1

Introduction to C Programming

The main objective of this initial programming exercise is to become familiar with using the programming language tools. The programs in this exercise will be fairly trivial, but serve as a springboard to later work. We will be using programs similar to the ones examined in lecture.

The precise C language package in use is not of extreme importance. Various companies will create different programming tools and although the features and fine points may differ, the basics remain the same. All C language tools need to compile C code and assemble it. Further, they need to link this assembled code with other assembled modules and libraries in order to create a finished executable program. In the simplest case, these tools will be in the form of command line utilities, i.e.; they run from a DOS prompt or shell. Ordinarily, the tools are part of a graphical integrated development environment, or IDE. IDE’s normally include a text editor. C compilers expect to work on raw text. Do not attempt to “feed” them the output of a word processor, such as .doc file. If you are using simple command line tools instead of an IDE, you will create your source files with a basic text editor such as Notepad.

C source code files utilize a “.c” extension. The output of the compiler is called an *object file*. It will normally have a “.o” or “.obj” extension. In the Windows world, finished executables usually have a “.exe” extension. Many IDE’s require that you create a *project* before you start entering your code. The project includes many attributes such as the file names used for source code (there may be several in larger projects), the appropriate libraries to link, the name of the finished executable, and so on. For simple programming chores involving small amounts of source code, this can be a bit of a pain, however, it is wonderful for larger endeavors. All the C source code of all of our exercises during this course can easily fit on a single very modestly-sized (64 MB!) USB drive. This includes the project files that can become much larger (many megabytes) than the C source. In networked labs on campus, project files and source files can be saved to the student’s network storage area. For those performing labs off-campus, it will probably be easiest to simply create new projects on your hard drive as needed.

This lab will use the **onlinegdb.com** online compiler application. There is nothing magical about **onlinegdb.com** though and other systems are perfectly acceptable. We shall only be using **onlinegdb.com** online compiler for the introductory exercises anyway. Once we get rolling we shall shift our emphasis to the Arduino development board.

Our first exercise focuses on creating a project, editing source code, compiling and linking it, and testing it. We shall then edit it and repeat the process. We shall also look at error reporting. If you’re using command line utilities, see the note at the end of this exercise before continuing.

To begin, open **onlinegdb.com** online compiler application. Sign up with your name, email and password. **Onlinegdb.com** will send you a verification link to your email and verify it by opening your email and click a verification link given to you. Then open **onlinegdb.com** online compiler, a source code black text edit window editor will appear.

Type the following code into the editor:

**Listing 1**

#include <stdio.h>

/\* It all begins here \*/

int main()

{

printf("Hello World ");

return 0;

}

Save this as hello.c. using *Save* button at the top. A dialog box will pop up asking if you want to add this file to the current project. Select “Yes”. Note that in some IDEs you will have to manually insert the file into the project (look for the appropriate menu item in such a case).

While it is possible to separately compile and link modules, most developers use the *Build* shortcut. This will compile and link all files as needed. In **onlinegdb.com** you can select *Create New Project* from from the toolbar. As the project is built, you will notice messages in the status area at the bottom.You can now check the program by choosing on *My Projects*. Select the *Run* toolbar from the menu bar. Choose a C Language on the list bar on the top right of the window. A small DOS shell should pop open with the message “Hello World!”. Press enter to clear this shell window and return to the IDE. You have successfully completed your first program!

Depending on the settings for your IDE, you may notice different colors on the text. In many IDEs you can specify different colors for different items such as keywords, constants, math operators, comments, and so forth. This makes reading the code a little easier.

Let’s edit the source code and have it do something else. Alter the text so that it looks like this:

**Listing 2**

#include <stdio.h>

/\* Program two \*/ int main( void )

{

int x, y;

x = 10;

y = x + 20;

printf("The result is %d\n",y);

}

Rebuild and test the resulting code. You should get a message that says “The result is 30”. Most IDE editors have the usual functionality of cut/copy/paste along with search and replace. Some also have automatic indenting, matching brace creation, and other advanced features.

