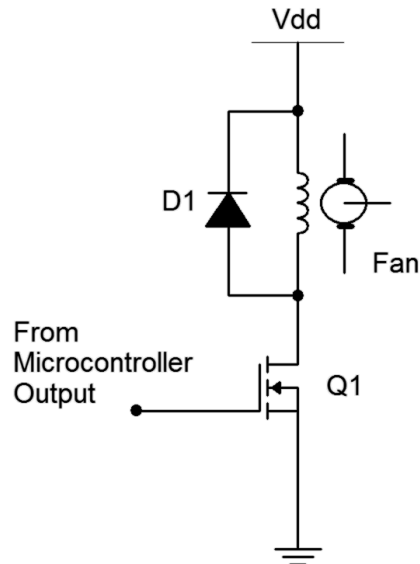
Fan controller with EMOSFET

Figure 12-2

Build one of the control circuits and connect it to Arduino pin 6 in place of the LED driver. (You might want to keep the LED driver wired up for upcoming modifications and additions.) There is no need to change the code. Once the motor circuit is connected, its speed should now be controlled by the potentiometer.

Assignment

Sometimes we need a form of visual feedback regarding the control variable; in other words, some manner of readout. In this case, we can make a simple "LED bar" to indicate the motor speed; the faster the speed, the wider the bar. There is no need to re-invent the wheel as we've done this before. Consider adapting the `lightLEDsBar()` function from the Arduino Analog Input exercise. It will require four LED drivers connected to port B.

Finally, let's consider setting the motor speed in accordance to some environmental variable such as temperature or light level instead of using a manually controlled potentiometer. The potentiometer can be replaced with a simple sensor such as the LM34 for temperature or a resistor and photoresistor (CdS cell) voltage divider to sense light levels. It will be relatively easy and safe to create extreme changes in light level using a flashlight and a finger (to cover and block the light). The range of the temperature sensor voltage will be comparatively narrow, so a different range of mapping will probably be in order. You may also wish to mount the temperature sensor away from the other circuit elements in order to isolate them from excessive heat and cold. No open flames should be used in lab to create heat!

Include your code and a proper schematic, as usual, along with a description of how you tested the result.

13

Arduino Event Counter
or Revenge of the Son of the Reaction Timer Redux

Often it seems we need our microcontroller to be doing several things at once such as monitoring inputs while controlling changing outputs. A single controller cannot do this in the literal sense but it can operate so quickly that it appears to be doing several things simultaneously. Sometimes, though, this can get quite complicated if polling or busy-wait style code is used. For example, consider the operation of multiplexing several seven segment displays. Unlike a simple LED which merely requires us to set or clear a port bit, the multiplexed displays need constant attention. In the Reaction Timer Redux exercise the displays required 15 milliseconds per rendering; 5 milliseconds for each of the 3 displays. If we wanted to display the value for a few seconds we simply looped around that code. Although this worked well for that application, such is not always the case. Consider the case of an event counter. This is a device that counts things and displays the current value. It might be used on an assembly line to keep track of the number of items that have passed by, perhaps via the item triggering a mechanical, photo-electric, magnetic or other kind of switch. We would like this device to present a display of the current count at all times yet we also need to be constantly checking the item switch. How do we do both? In the case of the Reaction Timer we could add code within the display loop that checks the item switch but if the switch activates and deactivates in under 15 milliseconds we could miss it while the displays are being written to. Fortunately, there is a nice solution to this problem.

We Interrupt Our Regularly Scheduled Code...

The solution is the use of an *interrupt*. Think of an interrupt as a scheme whereby a special bit of code called an interrupt handler or *interrupt service routine* (ISR) is triggered to run under special conditions. The execution of the existing code is suspended until the ISR is done. The interrupt can be triggered by either software or hardware events. A typical example would be to have the interrupt triggered by a signal change on an input port pin. Most microcontrollers have many different levels of interrupts and these can be sorted by importance. That is, some events can take precedence over other events. This means that it may be possible to have one ISR interrupted by a higher priority interrupt. Generally, ISRs should be short, quickly executing chunks of code so as to not clog-up the operation of the main code. In the case of our event counter, the item switch could be connected to an input pin that triggers an interrupt. The ISR would do nothing more than increment a global variable that keeps track of the item count. The main body of the code would then periodically check this global and respond accordingly (in this case, updating the display). We can think of the ISR scheme as "running in the background" constantly checking the input pin. This frees us from having to sprinkle port checking code throughout the main body of the program.

Ordinarily, none of the available interrupts are active with the exception of the system reset. These have to be enabled individually before they can be used. The development system defines a name for each interrupt routine so all we have to do is enable the proper bit(s) in the interrupt control register(s) and fill out the code for the ISR.

