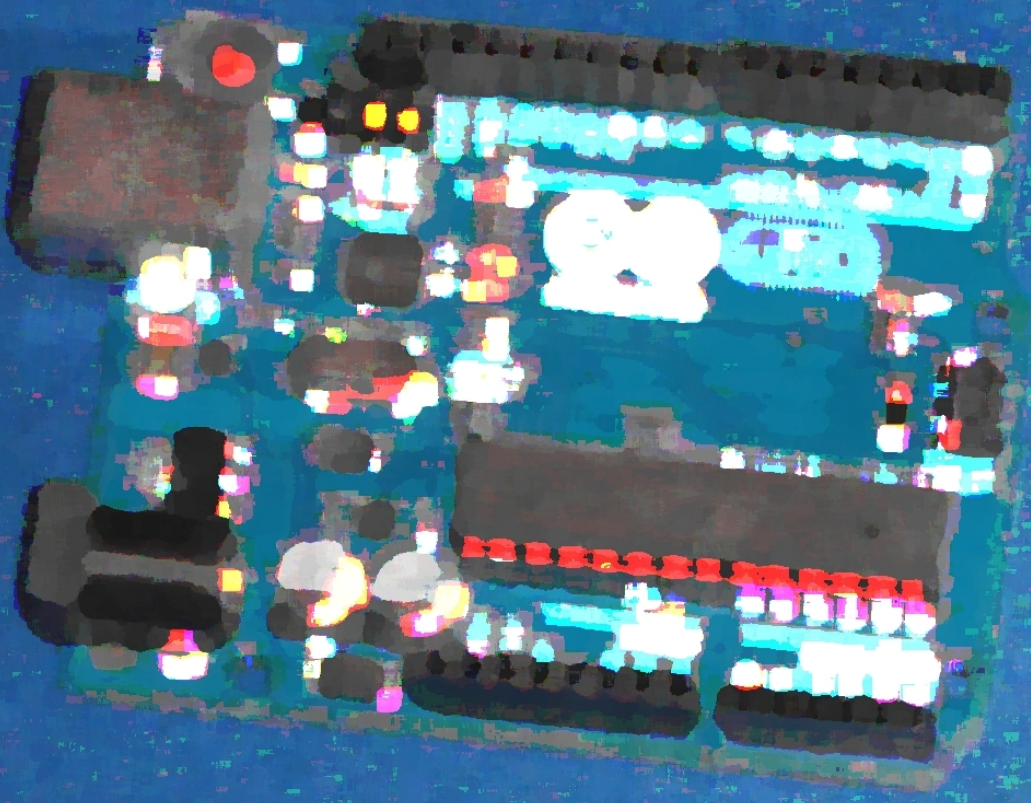
Embedded Controllers

Using C and Arduino



James M. Fiore

Lecture Notes

for

Embedded Controllers

Using C and Arduino

by

James M. Fiore

This **Lecture Notes for Embedded Controllers Using C and Arduino, by James M. Fiore** is copyrighted under the terms of a Creative Commons license:

This work is freely redistributable for non-commercial use, share-alike with attribution

Published by James M. Fiore via dissidents

For more information or feedback, contact:

James Fiore, Professor
STEM Center
Mohawk Valley Community College
1101 Sherman Drive
Utica, NY 13501
jfiore@mvcc.edu

or via www.dissidents.com

Cover art by the author

Introduction

These lecture notes are designed to supplement and expand upon material related to the C programming language and embedded controllers, and specifically, the Arduino development system and associated Atmel ATmega 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 that 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, these notes do 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 notes 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.

There is a companion lab manual to accompany this set of lecture notes. Other lab manuals in this series include DC and AC Electrical Circuits, Computer Programming with Python, and Linear Electronics.