OK, what happens if you make an error in your code? We shall insert a few on purpose and see what happens. Alter the program above so that it looks like this:

**Listing 3**

#include <stdio.h>

/\* Program three, with errors \*/

int main( void )

{

int x, y;

x = 10

y = x + 20;

printf("The result is %d\n",y);

}

Note that we have left off the trailing semi-colon on x=10; as well as the leading quote on the printf() function. Rebuild the project. This time you will receive a bunch of errors and warnings. They may differ in wording from development system to development system, but you should see something about a missing semi-colon before the y. You’ll probably also see an error concerning “The” being an undeclared identifier. You may see many warnings as well. Usually, double clicking on the error message will highlight the corresponding line in the code. Sometimes a single omission can cause the compiler to emit dozens of error messages. This is because the compiler sort of “loses track” of where it is and starts flagging perfectly good code as having errors. For this reason, if you get errors (and you will), always look at the **first** reported error and fix it. Do not look at the last reported error as it may lead you on a wild goose chase.

Finally, you may wish to save your code for backup. Simply select *File>>Save as* and choose an appropriate name. Again, C source files should use a “.c” extension. Note that you can create, read, or edit C source files without the IDE. All you need is a simple text editor. You won’t be able to compile or build it, but you can at least get some work done on an assignment without a compiler handy.

2

Using Standard I/O (Input/Output)

In this exercise we will be taking a closer look at the Standard I/O functions printf() and scanf(). We shall also investigate the use of multiple user functions and prototypes. Further, we shall investigate the importance of proper program specification as well as choosing proper variable types.

This exercise involves creating a program that will assist with a simple series DC circuit analysis given component tolerances. Suppose that we have a circuit consisting of a DC voltage source and a resistor. We are interested in determining the resultant current through the circuit and the power dissipated by the resistor. This is a fairly straightforward exercise involving just Ohm’s Law and Power Law. To make this a little more interesting, we will include the effects of resistor tolerance. The program will create a table of currents and powers for the nominal, maximum and minimum allowed resistance values. This would be a very useful program for someone just starting their electrical studies. Let us keep this in mind while we design the program.

When designing a program, try to keep the user interaction as simple and as clear as possible. Also, try to structure the code to facilitate maintenance. Studies have shown that majority of programming time spent on non-trivial applications is in the area of maintenance (adding features, fixing bugs, etc.). Strive to make your code as clear as glass and include appropriate comments. Do not, however, add comments to code that even a novice would understand as that just creates clutter. Here is an example of bad commenting:

a = b + c; /\* add b to c \*/

Duh! This comment adds nothing to the quality of the code. Similarly, use mnemonic variable names where possible. This helps to create *self-commenting* code. Here is an example:

x = y + z \* 60;

total\_seconds = seconds + minutes \* 60;

These two lines of code perform the same mathematical operations, but the second line gives you a hint as to what is intended. The first line would probably need a comment to make sense of it while the second line stands by itself.

The Program

The scanf function allows you to accept input from standard in, which for us is generally the keyboard. The scanf function can do a lot of different things, but can be unreliable because it doesn’t handle human errors very well. But for simple programs it’s good enough and easy to use.

The simplest application of **scanf** looks like this:

scanf("%d", &b);

The program will read in an integer value that the user enters on the keyboard (%d is for integers, as is printf, so b must be declared as an int) and place that value into b. The scanf function uses the same placeholders as

* **int** uses **%d**
* **float** uses **%f**
* **char** uses **%c**

You must put **&** in front of the variable used in scanf. The reason why will become clear once you learn about **pointers**. It is easy to forget the & sign, and when you forget it your program will almost always crash when you run it. Try this!

**Listing 4**

#include <stdio.h>

int main(void)

{

int a, b, c;

printf("Enter the first value:");

scanf("%d", &a);

printf("Enter the second value:");

scanf("%d", &b);

c = a + b;

printf("%d + %d = %d\n", a, b, c);

return 0;

}

Here is the (first try) specification for the program:

The program will prompt the user for a DC voltage source value, a nominal resistor value and a resistor tolerance. It will then print out the values for current and power dissipation based on the nominal, minimum and maximum acceptable values of the resistor.

Not bad, but we need to refine it. First, command line programs usually need some form of start-up message or print out of directions. Remember these are not GUI-driven programs with Help menus. Second, always prompt for input values *indicating expected units*. If the program expects Ohms but the user types in kilo Ohms, there’s going to be trouble. Unless there is a compelling reason not to, always use base units (Ohms versus kilo Ohms for example).

Here’s our refined specification:

The program will first give appropriate directions/explanations of use to the user. The program will prompt the user for a DC voltage source value in volts, a nominal resistor value in Ohms and a resistor tolerance in percent. It will then print out the values for current in Amps and power dissipation in Watts based on the nominal, minimum and maximum acceptable values of the resistor.