For the event counter, we're going to reuse a considerable amount of code and hardware from the Reaction Timer Redux exercise, most notably the seven segment displays and multiplexing code. The player's switch will be traded for the event counting switch. We shall make use of the ATmega 328P's *pin change interrupt* (PCINT) capability. When programmed, each of the input port pins can trigger an interrupt when its input signal changes from high to low or low to high. These are arranged into banks which correspond to the ports, for example bank 0 corresponds to port B. Each bank has an ISR associated with it. The bank 0 ISR is called PCINT0_vect. This stands for "Pin Change Interrupt Bank 0 Vector" (technically, a vector is a function address so this is really the starting address of the interrupt service routine). Each of the pins of a particular port is activated through its own interrupt bit. For bank 0, these bits are PCINT0 through PCINT7. So, if port pins 0 and 1 are enabled for interrupts (via PCINT0 and PCINT1), a change on either of their pins will generate the bank 0 pin change interrupt and cause the PCINT0_vect ISR to execute. Once the ISR is done, normal execution of the main code will continue where it left off.

Before we look at hardware let's consider some code. As mentioned, we can modify the existing Reaction Timer Redux code. We will no longer need the special "go" message so we can delete some items from the numeral array. Other modifications will be minimal if we keep the wiring the same. We'll use the generic term "sensor" in place of the former FSR as we have yet to determine how the event counting switch will be implemented. Note that having some documentation inside the original code as to how the circuit is wired saves us from re-inventing the wheel. All good programmers, technicians and engineers do this!

```
/* Event counter with common anode 7 segment displays X 3. Displays off of
D1:7 (segment a=7), Mux off of B0:2 (hundreds on B.2), w/ PNP drivers. Sensor
on B.3. Values > 999 show up as "Err" ALL SEGMENTS AND MUX ACTIVE LOW */

// Port B.0:2 for 7 segment mux
#define DIGIT1   0x01
#define DIGIT10  0x02
#define DIGIT100 0x04
#define DIGIT111 0x07

#define SENSORMASK  0x08

unsigned char numeral[]={
   //ABCDEFG,dp
   0b00000011,   // 0
   0b10011111,   // 1
   0b00100101,   // 2
   0b00001101,   // 3
   0b10011001,
   0b01001001,
   0b01000001,
   0b00011111,
   0b00000001,
   0b00011001,   // 9
   0b11111111,   // blank
   0b01100001,   // E
   0b01110011    // r
};

#define LETTER_BLANK  10
#define LETTER_E      11
#define LETTER_R      12
```

At this point we need to declare the afore-mentioned global variable and modify the `setup()` routine.

```
unsigned int g_count = 0;

void setup()
{
   // set Arduino pins 0-7, port D.0:7, for output to 7 segment LEDs
   DDRD = 0xff;

   // set Arduino pins 8-10, port B.0:2, for output to mux 7 segment LEDs
   DDRB |= DIGIT111;

   // Displays are active low so turn them off initially
   PORTB |= DIGIT111;

   // Arduino pin 11, port B.3 for input from sensor
   DDRB &= (~SENSORMASK);   // initialize the pin as an input.
   PORTB |= SENSORMASK;     // enable pull-up


   // enable pin change interrupt bank 0
   bitSet(PCICR,PCIE0);

   // enable pin change interrupt on port B.3
   bitSet(PCMSK0,PCINT3);
}
```

The last two lines of the code above are particularly important. The first line enables the pin change bank 0 interrupt (bit `PCIE0` or Pin Change Interrupt Enable bank 0) in the Pin Change Interrupt Control Register (`PCICR`). The second line then specifies which pin(s) in bank 0 will be used (Pin Change INTerrupt pin 3 or `PCINT3`) in the Pin Change MaSK for bank 0 (`PCMSK0`) register.

Once enabled, we now need to write the ISR itself. As mentioned, this routine already has a name (`PCINT0_vect`) and all it needs to do is increment the global count variable.

```
/* This is the bank 0 interrupt handler. It triggers on any change to the
appropriate pin(s) of bank 0 (port B), either H->L or L->H. */

ISR(PCINT0_vect)
{
   // increment only on sensor going low to avoid double counting
   if( !(PINB & SENSORMASK) )
      g_count++;
}
```

The former `loop()` function can be tossed. In its place we use the contents of the former `DisplayValue()` function that controlled the display multiplexing. Unlike the original, this does not render the display for a specified time. Rather, it does a single scan of the three displays for a total of 15 milliseconds. `loop()` will then be called again and the process repeats.

At the start, the contents of the global count variable are copied to a local variable which is then taken apart to determine the indices into the `numeral` array as before. Note a minor refinement, namely the snippet of code added to "blank" leading zeros. A question might remain though, and that is, why bother

to make a local copy of the global variable? The reason is because **an interrupt can happen at any time**. That means that partway through the h, t, u index code an interrupt might occur and the global variable will change. If this happens the computed indices will be incorrect because the code will be operating on two (or possible more) different values. If we make a local copy this won't happen. Only the global will change and the display will be updated appropriately the next time `loop()` is called.

```
void loop()
{
   unsigned int v;
   unsigned char h, t, u; // hundreds, tens, units

   // grab a local in case another interrupt comes by
   v = g_count;

   if( v > 999 ) // error code
   {
      h = LETTER_E;
      t = u = LETTER_R;
   }
   else
   {
      u = v%10;
      v = v/10;
      t = v%10;
      h = v/10;

      // blank leading zeros
      if( !h )
      {
         h = LETTER_BLANK;
         if( !t )
            t = LETTER_BLANK;
      }
   }

   // clear all displays then activate the desired digit
   PORTB |= DIGIT111;
   PORTD = numeral[h];
   PORTB &= ~DIGIT100;
   delay(5);

   PORTB |= DIGIT111;
   PORTD = numeral[t];
   PORTB &= ~DIGIT10;
   delay(5);

   PORTB |= DIGIT111;
   PORTD = numeral[u];
   PORTB &= ~DIGIT1;
   delay(5);

   // clear display
   PORTB |= DIGIT111;
}
```

At this point we have some workable code. If the earlier hardware is not still in place, rewire it. Enter the code above, compile and transfer it. Each press of the player button should show an increment on the display. Note: If the switch was not debounced you might get an extra count or two on random presses.

