Controlling The Real World With Computers

Putting It All Together - Controlling The Hardware With The Software

Much of the world's software is written in the C programming language. A significant improvement, the C++ language, is now finding some use in embedded systems and is extensively used in factory control systems. This site's tutorial sections concentrate on the basics of the C language.

C language source code is written in plain text format. C programs have a .c extension. C++ program files have a .cpp extension. Only basic C will be discussed here. The many benefits of C++ can be found in the tutorials offered in the software section of the technology education sites links.

Two programs are used to turn the source code into a program the computer can use. One is called a compiler. It produces what is called object code. Then a program called a linker builds the final executable program from one or more object code files. The linker is often left out in discussions (including here) but they are both needed. See Webopedia for a good description of the process.

The compiler I used in the examples is made by MIX Software. It's only $20 and comes with an excellent manual that includes a tutorial. It has been adopted by many colleges and training companies teaching C programming classes. Any compiler that complies with ANSI standards and has the appropriate PC extensions should do just fine. Even some of the free ones will work.

Text written for the native processor is called assembly language. An assembler makes object code out of assembly text. MIX's assembler is only $10 and includes their source code!

The following sample code and explanation will provide the information you need to start controlling the board using the C programming language. Text explaining what's going on will be in normal black and white, whereas the actual example code, important words and phrases and points needing emphasis will be in **c o l o r**.

It's very important to comment your code as you go along. It's tempting to put it off thinking all will be remembered. Murphy says it won't. There are two kinds of comments. One is for a single line and uses the double slash:

```
// double slash, single line comment
```

The other type of comment is used for either a single line or multiple lines. The comment section is started with /* and ended with */:

```
/* This is the type of
comment that can be used with
more than one line. */

/*
This is also a multi-line
comment.
*/
```

Some compilers recognize only the /* */ sequence. If you get errors related to comments and you are using the double-slash type, try changing to the /* */ sequence to see if that will fix the problem.

All of the action in a C program starts with something called **main()**, which looks something like this:

```
main()
{
}
```

Every C program **must** have a main(). In fact, that's all you need. If you have only the above in a program it will compile and run. It just won't do anything that can be seen or heard. Only a computer will love it.

Main() is the starting **function** of a C program. A function is a piece of code that performs a task. A C program is basically a collection of functions. Functions are **called** by other functions, which means a service is being requested which is provided by the called function. Functions can **return** something to the caller, perform a process or both. Functions can also accept **arguments** inside the parentheses following the function name. Arguments can be just about any type of data needed by the function. Multiple arguments are separated by commas. A more proper way to show main() if it does not accept arguments or return anything is to use the following form:

```
void main(void)
{
}
```

The **void** terms mean that main returns nothing and does not accept arguments. Everything inside the curley braces, **{ }**, belongs to the main() function. That's true of any function. **The code belonging to a function is placed inside curley braces.**

Main() would look like the following if it returned an integer. You need to see Data lines, bits, nibbles, bytes, words,binary and HEX if you don't know about the **int** keyword:

```
int main(void)
{
}
```

Now let's say we are going to call another function from inside main(). We will make it return an integer that is what we send it squared. Before we use it however, we must describe it. This is accomplished with what is called a **prototype**. Our prototype for the squaring function would look something like this:

```
int squared(int x);
```

This says that the function we have named "squared" will return an integer. It also says that it is to be sent a single integer as an argument. The **int x** inside the parenthesis could just as well have been **int xyz**, because here we are just describing the function, not actually sending it a variable with a specific name. Speaking of names, C has some restrictions in the use of names and labels. From C Programming by Steve Holmes of The University of Strathclyde Computer Centre , "Every variable name in C must start with a letter, the rest of the name can consist of letters, numbers and underscore characters. C recognizes upper and lower case characters as being different. Finally, you cannot use any of C's keywords like main, while, switch etc as variable names." This applies to any label, including function names. The use of a blank space in a label name is also not permitted. The label "this is a name" is not allowed, but "this_is_a_name" and ThisIsAName are OK.

```
void main(void)
{
  int x,y;            // declare the integers x and y
                      // notice the semicolon
                      // all C statements or groups of
                      // statements end with a semicolon

  x = 5;              // set x equal to 5
```

```
  y = squared(x);    // set y equal to whatever squared returns
                     // squared is sent a copy of x, not x itself,
                     // so x is not modified by squared
}


/* ================================================================
   The following begins "squared." Notice that there is no colon after
   the parenthesis as there was in the prototype, and that the argument
   name is different.  We have given the input argument the name "indat."
   We could have called it anything as long as we declared it an
   integer the same as the prototype.  Also notice the way this comment
   is set off.  Use any format you wish.  Just be sure to declare a comment
   with // or the /* */ sequence.  Most importantly, be sure to describe
   a subroutine with a comment section above it.
  ================================================================ */
int squared(int indat)
{
  return indat * indat; // return the input int times itself -- square it
}
```

The problem with the above is that it's not apparent what's going on. To see, let's print the results of several calls to squared.