A Note from the Author

These lecture notes are 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

1. Course Introduction 8
2. Hardware Architecture 10
3. C Language Basics 14
4. C Language Basics II 22
5. C Storage Types and Scope . . . 30
6. C Arrays and Strings 34
7. C Conditionals and Looping . . . 38
8. C Pointers 46
9. C Look-Up Tables 50
10. C Structures 55
11. C Linked Lists* 58
12. C Memory* 62
13. C File I/O* 66
14. C Command Line Arguments* . . . 70
15. Embedded Programming 72
16. AVR ATmega 328p Overview . . . 76
17. Bits & Pieces: includes and defines . . 82
18. Bits & Pieces: pinMode 90
19. Bits & Pieces: digitalWrite 96
20. Bits & Pieces: delay 100
21. Bits & Pieces: digitalRead 108
22. Bits & Pieces: analogRead 112
23. Bits & Pieces: analogWrite 118

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

1. Course Introduction

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 (entirely optional, but programming these little beasties can be addicting so be fore warned). Along with the main course text (Intro to Embedded Programming by Russell), 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 'net 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, not C++. You will also need a desktop C compiler. Just about any will do, including Visual C/C++, Borland, Code Warrior, or even GCC. A couple of decent freeware compilers available on the 'net include Pelles C and Miracle C.

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 (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. For more complicated tasks C tends to cut to the heart of the matter. Many kinds of data manipulation are much easier and more 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. Hardware Architecture

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.

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.

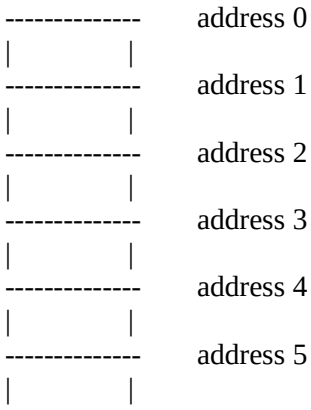
Processors are often connected to external memory (RAM chips). Microcontrollers generally contain sufficient on-board memory to alleviate this requirement, but it is worthwhile to note that we are not talking about large (megabyte) quantities. A microcontroller may only contain a few hundred bytes of 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 variable, the value of the variable must first be copied into a register. The addition is then 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 takes care of most of those details for you.

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 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:



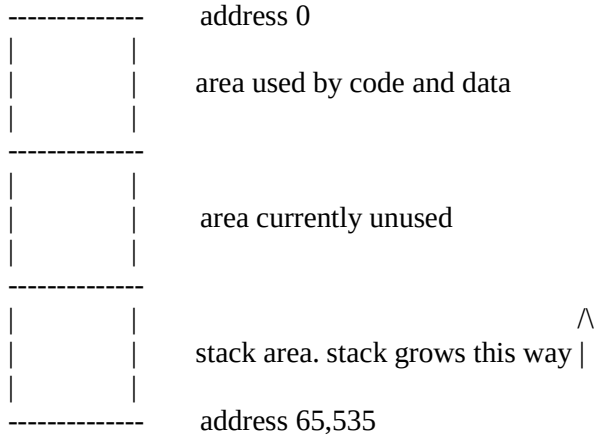
Each address or slot represents a place we can store one byte. If we had to remember specific addresses we would be doing a lot of work. Instead, the C compiler will keep track of this for us. For example, if we declare a `char` named `X`, it might be at address 2. If we need to print that value, we 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`.

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.

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 try 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.



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

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.

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. `char`s and `int`s 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

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;            This declares a signed 8 bit integer called x
unsigned char y;   This declares an unsigned 8 bit integer called y
short z, a;        This declares two signed 16 bit integers named z and a
float b =1.0;      This 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 ;
```

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 )
{
      …statements…
}
```

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 required values nor returned 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. OK, so here's our first program:

```
/* Our first program */

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

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

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);}
```

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 its own function. Our program now looks like:

[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 );
}
```

The new math function takes two `float`s 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 `float`s 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).

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. 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");
}
```

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.

OK, so if we want to print out the answer to the first program, we'd wind up with something like this:

```
/* 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);
}
```

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
```

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.

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.

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.

4. C Basics II

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

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" 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 `raw_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!

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[3]. 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.

[3] In some systems, 80 bit doubles and 64 bit integers are also available.

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.

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

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;
```

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 0^{th} bit to input mode, you could write[4]:

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

If sometime later you wanted to also set the 1^{st} and 2^{nd} 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;
```

or using the more common shorthand:

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

Note that the code above 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*.

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

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`.

#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 really just 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.

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
```

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!

5. C Storage Types and Scope

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 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).

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.

6. C Arrays and Strings

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 with 15 chars each, or 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.[5] 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

[5] 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.

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)

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

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).

7. C Conditionals and Looping

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

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):

||	OR
&&	AND
!	NOT

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 code fragment below:

```
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.

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.. */
}
```

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.

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.