The Switch

The trick now is to implement an appropriate switch to count the items or events in question. All it needs to do is make or break contact, or produce a high-low-high voltage transition. Let's assume we're counting items passing by on a conveyor belt. The implementation will depend on the items themselves. Are they all the same size and orientation? How heavy are they? Are they randomly placed and timed? In some cases a simple mechanical spring-mounted "flip" switch can be used: The item brushes up against the switch and pushes it to the side, activating it. The spring then restores the switch to the off position to await the next item. If the items are small, irregularly positioned or shaped, this technique may have difficulty. Another approach is to use a photo-electric scheme. Here, a beam of visible or infrared light is shown across the conveyor belt. Opposite it is a detection device. When an item passes by, the beam is broken and a signal generated. This scheme will work with oddly shaped or placed items as well as very light items as no physical contact is made. Not everything will work with this, though. Possible examples of difficult items include aquariums and thinly sliced chunks of Swiss cheese (really, any oddly shaped device with gaps might cause multiple triggers).

Mechanical switching will require some fabrication but the idea is straightforward enough. For photo-electric, there are a few options. One possibility is the use of a light dependent resistor or CdS cell and a strong light source. The CdS cell will exhibit high resistance under low light and the resistance will decrease as light levels increase. The CdS cell could be connected directly to an input port pin with the internal pull-up resistor enabled. A major challenge here is the ambient light. Another possibility is an infrared emitter-detector pair (IR LED plus IR phototransistor). This would require a little more circuitry but would tend to be less sensitive to ambient light conditions.

Alignment of photo-electric devices and their separation distance can be critical to proper and consistent triggering. For ease of testing in lab, it is best to focus on "proof of concept", keeping the devices only an inch or two apart and simulating the items to be counted by passing a black card between the sensor/detector pair.

Assignment

Remove the former player switch and insert your new item counting switch in its place. Also, add a second switch that will reset the item count back to zero (it is not acceptable to simply use the board reset switch). Include a description, rationale and schematic of the item counting switch.

14

Arduino Arbitrary Waveform Generator

This exercise combines much of the preceding work, namely sensing digital input, creating digital output, timing, etc. A major new element is producing analog output voltages without resorting to pulse width modulation schemes. Once completed, we will have a working low frequency waveform generator with at least three waveforms to choose from (sine, square and ramp with more wave shapes possible) and adjustable frequency. The quality of the sine will be at least as good as those from typical analog function generators (measured THD < 1%). An exercise of this magnitude will cover a lot of ground and take considerably more time to work out software issues and wire up the corresponding hardware. A methodical approach to the problem will be invaluable.

Let's start with the core concept of *direct digital synthesis*. Instead of using analog oscillators, we can make waveforms by feeding appropriate data values to a digital to analog converter (DAC). We could obtain these values by digitizing an analog signal or via computation. Whatever the source of the data, we can store them in an array and send them one at a time to a DAC. Once we reach the end of the array, we loop back to the beginning of the array and repeat the process. This array is usually called a *wave table*. Theoretically, given a large enough table we can reproduce any wave we desire. The obvious question then is how large is large enough? Quite simply this will depend on the complexity of the wave, the accuracy desired and other practical considerations. For something like a sine wave, reasonable quality results can be obtained with a few hundred 8 bit values. For very high quality, a few thousand 16 bit data values may be used.

Given something as simple as a sine wave, the question arises as to why we even bother with a wave table. After all, we could compute each data point directly using the `sin()` function. While this would probably save some memory (the computational code taking up less space than the wave table), it would be much too slow. Direct computation of 8 bit data using the `sin()` function on the Arduino Uno requires about 140 microseconds per data point. If the wave table contained just 100 points this would lead to a period of 14 milliseconds, or a maximum frequency of only 71 hertz. In contrast, accessing data from a pre-computed table and writing it to an output port takes a fraction of a microsecond. Even with a larger table we could achieve output frequencies in the kilohertz range.

So how do we fill the table? First, for practical reasons, wave tables are usually a power of two in size. 256 elements is a nice size because we can using an `unsigned char` for the index and simply increment it continuously. Once it hits 255, the next increment will cause an overflow and the value returns to zero, thus removing the need for a limit test. While it's possible to compute the wave table in the `setup()` function, that will eat up some memory. It is better to do it separately. Although we could grab a calculator and churn out the values manually one by one, it would be much quicker to write a little program that creates a simple text file. We can then copy and paste this into the C program. Python is a good choice for just such a programming task.