But first, let's look at another feature of C. Very often you will find what are called compiler directives near the top of a C source code file. They are prefixed with **#** and tell the compiler to perform certain tasks. One of the most common is the directive to include other files. When the compiler encounters an **#include** statement, it suspends compiling the current file, compiles the included file then returns to compiling the current file. Most of the time the included files are **header** files, with the **.h** extension. The file name is usually bracketed with <> if the information in the header deals with material belonging to the compiler. The name is usually set off in quotation marks if it belongs to the programmer.

The sample program is below:

```
// exmpl1.c -- it's a good idea to have a comment at the top of a
// C source file to show its name so you will know what you are
// looking at if you print it

#include <stdio.h> // contains the printf prototype and others
                        // that deal with Input and Output

int squared(int x); // prototype for squared

void main(void)
{
  int x,y;

  for(x=0; x<11; x++)  // set x equal to 0, then 1, etc. through 10
  {
    y = squared(x);    // set y equal to whatever squared returns
    printf("%02d squared = %03d\n",x,y);  // print the results
```

```
  }

}


/* ================================================================
   squared(int indat) returns indat times itself
  ================================================================ */
int squared(int indat)
{
  return indat * indat; // return the input int times itself -- square it
}

// end exmpl1.c -- it's a good idea to show where the end of the file
// should be -- if it is missing, you know the file got corrupted
```

The output of the above program is below:

```
00 squared = 000
01 squared = 001
02 squared = 004
03 squared = 009
04 squared = 016
05 squared = 025
06 squared = 036
07 squared = 049
08 squared = 064
09 squared = 081
10 squared = 100
```

Let's see what the program did. The first addition in this version is what is called a for loop. The following shows how it works:

```
      _Start x at 0
     |      _Run the loop as long as x
     |     | is less than (<) 11
     |     |       _Increment x by 1 each time
     |     |      | through the loop
     |     |      |
     |     |      |
for(x=0; x<11; x++)
```

What this says is to first set x to 0, then run the loop as long as x is less than 11, incrementing it each time around. Everything inside the curley braces ({ }) under the for() loop statement is part of the loop:

```
for(x=0; x<11; x++)
```

```
{
  // everything in here is part of the for() loop
}
```

For loops are very flexible. Consider the following:

```
for(x = 12; x >= -20; x-=2)
{
}
```

Notice the spaces between x, = and 12, as well as others. C does not care if you add **white space**, which is anything that is not readable, such as tabs, spaces, blank lines, etc. C ignores white space. It is a very good idea to use that fact to spread things out a little if it makes the code easier to read. Here, x is started at 12. The loop continues as long as x is greater than or equal to (**>=**) -20. Each time around, x is decrement by 2 (**-=2**). Thus, x would follow the sequence 12, 10, 8 etc., 2, 0, -2 etc., -14, -16, and finally reach -20.

Back to the original program. The first thing that happens inside the for loop is a call to the squared() function. The x variable is used as the value for the argument to be squared, and y gets the result from the squared() function:

```
y = squared(x); // set y equal to whatever squared returns
```

After y is set by the return of squared, both x and y are printed using the **printf()** function. Printf is one of the built-in C functions and means print formatted. The formatting comes from what you put in the quotation marks:

```
printf("%02d squared = %03d\n",x,y); // print the results
```

Notice the percent signs. They introduce formatting instructions for the printf() function. A **%d** would tell printf() to print a signed integer in **d**ecimal format. A **%u** would tell printf() to print an integer in **u**nsigned format. A **%x** would tell printf() to print an integer in he**x**adecimal format using lower case letters. To print upper case letters, use **%X**. The **%02d** instruction says, "print a signed integer in decimal format using 2 character positions and provide leading zeros." You can designate leading zeros and/or the number of digits with virtually any of the formats in a similar manner. After the last quotation mark is a list of the things to print: **",x,y);**. The first print format instruction after a percent sign refers to the first item in the list and the second one refers to the second item in the list. There can be many format instructions and many items in the list. The second item in the list is the y. The format instruction for it is slightly different. The **%03d** says, "print a signed integer in decimal format using 3 character positions and provide leading zeros." Two percent signs together, **%%**, mean to print a percent sign. The **\n** is what is called an escape sequence. Escape sequences are introduced by the back slash. The **\n** simply means to go to a **n**ew line after printing this one. A **\t** means **t**ab to the right and a **** means to print a back slash. All characters inside the quotation marks for printf() that are not format instructions or escape sequences are printed literally as you see them. The blanks, the word "squared" and the equal sign are all literals. The following shows how the call to the printf() function is made for what we want to print here:

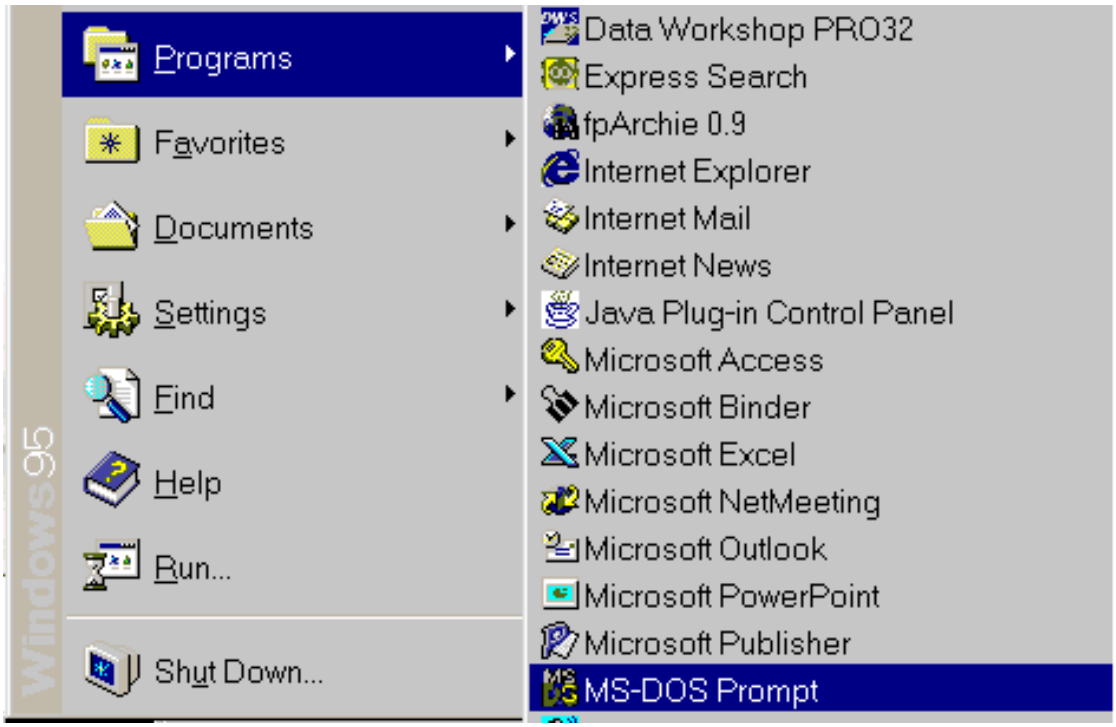
```
start with a parenthesis and a quotation mark
|  Then tell printf to print the following:
|  the signed decimal value of x as 2 characters with leading zeros
|  |  blank, "squared", blank, an equal sign then blank
|  |  |           3 character signed decimal value of y with leading zeros
|  |  |           |  and finally a new line
|  |  |           |  | put a quotation mark just before the variable list
|  |  |           |  | | x is the first item in the list
|  |  |           |  | | | y is the second item in the list
|  |  |           |  | | | |end with parenthesis and semicolon
|  |  |           |  | | | ||
```

```
      |   |   |             |   | | | ||
      |   |   |             |   | | | ||
printf("%02d squared = %03d\n",x,y);
```

Please see your compiler manual or the printf() section of The Free On-Line Dictionary of Computing for more information on printf().