8. C Pointers and Addresses

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.

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 32 bits (4 bytes) although there are 64 bit systems and 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).

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 few 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 `a`, which was assigned the value `1` at the start. That is, `b` gets the value at the address `pc` points to, which is `a`.

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 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. `double`s 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.

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 straight forward, 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!

9. C Look-up Tables

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 you can trade performance in one area for another. For example, a certain technique might be very memory efficient but rather slow, or vice versa. We're going to take a look at a common programming technique that is very fast. Sometimes it can require a lot of memory, sometimes not. It's called a look-up table.

"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 multiples 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)[6]. 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, `float`s might have sufficient precision for the application which will cut the memory requirements in half again when compared to `double`s.

[6] 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.

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. This is possible but can be tricky to do with analog oscillator techniques coupled with waveshaping circuits. 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;

while ( 1 )  // loop forever
{
      PORT = ramp_table[i++];
      delay();    // wait between each value, dependent on sample rate
}
```

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 0ºC 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

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");
```

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 `double`s 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 */
```

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[7] 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.

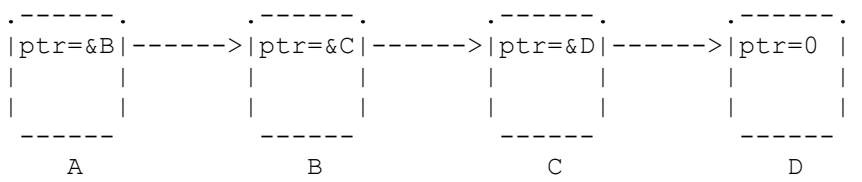
[7] i.e., try to access the fields of.

11. C Linked Lists

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:



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

[8] 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`.

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. It would be called like so:

```
PrintMarmotAges( MarmotList );

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 );
      }
}
```

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).

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

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.

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!

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 ... */
```

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 );
}
```

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.

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).

13. C File IO

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.

```
/*                   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

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 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.

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.

As an exercise, add `DEBUG printf()` statements to any of your existing programs.

15. Embedded Programming

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?

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 like 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*. 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 max 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.

Math

Usually, embedded code is not math intensive. There are some exceptions to this rule, but generally VCR code 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.