Parte Uno – A Simple Sine Generator

Here is a short Python program that will create a sine table for us. Each data point is an unsigned char so they range from a minimum of 0 to a maximum of 255 with an effective DC offset of 128. This will work perfectly with a simple unipolar 8 bit DAC.

```python
# Python 3.3 wavetable creator for C language data array using unsigned bytes
# 256 entry wave table, amplitude range is 0-255
# Note: Rename array as needed

import math

fn = input("Enter name of the wavetable file to be created: ")
fil = open( fn, "w+" )
fil.write( "unsigned char sinetable[256]={" )

for x in range( 256 ):

    # each data point is 1/256th of a cycle, peak value is 127
    f = 127.0 * math.sin( 6.28319 * x / 256.0 )

    # turn into an integer and add DC offset
    v = int( round( f, 0 ) ) + 128

    # guarantee no out-of-bounds values just in case
    if v > 255:
        v = 255
    if v < 0:
        v = 0

    if x < 255:
        fil.write( str(v) + ", " )
    else:
        fil.write( str(v) + "};" )

fil.close()
```

To generate the sine table we simply run this program, enter an appropriate file name when prompted, and literally wait a second for it to process. We then open the resulting file with a text editor and paste the contents into the Arduino editor. We can use this program as a template to generate other wave tables as needed.

Now that we have the wave data, what sort of circuitry will we use for the DAC? Obviously, we can use an off-the-shelf 8 bit DAC but we can also achieve workable results by using a simple R-2R ladder network as shown in Figure 12-1. R might be around 10 kΩ thus making 2R 20 kΩ. While resistor tolerance and port output voltage fluctuations will add some non-linearity, the results for 8 bit data are good enough for now.

Choose an appropriate value for R and build the ladder network. Wire the bits to port D of the Arduino Uno. Port D corresponds to pins 0 (LSB) through 7 (MSB). The Analog Output connection should be monitored with an oscilloscope.
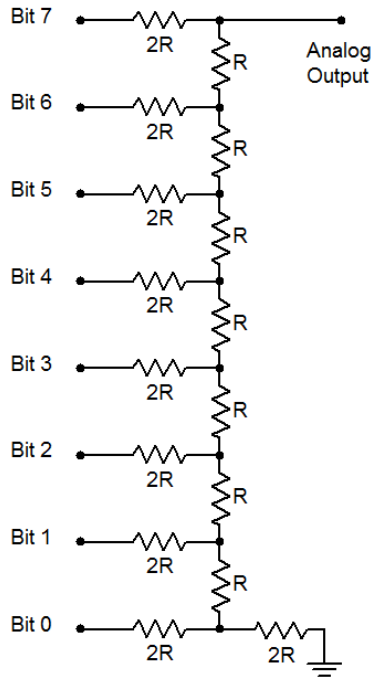
R-2R Ladder

Figure 13-1

The code to generate the sine wave is fairly straightforward, in fact it is actually quite sparse. Most of the space is taken up by the data array. Here is the first version:

```
unsigned char sinetable[256]={128, 131, 134, 137, 140, 144, 147, 150, 153,
156, 159, 162, 165, 168, 171, 174, 177, 179, 182, 185, 188, 191, 193, 196,
199, 201, 204, 206, 209, 211, 213, 216, 218, 220, 222, 224, 226, 228, 230,
232, 234, 235, 237, 239, 240, 241, 243, 244, 245, 246, 248, 249, 250, 250,
251, 252, 253, 253, 254, 254, 254, 255, 255, 255, 255, 255, 255, 255, 254,
254, 254, 253, 253, 252, 251, 250, 250, 249, 248, 246, 245, 244, 243, 241,
240, 239, 237, 235, 234, 232, 230, 228, 226, 224, 222, 220, 218, 216, 213,
211, 209, 206, 204, 201, 199, 196, 193, 191, 188, 185, 182, 179, 177, 174,
171, 168, 165, 162, 159, 156, 153, 150, 147, 144, 140, 137, 134, 131, 128,
125, 122, 119, 116, 112, 109, 106, 103, 100, 97, 94, 91, 88, 85, 82, 79, 77,
74, 71, 68, 65, 63, 60, 57, 55, 52, 50, 47, 45, 43, 40, 38, 36, 34, 32, 30,
28, 26, 24, 22, 21, 19, 17, 16, 15, 13, 12, 11, 10, 8, 7, 6, 6, 5, 4, 3, 3,
2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 3, 3, 4, 5, 6, 6, 7, 8, 10, 11, 12,
13, 15, 16, 17, 19, 21, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 43, 45, 47,
50, 52, 55, 57, 60, 63, 65, 68, 71, 74, 77, 79, 82, 85, 88, 91, 94, 97, 100,
103, 106, 109, 112, 116, 119, 122, 125};


void setup()
{
     // set all of port D for output
     DDRD = 0xff;
}
```

Laboratory Manual for Embedded Controllers

```
void loop()
{
      unsigned char u=0;
      unsigned char *p;

      p = sinetable;

      // loop forever
      while( 1 )
      {
            // set data and wait one sample period
            PORTD = p[u++];
            delayMicroseconds(100);
      }
}
```

The code first sets all of port D to output mode. The `loop()` function consists of a `while()` loop that never terminates. Interestingly, `loop()` doesn't loop at all, it is only entered once. The contents of the `loop()` function could just as easily be placed in the `setup()` function but they are placed here simply as a means of visually separating the initialization code from the looping code.