Note that we have specified tolerance as a percentage rather than as a factor. This is because the typical user would be prepared to enter 10 for 10%, not 0.1. You can use this specification to create a pseudo code or flow chart. Here is a possible pseudo code:

1. Print out directions for user.
2. Prompt user for voltage (in Volts) and obtain value.
3. Prompt user for resistance (in Ohms) and obtain value.
4. Prompt user for tolerance (in percent) and obtain value.
5. Determine maximum and minimum resistance values.
6. Calculate currents based on the three resistances.
7. Calculate powers based on the three resistances.
8. Print a heading for the values.
9. Print out the values.

You could of course choose an alternate algorithm or method of solution. For example, you might prefer to print the heading before the calculations and then print values following each calculation. You might prefer to change the format so that you get rows for each resistor rather than for the current and power. You might even choose an entirely different approach using loops and/or arrays. There will be upsides and downsides to each approach. Often, the question is not “Can I solve this problem?” but rather “What is the most effective way of solving this problem?” Extend a little forethought before you begin coding.

Based on the above pseudo code, the following program should fit the bill. We will refine it later. Note the use of double as we will most likely have fractional values to deal with.

**Listing 5**

#include <stdio.h>

int main(void)

{

double v, tol;

double rnom, rlow, rhigh;

double inom, ilow, ihigh;

double pnom, plow, phigh;

printf("This program determines current and power.\n");

printf("Please enter the voltage source in Volts.\n"); scanf("%lf", &v);

printf("Please enter the nominal resistance in Ohms.\n"); scanf("%lf", &rnom);

printf("Please enter the resistor tolerance in percent.\n"); scanf("%lf", &tol);

tol = (tol/100.0);

rlow = rnom - (rnom\*tol);

rhigh = rnom + (rnom\*tol); inom = (v/rnom);

ihigh = (v/rlow); ilow = (v/rhigh);

pnom = v \* inom; plow = v \* ilow; phigh = v \* ihigh;

printf("Resistance (Ohms) Current (Amps) Power (Watts)\n");

printf("%lf %lf %lf\n", rnom, inom, pnom );

printf("%lf %lf %lf\n", rhigh, ilow, plow );

printf("%lf %lf %lf\n", rlow, ihigh, phigh );

}

A word of caution here: Note that the variable ihigh is the highest current, not the current associated with the highest resistor. This can make the print out code seem incorrect. This is a good place for some comments! Also, the initial “directions” are skimpy at best. In any case, enter and build the code above and verify that it works.

You may have noticed that there is a bit of repetition in this code in the form of calculations and printouts. It may be more convenient if we created functions to handle these. For example, we could create a function to calculate the current:

double find\_current( double voltage, double resistance )

{

double current;

current = voltage/resistance; return( current );

}

You could also do this in one step:

double find\_current( double voltage, double resistance )

{

return( voltage/resistance );

}

Updating the program produces the following:

**Listing 6**

#include <stdio.h>

double find\_current( double voltage, double resistance )

{

return( voltage/resistance );

}

int main(void)

{

double v, tol;

double rnom, rlow, rhigh;

double inom, ilow, ihigh;

double pnom, plow, phigh;

printf("This program determines current and power.\n");

printf("Please enter the voltage source in Volts.\n"); scanf("%lf", &v);

printf("Please enter the nominal resistance in Ohms.\n"); scanf("%lf", &rnom);

printf("Please enter the resistor tolerance in percent.\n"); scanf("%lf", &tol);

tol = (tol/100.0);

rlow = rnom - (rnom\*tol);

rhigh = rnom + (rnom\*tol);

inom = find\_current( v, rnom );

ihigh = find\_current( v, rlow );

ilow = find\_current( v, rhigh );

pnom = v \* inom;

plow = v \* ilow;

phigh = v \* ihigh;

printf("Resistance (Ohms) Current (Amps) Power (Watts)\n");

printf("%lf %lf %lf\n", rnom, inom, pnom );

printf("%lf %lf %lf\n", rhigh, ilow, plow );

printf("%lf %lf %lf\n", rlow, ihigh, phigh );

}

This doesn’t seem to be much of an improvement. In fact, it just seems longer! This is true, but extend the idea a moment. What if the calculation for current involved a dozen lines of code instead of just one? This new format would save considerable code space. Note that this is not just a matter of saving some typing, but rather in saving memory used by the executable. This is particularly important when using constrained embedded systems with only a small amount of available memory.