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 power of the processors tend 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:

```
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.

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. AVR ATmega 328p Overview

Or "Controlablanca: A Film Noir Microcontroller"[9]

"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 an MIT grad, maybe Stanford. 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, some 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.

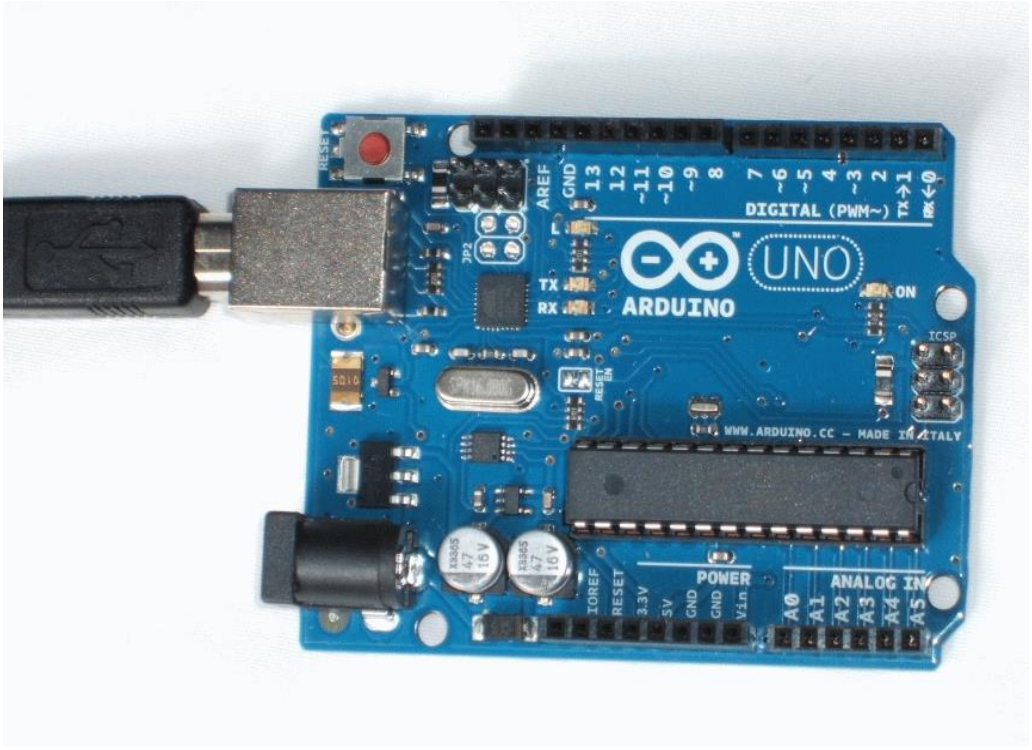
"Arduino, this is Missy", said Rick. "Che?" came the young man's response. "Not Kay, C", said the dame. Arduino looked at her and asked, "Not Casey, but who?" She sensed a possible language barrier and tried to meet him halfway. "Miss C, si?" she said. "Missy C", nodded Arduino. "No", she replied, her 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

[9] 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.

these ones, must be in their DNA. Why, if I was a bug she'd be a regular *Venus De Flytrap* – beautiful *and* 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."

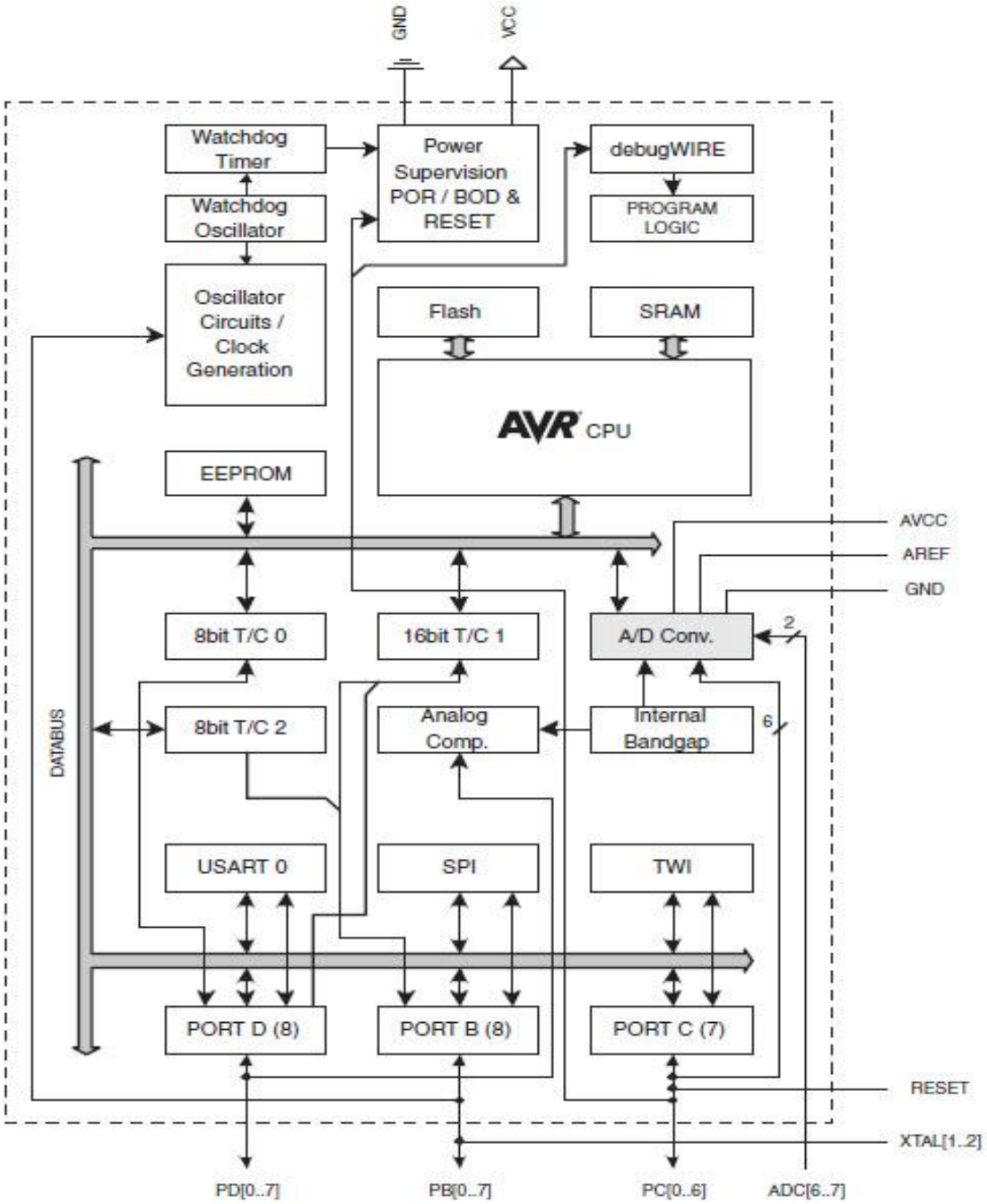
Arduino walked across the office and nibbled at some biscotti. He turned back to her, the twice-baked biscuit still at his fingertips, held softly the way an orchestra conductor might balance a baton. "This application…" he said, his head slightly raised and cocked to one side revealing the beginning 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.