A pointer is used to identify the start of the wave table. Strictly speaking this is not necessary here as we could just as well use `sinetable[]`. The reason for this will become apparent in a moment. The first value of the wave table array is copied to port D. The ladder network transforms the eight 0 volt/5 volt combinations into a single voltage. Notice the efficiency of the single write to the port when compared to the bit oriented function `digitalWrite()` examined earlier. Trying to write a single bit at a time would create a complete mess as the various bits would not all change together but would create seven wrong values leading up to the correct value once all eight bits had been written. In any case, after the value is written, the program pauses for 100 microseconds. This is the sample period. After the pause, the next array value is copied to the port and the process repeats. Ignoring the looping and accessing/writing overhead, a single pass of the table requires 256 entries times 100 microseconds each, or 25.6 milliseconds. This is an output frequency of approximately 39 Hz.

Enter the code, compile and transfer it the board. Hook up an oscilloscope to the Analog Output connection. You should see a fairly nice 39 Hz sine wave of about 5 volts peak to peak riding on a 2.5 volts DC offset. Zoom in on the time base to inspect the wave. The individual stair steps of the R-2R ladder should be apparent. Also, the timing of 100 microseconds per step should be obvious.

We can increase or decrease the frequency by adjusting the value fed to the `delayMicroseconds()` function. Try a higher value and see what you get. 10 microseconds should yield a frequency of about 390 Hz. Using this frequency, measure the residual THD of the waveform. With typical 5% resistors the total harmonic distortion will likely be less than 1%. A portion of this is due to the variance of the resistors as well as the port pin output voltages not all being identical.

A different wave shape can be generated and added to the program. The only change to the body of the program would be to change what the pointer `p` is referencing. For example, you could copy and paste the sine table, change a few of the values by extreme amounts so they're obvious to the eye, and then change the name to something like `wackysinetable[]`. Finally, change the pointer reference to `p=wackysinetable;` Try this. If available, you might consider connecting the output to a small amplifier and headphones and listen to the newly altered wave. Why, it's hours of fun for the making! Call your spouse/bf/gf/so, get the kids or siblings, grab the neighbors and unleash the dog; they'll all agree that it's way better than watching another inane game show, "reality" series or TV evangelist.

Parte Due – Changing Wave Shape

While it can be fun to change the delay, edit the wave tables by hand and then listen to the results, what would be more useful would be to control both the frequency and wave shape with some external controls. This is more in keeping with a standard lab function generator.

How might we offer the user control over the wave shape? Many lab function generators use a set of interdependent push-buttons where only one button can be depressed at a time. This offers nice visual and tactile feedback. A simpler and less expensive technique utilizes a single button that cycles through the choices, flipping back to the first choice once the last one is reached. This is particularly effective if the number of choices is fairly small. We could couple this to a series of LEDs to indicate the selected item. Behind the scenes a single variable will hold the "state" of the selection and will be updated as the switch as activated. This switch should be debounced in hardware as a software debounce will require too much time. We can check the state of this switch at the start of each cycle. This has two potential advantages over checking it after each value is written to the output port. The first advantage is less processing. The second advantage is that if we set up the tables correctly, we can guarantee that wave shape switches will always occur at a positive slope zero volt crossing thus avoiding potentially dangerous transients.

Let's use three wave shapes: sine, square and ramp. We're going to need three LEDs to indicate the shape so that's three output pins. We'll also need an input pin for the shape switch. Although our wave shape variable could just use the numbers 1, 2 and 3 to represent these three waveforms, with a little forethought we can save ourselves some work. Why not just use the corresponding LED port bit value to represent the wave choice? Doing so means there will be no translation needed between the wave shape choice and which LED to light. Let's use port B.1 (port B bit 1) for the input switch and B.2:4 for the sine, square and ramp LEDs, respectively. We could use the following:

```
// all port B
#define WAVEBIT        0x02
#define SINEWAVE       0x04
#define SQUAREWAVE     0x08
#define RAMPWAVE       0x10
```

We then need to modify the `setup()` function with the following:

```
// set B.2:4 for output
DDRB |= (SINEWAVE | SQUAREWAVE | RAMPWAVE);

// set B.1 for input, enable pull-up
DDRB &= (~WAVEBIT);
PORTB |= WAVEBIT;
```

In the `loop()` function we will need several new variables:

```
unsigned char wavechoice=SINEWAVE;
unsigned char waveswitch=WAVEBIT, currentwaveswitch=WAVEBIT;
```

`wavechoice` is the variable that holds the user's current choice of wave shape. It is initialized to sine. It is important to remember that we'll only change the wave shape on a push-button transition, not on it being high or low per se. Thus, we need to know the switch's prior state as well as its current state. The two `waveswitch` variables are used to hold these two states, pressed or released. Instead of setting these to 1/0 (true/false), we use the actual bit value to represent true. This will make coding a little more

transparent. So, before entering the `while()` loop we need to set the wave table pointer to the current default:

```
// select default sine and light corresponding waveshape LED
PORTB |= wavechoice;
p = sinetable;
```

Because the `wavechoice` variable uses the port bit values for the LEDs, we can just copy the value to the output port to light it (remember, by default on reset all other bits are zero); nice and simple.