Note that the new function was added before main(). This is not required. We could also have added it after main(), but in that case we’d have to add a function prototype so that the compiler would know what to expect when it saw the function call in main(). It would look something like this:

#include <stdio.h>

/\* this is the prototype so the compiler can do type checking \*/ double find\_current( double voltage, double resistance );

int main( void )

{

....

}

double find\_current( double voltage, double resistance )

{

return( voltage/resistance );

}

Alter the program to use this new current calculation function and test it. Once this is complete, alter the program one more time to use a function to calculate the power and another to print out the three values. Use the current calculation function as a guide. Finally, consider what might go wrong with the program. What would happen if we the user entered 0 for the resistor value? How could you get around that problem?

3

Using Conditionals

In this exercise we will be taking a look at using conditionals. These include the if/else construct and the switch/case construct. Conditionals are used to make a choice, that is, to split the program flow into various paths. The if/else works best with simple either/or choices while the switch/case is designed to deal with one (or possibly multiple) choice(s) from a list of fixed possibilities.

The simplest conditional is the straight if(). Depending on whether or not the item(s) to be tested evaluate to true determines if subsequent action is taken. By grouping clauses with the logical operators

|| and &&, complex tests may be created. if() statements may be nested as well as include processing for the test failure by using the else clause. If you need to choose a single item from a list, for example when processing a menu selection, this may be achieved through nesting in the following fashion:

if( choice == 1 )

{

}

else

{

/\* do stuff for 1 \*/

if( choice == 2 )

{

}

else

{

/\* do stuff for 2 \*/

if( choice == 3 )

{

/\* do stuff for 3 \*/

}

/\* and so on \*/

}

}

This arrangement is a bit cumbersome when choosing from a large list. Also, it is difficult to deal with a plural choice (e.g., picking items 1 and 3). For these situations the C language offers the switch/case construct. There is nothing a switch/case can do that you can’t recreate with nested if/else’s and additional code, but the former offers greater convenience as well as clearer and more compact code. The switch/case is used frequently, but ultimately, the if/else is more flexible because it is not limited to choosing from a list of numeric values. Ordinarily, numeric constants are not used in production code, as in the example above. Instead, #define’s are used for the constants in order to make the code more readable. A real-world switch/case version of the previous example might look like:

#define WALK\_DOG 1

#define LET\_OUT\_CAT 2

#define COMB\_WOMBAT 3

switch( choice )

{

case WALK\_DOG:

/\* c’mon poochie... \*/ break;

case LET\_OUT\_CAT:

/\* there’s the door... \*/ break;

case COMB\_WOMBAT:

/\* first the shampoo... \*/ break;

/\* and so on \*/

}

In this exercise we’re going to make use of both constructs. The program will involve the calculation of DC bias parameters for simple transistor circuits. We shall give the user a choice of three different biasing arrangements (voltage divider, two-supply emitter, and collector feedback). The program will then ask for the appropriate component values and determine the quiescent collector current and collector-emitter voltage. It will also determine whether or not the circuit is in saturation. These values will be displayed to the user.

One approach to this problem is to consider it as three little problems joined together. That is, consider what you need to do for one bias and then replicate it with appropriate changes for the other two. The three are then tied together with some simple menu processing. Here is a pseudo code:

1. Give the user appropriate directions and a list of bias choices.
2. Ask the user for their bias choice.
3. Branch to the appropriate routine for the chosen bias. For each bias,
   1. Ask for the needed component values (resistors, power supply, beta).
   2. Compute Ic and Vce and determine if the circuit is in saturation.
   3. Display values to the user.

The appropriate equations for each bias follow. All biases use the following: Vcc is the positive supply. Re is the emitter resistor while Rc is the collector resistor. beta is the current gain (hfe). The base-emitter (Vbe) may be assumed to be 0.7 volts. Note that if Ic-saturation is greater than Ic, then the actual Ic is equal to Ic-saturation and Vce will be 0.

Voltage Divider: also requires R1, R2 (upper and lower divider resistors). Vth = Vcc\*R2/(R1+R2)

Rth = R1\*R2/(R1+R2)

Ic = (Vth-Vbe)/(Re+Rth/beta) Vce = Vcc-Ic\*(Re+Rc)

Ic-saturation = Vcc/(Rc+Re)

Collector Feedback: also requires Rb (base resistor).