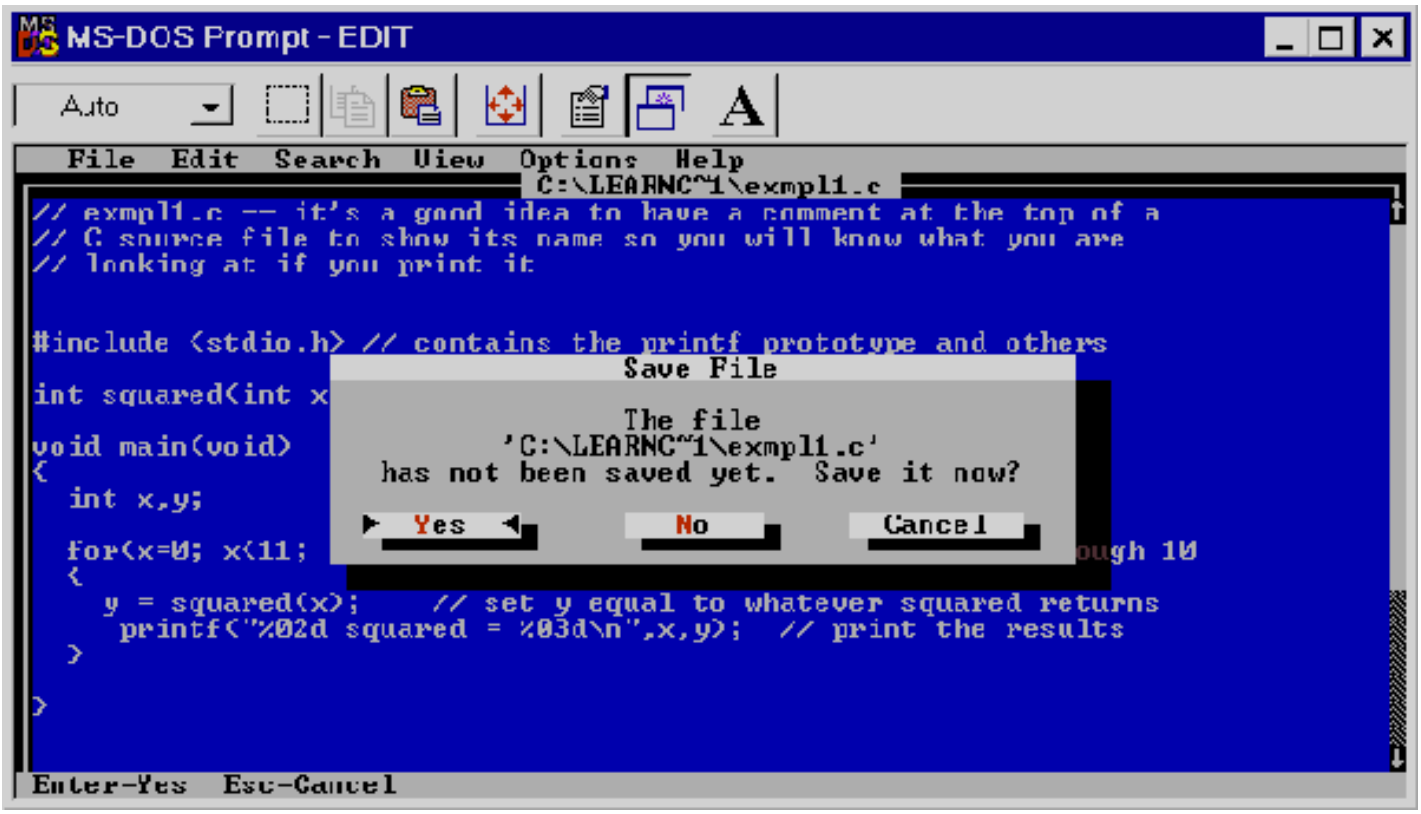
It would be a good practice exercise to type the above program into a text file with a .c extension, compile it and run it. The best way to learn any language, be it human or machine, is to use it. For this exercise, it would probably be a very good idea to print this page, especially if you are using a computer other than this one for the experiments. Even if you aren't, it's a hassle to bounce back and forth between windows and DOS.

If you are using Windows 95, first get into DOS by going to Start/Programs/MS-DOS Prompt:



Now type in cd\ then press enter. This means **c**hange **d**irectories and go to the **root**, which is actually outside any directory (a directory is the same as a folder). Now type in md LearnC to **m**ake a **d**irectory called LearnC, then press enter. Finally, type in cd LearnC to **c**hange **d**irectories to get into the LearnC directory.

Now type in edit exmpl1.c then press enter. Type in the above program text then press ALT-F then X to leave. The edit program will ask if you want to save the file. Click on yes:



Install your compiler if you have not already done so. Pay close attention to the path requirements and how to set your computer up for the proper defaults.

To compile a program using the MIX PowerC compiler, type in the following then press the enter key:
\powerc\pc /e /n /ml progname <enter>
Where progname is the name of the C source file without the .c extension. In this case, you would type:
pc /e /n /ml expl1 <enter>.

This translates to PowerC Compile (pc), linking to make an executable (/e), allowing nested comments (comments within comments, /n) and use the large memory model (/ml) for unlimited code and data space.

Download do.bat
do.bat is a DOS batch file that contains the above compiler commands to permit simply typing in do progname, for example, then pressing the enter key. It can be used with any of the experiments that use only one file. The name should be no more than 8 characters not counting the period and extension (12345678.c).

To run the program, simply type in expl1 <enter>. You should get the same output as above.

If you are using Windows, be sure to type cd\learnc <enter> each time you get out of Windows and go into DOS so you will be in the correct directory.

Previous: Hardware *Next:* Experiment 1 - Basic Switch Input Detection