Now let's take a look at the switch processing. First we need to determine the current state:

```
currentwaveswitch = PINB & WAVEBIT;
```

We will only process the switch on a depressed state (i.e., a high to low transition). Note that if you're using a hardware debounce circuit that inverts, this logic will also be inverted (remove the "not").

```
if( !currentwaveswitch ) // selected (down)
{
      if( currentwaveswitch != waveswitch ) // transition
      {
```

We then check the current state of the `wavechoice` variable and cycle through the available values, updating the generic wave table pointer as well:

```
            if( wavechoice == SINEWAVE )
            {
                  wavechoice = SQUAREWAVE;
                  p = squaretable;
            }
            else
            {
                  if( wavechoice == SQUAREWAVE )
                  {
                        wavechoice = RAMPWAVE;
                        p = ramptable;
                  }
                  else
                  {
                        wavechoice = SINEWAVE;
                        p = sinetable;
                  }
            }
```

We turn off all of the LEDs and activate the newly chosen item:

```
            //turn off all LEDs
            PORTB &= (~(SINEWAVE| SQUAREWAVE| RAMPWAVE));

            // turn on newly selected LED
            PORTB |= wavechoice;
      }
}
```

Lastly, we update the `waveswitch` variable so our transition detection logic properly works:

```
waveswitch = currentwaveswitch;
```

The only thing left is to simply place an `if()` clause around this chunk of code so that we only check the switch at the start of the wave table. This occurs when the index variable `u` is zero:

```
if( !u )
{
     // wave shape switch code goes here
}
```

Create two new wave tables for the square and ramp, and add the code above. Also wire in the three LEDs and wave shape switch. Compile, transfer and test the operation using an oscilloscope. The Python program presented earlier can be used to create the new arrays. The square wave consists of 128 entries of 255 followed by 128 entries of 0. The ramp wave consists of 256 entries starting at 0 and incrementing by 1 each time to end at the value 255.

Parte Terza – Changing Frequency

Changing the wave shape is certainly useful. What about giving the user control over the frequency? At first thought, it seems obvious to use a potentiometer for a frequency control: We could read a voltage off it using the `analogRead()` function and use that for the delay value. The problem with this is the same as direct computation of a sine value, namely lack of speed. The acquisition time of the ADC will severely limit the output frequency we can create. On the other hand, checking a switch for high/low is a very fast process. It would be a simple matter to wire in another switch, similar to the wave shape switch, which would increment a global variable at each push of the button. This variable would then be used by the `delayMicroseconds()` function as follows:

```
unsigned int g_period=100; // init at 100 microseconds

// set data and wait one sample period
PORTD = p[u++];
delayMicroseconds( g_period );
```

When the frequency switch is activated, the delay time would be incremented like so:

```
BumpPeriod( 1 );
```

The `BumpPeriod()` function "hides" the global variable and also allows for constraining its range with wrap-around at the extremes:

```
void BumpPeriod( char c )
{
     g_period += c;

     // wrap within range, c could be negative
     if( g_period < 1 )      g_period = 200;
     if( g_period > 200 )    g_period = 1;
}
```

A delay time of 1 microsecond produces a period of 256 microseconds (ignoring processing overhead) for a maximum frequency of about 3.9 kHz. Similarly, a delay of 200 microseconds yields a minimum frequency of about 20 Hz. Remember, `g_period` is a global variable so it is defined outside of and prior to all functions. In this way all functions can access it.

While we could copy and paste the wave shape code for the frequency switch (with edits, of course), there is a practical problem with it. Every push of the button will change the sample period by just one microsecond. We have a 200:1 period range so the entire span would require 200 discrete button pushes. That's probably not something most people would like. A common solution to this problem is to have the button "auto-increment" if it is held down for a long enough period of time. This hold-off time might be on the order of 500 milliseconds. We don't want the auto-increment to run too fast so we might increment the period one unit for each following 100 milliseconds. That's equivalent to 10 button pushes per second, or the equivalent of way too much espresso. At that rate we can cover the span in 20 seconds just by keeping a finger on the button. This process can be sweetened by adding a decrement button to go along with the increment button. The decrement switch code is virtually the same as what we'll use for the increment except that the call to `BumpPeriod()` passes a negative value. If the span was much larger we might also consider adding "coarse" buttons for increment and decrement which would simply pass a larger value to `BumpPeriod()` (e.g., 10). Another option would be to pass a larger value to the function after a longer time period has elapsed (e.g., 1 each 100 milliseconds if the button is held between 0.5 and 2 seconds, and 10 if it's held longer).