Ic = (Vcc-Vbe)/(Re+Rc+Rb/beta) Vce = Vcc-Ic\*(Re+Rc)

Ic-saturation = Vcc/(Rc+Re)

Two-supply Emitter: also requires Vee (negative emitter supply) and Rb (base resistor). Ic = (Vee-Vbe)/(Re+Rb/beta)

Vce = Vee+Vcc-Ic\*(Re+Rc)

Ic-saturation = (Vee+Vcc)/(Rc+Re)

where Vee is an absolute value in all cases.

The Program

The program is presented in chunks, below, in the sequence you might write it. First comes the main skeleton.

**Listing 7**

#include <stdio.h> #include <math.h>

#define VOLTAGE\_DIVIDER 1

#define EMITTER 2

#define COLLECTOR\_FEEDBACK 3

#define VBE .7

int main( void )

{

int choice;

give\_directions(); choice = get\_choice();

switch( choice )

{

case VOLTAGE\_DIVIDER:

voltage\_divider(); break;

case EMITTER:

emitter(); break;

case COLLECTOR\_FEEDBACK:

collector\_feedback(); break;

default: /\* tell user they’re not so bright... \*/ printf(“No such choice!\n”);

break;

}

}

The first two functions might look something like this (don’t forget to add their prototypes later):

void give\_directions( void )

{

printf(“DC Bias Q Point calculator\n\n”); printf(“These are your bias choices:\n”); printf(“1. Voltage Divider\n”); printf(“2. Two Supply Emitter\n”); printf(“3. Collector Feedback\n”);

}

int get\_choice( void )

{

int ch;

printf(“Enter your choice number:”); scanf(“%d”, &ch);

return( ch );

}

Now it’s time to write the bias functions. Here is how the voltage\_divider() function might look:

void voltage\_divider( void )

{

double vcc, vth, r1, r2, rth, re, rc, beta, ic, icsat, vce;

printf(“Enter collector supply in Volts”); scanf(“%lf”, &vcc);

printf(“Enter current gain (beta or hfe)”); scanf(“%lf”, &beta);

printf(“Please enter all resistors in Ohms\n”); printf(“Enter upper divider resistor”); scanf(“%lf”, &r1);

printf(“Enter lower divider resistor”); scanf(“%lf”, &r2);

printf(“Enter collector resistor”); scanf(“%lf”, &rc);

printf(“Enter emitter resistor”); scanf(“%lf”, &re);

vth = vcc\*r2/(r1+r2); rth = r1\*r2/(r1+r2);

ic = (vth-VBE)/(re+rth/beta); icsat = vcc/(rc+re);

if( ic >= icsat )

{

}

else

{

}

}

printf(“Circuit is in saturation!\n”);

printf(“Ic = %lf amps and Vce = 0 volts\n”, icsat );

Vce = vcc-ic\*(re+rc);

printf(“Ic = %lf amps and Vce = %lf volts\n”, ic, vce );

The other two bias functions would be similar to this. A few points to note: In order to obtain the absolute value, consider using the fabs() function (floating point absolute). Another approach to the program would be make the vce, icsat, and ic variables globals and move the printout section to main() because the final comparison and printout will be the identical in all three functions. (It is also possible to have the functions return the values via pointers, avoiding the need for globals.)

Complete the other two bias functions, build, and test the program. To avoid compiler warnings, you will need to place function prototypes before main(). You could place these in a separate header file, but there are too few to bother with. To create the prototypes, simply copy and paste the function declarations and add a trailing semi-colon. For example:

#define VBE .7

void give\_directions( void ); int get\_choice( void );

/\* and so forth \*/

void main( void )

{

...

Without the prototypes, the compiler won’t “know” what the functions take as arguments nor what sort of variables they return (if any). Thus, the compiler can’t do type checking and will warn you about this.

Also, the compiler will assume default argument and return types of int, so when the compiler sees the function code, it will complain that the types don’t match the default. This might seem like a pain at first, but it is a cross-checking mechanism that can prevent many software bugs.

Alterations

This program might be useful to a student studying transistors, but it may be a little cumbersome to use. After all, the program needs to be restarted for every circuit. How would you alter the program so that it would ask whether or not the user wanted to try another circuit? In other words, the user could start the program and run it continually for 100 circuits if they so wished.

Perhaps more interestingly, what would be required so that the program could be used for homework generation? That is, the program would create appropriate circuits using new component values each time, and inform the user. It would then ask user to calculate the current and voltage by hand and enter them. The program would then determine if the user’s answers were correct (within a certain percentage to compensate for round off error). Another possibility would be to present the user with a multiple choice list of possible answers instead of having them enter precise values.

Lastly, how might you attempt to “draw” the circuits so that the components are visually identified?