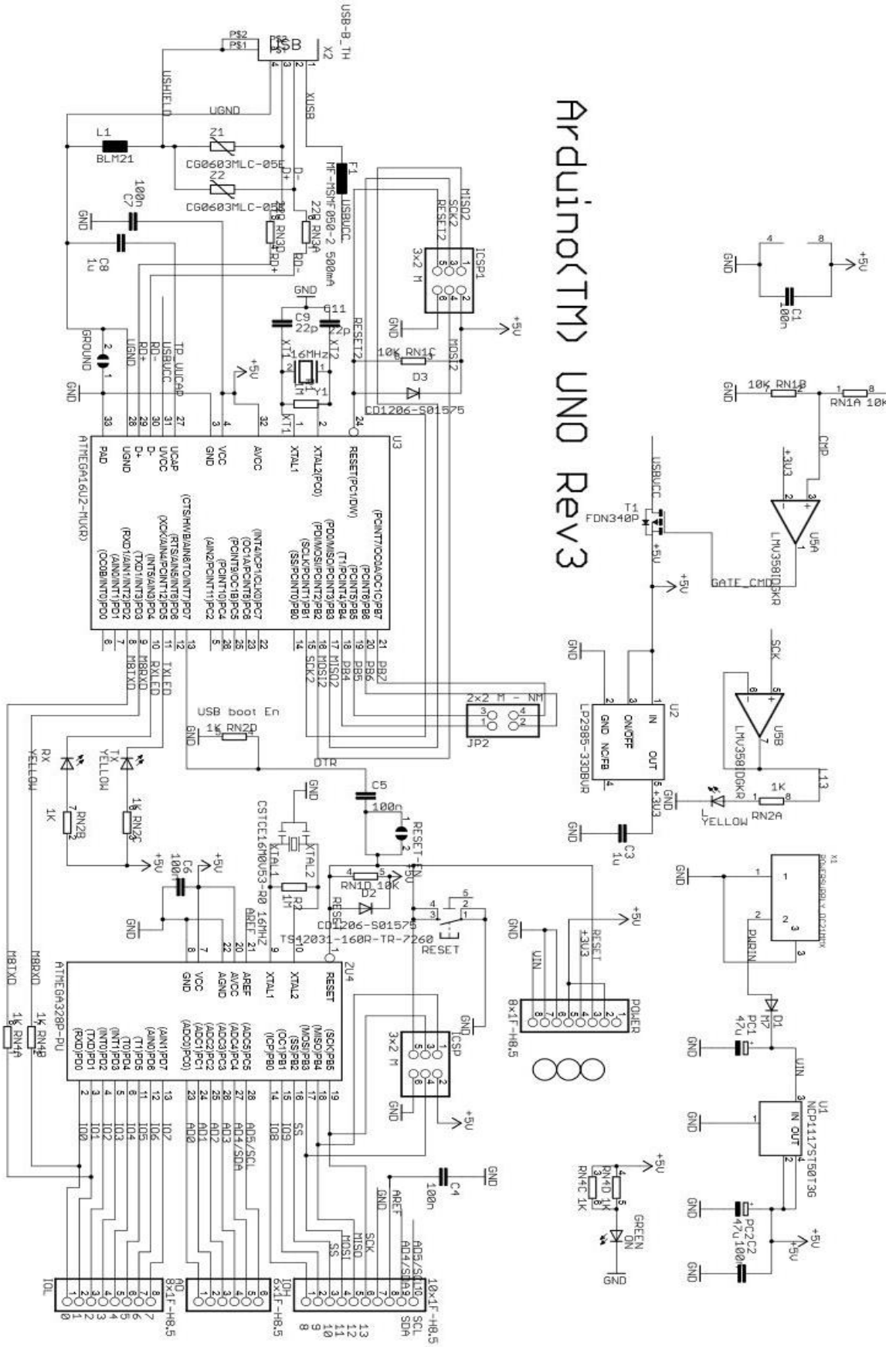


"Pretty spiffy", she said. "What is it?" Arduino's eyes lit up. "It's an open source microcontroller development board, and when I say 'open source' I mean both software *and* hardware. The software distribution includes extensive examples, complete library source code, and all the data you could want 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
USB-B_TH
BLM21
CG0603MLC-05E
CG0603MLC-05E
MF-MSMF050-2 500mA
ICSP1
3x2 M
ICSP
3x2 M
JP2
2x2 M - NM
USB boot En
DTR
RESET-EN
RESET
TS42031-160R-TR-7260
CD1206-S01575
CD1206-S01575
16MHz
CSTCE16M0V53-R0 16MHZ
FDN340P
LP2985-33DBVR
LMV358IDGKR
LMV358IDGKR
GATE_CMD
CMP
YELLOW
GREEN
ON
ATMEGA16U2-MUXR
ATMEGA328P-PU
MBTXD
MBRXD
RXLED
TXLED
RX
TX
YELLOW
YELLOW
USB
SCK
PWRIN
VIN
RESET
+3V3
POWER
8x1F-H8.5
10x1F-H8.5
NCP1117ST50T3G
GND
+5V

"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."[10]

[10] 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.

17. Bits & Pieces:

#includes and #defines

Welcome to Bits & Pieces (catchy name, eh?). This sequence of notes 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[11]. 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).

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"
```