Before we look at the auto-increment feature, we must remember to add code for the frequency switch initialization (along with the switch itself, of course). If we place this switch at Arduino pin 1 (port B.0), our initialization code now looks something like this:

```
// all port B
#define FREQBIT         0x01
#define WAVEBIT         0x02
#define SINEWAVE        0x04
#define SQUAREWAVE      0x08
#define RAMPWAVE        0x10

void BumpPeriod( char c );
unsigned int g_period=100; //init at 100 microsec

void setup()
{
      // set all of port D for output
      DDRD = 0xff;

      // set B.2:4 for output
      DDRB |= (SINEWAVE | SQUAREWAVE | RAMPWAVE);

      // set B.0:1 for input, enable pull-up
      DDRB &= (~(FREQBIT | WAVEBIT));
      PORTB |= (FREQBIT | WAVEBIT);
}
```

The processing of the switch itself is similar to the wave shape switch except that we will need some timing information. After all, the auto-increment feature needs to determine if the initial 500 millisecond hold-off time has passed, and once it has, it must only increment once each following 100 milliseconds. Therefore, along with the variables to hold the current and prior states of the switch, we'll need variables

to hold the time the switch was activated (time of switch transition), the current time, and the auto-increment or "bump" time:

```
unsigned char freqswitch=FREQBIT, currentfreqswitch=FREQBIT;
unsigned long transitiontime, currenttime, bumptime=500;
```

The xxxtime variables must all be unsigned longs as they will be used with the millis() time function.

Here's how the algorithm works: First we obtain the current state of the switch. We do nothing unless the switch is "down". If it's down, we record the current time and see if the prior switch state was "up". If so, this is the initial transition. We record the time of this transition and increment the period. We also set a variable (bumptime) to the hold-off time in milliseconds (500).

```
// check if freq switch active
currentfreqswitch = PINB & FREQBIT;

if( !currentfreqswitch ) // selected (down)
{
      currenttime = millis();
      if( currentfreqswitch != freqswitch ) // transition
      {
            transitiontime = currenttime;
            BumpPeriod( 1 );
            bumptime = 500;   // msec before auto inc
      }
```

If the switch states are the same, then we know the switch is being held. Before we can start the auto-increment, we need to know if it's been held long enough (bumptime, or 500 milliseconds initially). If we simply subtract the current time from the initial transition time we'll know how long the switch has been held. If this difference is at least as big as our target then we can increment the period. Since we don't want to increment it again until another 100 milliseconds pass, we need to increment bumptime by 100.

```
      else  // being held, is it long enough?
      {
            if( currenttime-transitiontime > bumptime )
            {
                  BumpPeriod( 1 );
                  bumptime += 100;
            }
      }
}
```

Lastly and regardless of the state of the switch, we update the prior frequency switch variable to the current state:

```
freqswitch = currentfreqswitch;
```

Notice that once the button is released, it is ignored until it is again depressed. At that instant a new transition time will be recorded and the bump time will be reset back to 500 milliseconds and the process will repeat as outlined above. If the button is released prior to the hold-off time, no auto-increment occurs, and the next button press will reinitialize the transition and bump times. As in the case of the

wave shape switch, it is important to note that if you're using a hardware debounce circuit that inverts, the "switch selected" logic will also be inverted (remove the "not").

The frequency switch processing code should be placed in line with the wave shape switch code, either immediately before or after it. The result inside the `loop()` function should look something like this:

```
        // select default sine and light corresponding wave shape LED

        while( 1 )
        {
                // only check switches at start of each cycle
                if( !u )
                {
                        // process wave switch
                        // process freq switch
                }

                // set data and wait one sample period
                PORTD = p[u++];
                delayMicroseconds( g_period );
        }
}
```

Edit, compile, transfer and test the code. If you're feeling energetic, add a decrement button as well.

Parte Quattro – Using a DAC IC

The R-2R ladder network is certainly workable for this application. It is by no means highly accurate. If higher bit resolutions are desired, the accumulated error from resistor tolerance variation and port pin voltage variation could be significantly greater than the LSB. Also, for proper operation the ladder must be only very lightly loaded. It would be much better if we could isolate the output from the load and increase the system accuracy. These goals can be achieved with only modest rewiring through the use a simple parallel input DAC feeding an op amp buffer. A good example is the DAC0808. A datasheet may be found here: http://www.ti.com/product/dac0808

The DAC0808 is an 8 bit parallel input DAC with current mode output. It requires minimal external circuitry. The eight output pins from the microcontroller are connected directly to the DAC's eight input pins (in our case, the entire R-2R network may be removed). As the DAC uses a current mode output, it is common to use an op amp in the current-to-voltage transducer mode (current controlled voltage source) to produce a stable output voltage. A good example can be found on the data sheet listed as "Typical Application". The LF351 op amp may be substituted with the newer LF411. Also, note that the required power supply connections to the op amp are not shown and an output coupling capacitor may be needed if the output DC offset is to be avoided. Wire up this variation and test it. Also, set the output frequency to 390 Hz using a sine wave and measure the residual THD. Compare the result to the THD measured with R-2R network earlier.

Osservazioni Finali – Final Comments

Assuming a period decrement button was added, all of ports B and D have been used on the Uno leaving us with just six pins (the analog pins A0 through A5). If more ports were available, a good option to add would be a frequency readout. This could be achieved with seven more pins (or eight if using the decimal point) driving the segments of the LED displays plus three or four more pins for the multiplexing (depending on the desired accuracy of the display and frequency range). Some of the other boards in the Arduino family, such as the Mega, would be able to handle this.

The technique used here to change frequency is referred to as "variable sample rate". It is relatively simple but has certain drawbacks, not the least of which is difficulty in filtering the output (the process of removing the wave form "stair steps" which is something we ignored). An alternate technique uses a constant, high sampling rate. The frequency is altered by generating the desired wave via interpolation in a larger table. The computational load is a little higher but even simple linear interpolation can achieve very high quality results without that much extra processing time. Finally, this sort of application can benefit from the use of timed *interrupts*. These can be thought of as little snippets of code that run at tightly defined intervals, in this case controlled by on-board timer/counters. The timer/counter would be programmed to trigger a software interrupt at regular intervals thus removing the need for the `delayMicroseconds()` function. The interrupt service routine would do little more than copy the next value in the wavetable to the output port. This technique would create very stable timing for the waveform and free up the `loop()` function to just monitor the input switches and such.

15

Arduino Interruptus
or Revenge of the Son of the Reaction Timer Redux Revisited

The concept of interrupts was presented in the Event Counter exercise as a means of responding to external events in an accurate and timely manner. Interrupts can be used in other ways. One good example would be to maintain a multiplexed display, like the ones used in the Reaction Timer and Event Counter exercises. As presented originally, the display code is a bit problematic in that it runs in the main loop and wastes a lot of processor cycles through the use of `delay()` calls. The other code is forced to "fit around" that timing. A better scheme would be to use software interrupts to generate the timing for the display digit scan. A timer would be programmed to trigger an interrupt every 10 milliseconds or so, corresponding to the individual scan time for each display digit. When triggered, the ISR would extinguish the display and then enable the next digit, working in a round-robin fashion of units-digit, tens-digit, hundreds-digit, units-digit, tens-digit, and so on. Nearly all of the original display code could be used to create the ISR. The ISR would just need one additional variable to keep track of the currently illuminated digit. Also, the timer/counter would have to be set up to achieve a proper trigger rate.

If we use a 1024 prescale with a 16 MHz clock, timing ticks come along at about 16 kHz. Using the Normal mode, a full 0 to 255 count would take about 16 milliseconds. With three digits, this requires nearly 50 milliseconds for one scan and will likely produce noticeable scanning artifacts on the display (i.e., a "beating" or "throbbing" effect). Unfortunately, the Uno's 328p does not have a 512 prescale (the 256 being perhaps a bit too fast). The solution to this is to shorten the time by preloading the counter's register so that it doesn't have as far to count (e.g., instead of counting from 0 to 255, we could count from 150 to 255). This is a nice technique to remember when we need to obtain specific timing values. In this case, we can adjust the start value "by eye" without resorting to precise calculations.

The code presented below is built on the original Event Counter code and assumes identical hardware. First, we need to add the initialization for timer/counter number 2 in the `setup()` function:

```
// Set up timer-counter for muxed 7 seg displays
TCCR2A = 0;               //  normal mode, OC2x pin disconnected
TCCR2B = 0x07;            // 1024x prescale tics at ~16kHz
TCNT2 = OVF_COUNT_START;  // init counter: count up from here to 256
TIMSK2 = (1<<TOIE2);      // enable overflow interrupt
```

We also introduce a constant that will allow us to adjust the scan time, with larger values producing faster scan times.

```
#define OVF_COUNT_START 150
```

We now lift the display code from the old `loop()` function and use it to form the ISR for the overflow interrupt. We need to create a global variable to keep track of the currently active digit and we also need to remove any reference to the `delay()` function as it is no longer being used.

```
char g_current_digit=0; // 0 for units, 1 for tens, 2 for hundreds

ISR(TIMER2_OVF_vect)
{
   unsigned int v;
   unsigned char h, t, u; // hundreds, tens, units

   // Break the global count variable into units, tens and hundreds digits
   // Grab a local in case another interrupt comes by
   v = g_count;

   if( v > 999 ) // error code
   {
      h = LETTER_E;
      t = u = LETTER_R;
   }
   else
   {
      u = v%10;
      v = v/10;
      t = v%10;
      h = v/10;

      // blank leading zeros
      if( !h )
      {
          h = LETTER_BLANK;
          if( !t )
              t = LETTER_BLANK;
      }
   }

   // clear all displays then activate the desired digit
   PORTB |= DIGIT111;

   if(!g_current_digit) // use units digit
   {
      PORTB &= ~DIGIT1;
      PORTD = numeral[u];
   }
   else
   {
      if(g_current_digit==1) // use tens digit
      {
         PORTB &= ~DIGIT10;
         PORTD = numeral[t];
      }
      else        // must be hundreds digit
      {
         PORTB &= ~DIGIT100;
         PORTD = numeral[h];
      }
```

```
  }
  g_current_digit++;      // prep for next digit on next iteration
  g_current_digit %= 3;  // wrap back to units

  TCNT2 = OVF_COUNT_START;  // preload counter: count up from here to 255.
}
```

Comparison of the code above with the original shows that very little modification was required. Now that the display updating has been removed from the main event loop, there is nothing to do there:

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

The obvious question at this juncture is, "What's the point of having an empty loop function?" By itself, not much, but in a situation where other input and/or output devices, sensors and the like need to be monitored or set, the new loop() function can be designed with just that utility in mind, and without the worry of inserting specific bits of timing code into it. This makes for a much cleaner and easier to maintain system. Essentially, you have "handed off" the maintenance of the display to an automatic background process.

In the testing of this new version, it is worthwhile to first compare it to the original version to see if the performance is identical. Second, try several different values of OVF_COUNT_START in order to see the effect of display flicker.