[11] 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 HIGH  0x1
#define LOW   0x0
#define true  0x1
#define false 0x0

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

#define PI          3.14159265358979323846264338327795
#define HALF_PI     1.57079632679489661923132169163975
#define TWO_PI      6.28318530717958647692528676655901
#define DEG_TO_RAD  0.01745329251994329576923690768488
#define RAD_TO_DEG 57.29577951308232087679815481410517
```

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 noInterrupts() 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.

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 `#define`s 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[12].

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…

[12] 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.

18. Bits & Pieces:

pinMode()
or "Programming Port Directions With 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 0^{th} 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[13]:

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

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 pin designations versus ATmega 328p port and pin naming.

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

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

Table 1, Arduino Pin Definitions

Note that theA0 through A5 designators are for the analog inputs and the remaining are for digital IO. The naming convention is reasonably logical but less than perfect. It should also be noted that the analog channels are input-only. The controller cannot produce simple continuously variable analog voltages on its own. This is not to say that'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 1, below.

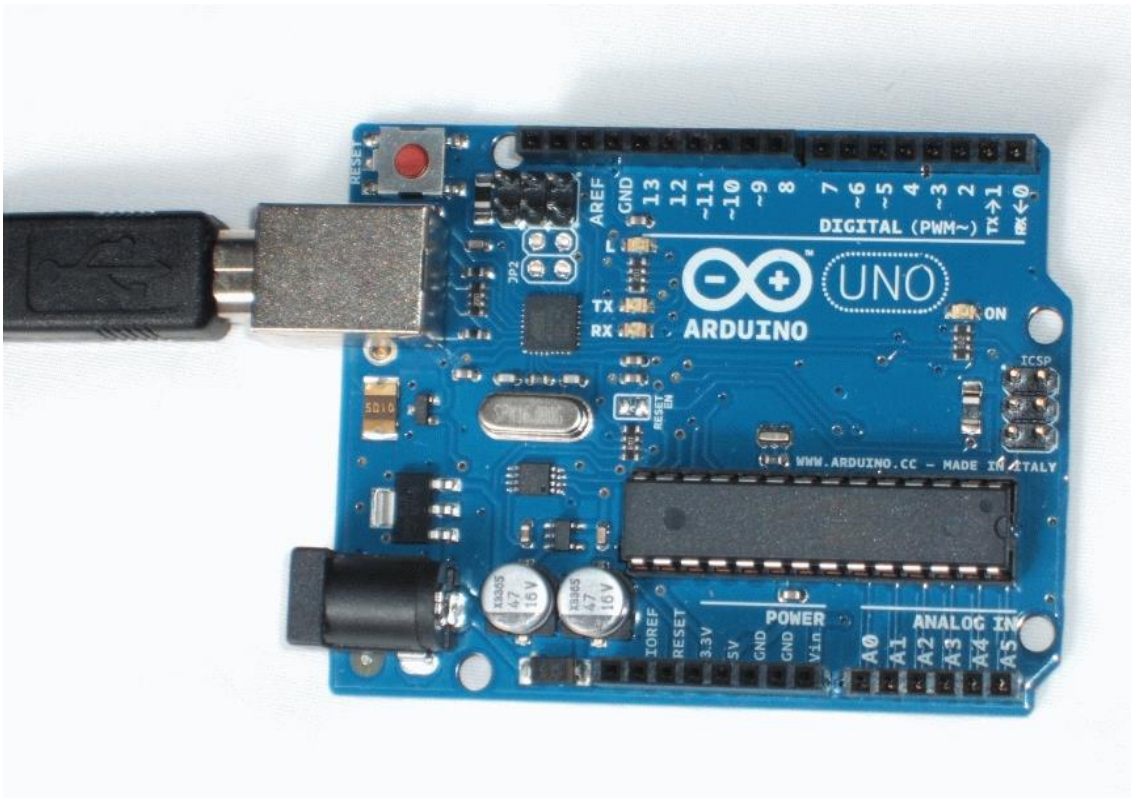


Figure 1, Arduino Uno

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 );
```

```
        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;
                }
        }
}
```

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 four of these, `DDRA` 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[14] filled with the pre-defined error symbol and the addresses of the appropriate data direction registers:

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

```
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.

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[15] 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.

[15] 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.

19. 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[16] 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 straight forward 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.

[16] 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

none

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
}
```

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[17]:

[17] 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;
}
```

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.

20. 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()[18]

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

[18] 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.

(end of copied reference material)

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

micros()

Description

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*)

(end of copied reference material)

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;
}
```

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++;
}
```

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());
}
```

So the `delay()` function itself is pretty straight forward. 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;
		}
	}
}
```

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
      );
}
```

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.

21. Bits & Pieces:

digitalRead()

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()[19]

Description

Reads the value from a specified digital pin, either HIGH or LOW.

Syntax

digitalRead(pin)

Parameters

pin: the number of the digital pin you want to read (*int*)

Returns

HIGH or LOW

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
}
```

[19] 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

(end of copied reference material)

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
      }
}
```

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 resister. 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
```

22. Bits & Pieces:

analogRead()

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()[20]

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, .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.).

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

See also

- analogReference()
- Tutorial: Analog Input Pins

analogReference()[21]

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, 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.

(end of copied reference material)

[21] http://arduino.cc/en/Reference/AnalogReference

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). First off, the ADC system includes a few important registers including `ADCSRA` (ADC control and status register A), `ADMUX` (ADC multiplexer selection register), and `ADCH` and `ADCL` (the high and low bytes of the result, respectively). These registers include the following bits:

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

For `ADCSRA`, set `ADEN` to enable the ADC. Setting `ADSC` starts a conversion. It will stay at one until the conversion is complete. `ADIF` and `ADIE` (`AD I`nterrupt `F`lag and `I`nterrupt `E`nable) are used with an interrupt-based mode not discussed here (`ADIF` is set to one by the controller when a conversion is complete). `ADATE` stands for `AD A`uto `T`rigger `E`nable which allows triggering of the ADC from an external signal (again, not discussed here). `ADPSx` are pre-scaler bits which set the conversion speed. Basically, they set divisions of the system clock. 000 = no divide, 001 = /2, 010 = /4, 011 = /8, up to 111 which divides by 128.

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 `#ifdef`s 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 and left-justify result
      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;
}
```

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 S`tart `C`onversion 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` is 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 pre-scaler 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 straight forward to use as is. Use this function as a guide if you wish to produce left-justified data 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.

23. 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()[22]

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.

[22] http://arduino.cc/en/Reference/AnalogWrite

You do not need to call pinMode() to set the pin as an output before calling 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

(end of copied reference material)

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);
                }
          }
    }
}
```

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 which means that 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).