

C Essentials for Embedded Control

David A. Torrey

Department of Electric Power Engineering
Rensselaer Polytechnic Institute

1. Introduction

This document is intended to give you a *very* brief outline of the C programming language. While this introduction is insufficient to get you through the Embedded Control course, hopefully it will get you started up the learning curve. You will inevitably need additional references as you work with the C programming language, such as the Embedded Control lab manual and **The C Programming Language** by Kernighan and Ritchie, among others.

You should read these notes carefully. *You should also understand that these notes are incomplete; no attempt has been made to cover the entire C language or all circumstances that you will encounter in the Embedded Control course.*

2. Program Structure

All of the C programs written for the Embedded Control course fall into a general structure. While there is room for individual programming style, remember that in Embedded Control we are concerned with real-time applications. This implies that we must understand the differences between different variable classes and we must be efficient with our program structure. Do not use three lines of code to do what can be accomplished in one line. General computing is more tolerant of verbose code than real-time applications.

All programs written for the Embedded Control course can be put into the general structure:

- Compiler directives
- Global variable declarations
- Function prototypes
- Main function
- Function definitions

Compiler directives provide specific instructions to the compiler. These instructions include standard function definitions to include in the program and the definition of constants. Sample directives include:

```
#include <stdio.h>    /* Include standard I/O functions */
#include <EVB.h>      /* Include register definitions for the 68HC11 evaluation board*/
#include <math.h>     /* Include standard math functions */
#define PI 3.141592  /* Obvious? */
```

The first three lines tell the compiler to *include*, that is, read in, function and register definitions that are contained in header files. The header files prevent us from having to do a lot of typing and retyping. We only need to define functions once in a header file, then simply include the header file whenever we need those functions. The definition of pi allows us to use a symbolic representation for a number constant. This helps to keep our program more readable. It allows us to define all of our program constants up front. If we then want to change a constant, we only need to do so once in the entire program. This is much safer than doing a find and substitute operation with a word processor. Making constants all capitals

will make the constants stand out in your code.

Global variable declarations allow us to define variables of any type that may be used anywhere in the program. These variables are created at the start of the program and remain in place throughout the execution of the program. All functions of the program have access to global variables. Global variables are generally not needed, except perhaps when you start using the interrupt system of the 'HC11. With interrupts, you never know when processor operation is going to be diverted. In this case it may make sense to use global variables to store results created by the interrupt service routines.

Function prototypes are essentially declaration statements for each function of your program. These declarations tell the compiler the variable type returned by the function and the variable types of the arguments.

Every program must have a function called `main()`. This function can be used to do many things. I generally try to control the program flow in the main function, but I try to use dedicated functions for the actual computations and other actions. I believe this keeps my program more readable by keeping function size small. This is, to a large extent, a matter of my personal programming style. Small functions are, however, generally easier to debug than long unwieldy functions.

Function definitions are required for each function in the program. Within the function definitions are the declaration of local variables, computations, and input/output operations that are performed each time the function is executed. I put all of my function definitions in alphabetical order. This makes it easier for me to find each function as I work to debug it; again, this is my programming style.

3. Variables

There are many kinds of variables in C; there are also many types. The variable type is what you might normally think of: integer, character, float (single-precision), double (double-precision), etc. The maximum size of numbers that will fit into these types is provided in almost any C reference. Remember, in real-time applications, we always use the most compact variable class that will perform the required purpose. Using double precision variables when single precision is sufficient not only makes our program need more storage space, but it also makes the program execute more slowly.

The kinds of variables you are likely to encounter include global, local, static and dynamic. The differences among these kinds of variables are in their scope (where in the program you can access them) and how they are created. Global variables can be accessed anywhere in the program. They are in existence for the entire execution of the program. Local variables are only of use within the function where they are declared. Local variables can be forced to remain in existence during the entire execution of the program if they are declared to be static. That is, they are created and never destroyed. Dynamic variables are local variables that are not static; they are created each time the function is entered.

It is important to understand the subtle differences in these kinds of variables. Their differences are often a source of frustration when they are misapplied. Thinking about the most appropriate variable types as you develop your programs can ultimately save you time and make your code easier to debug. You should also be careful to define your variables at the beginning of the code. For global variables, this is at the top of the program. For local variables, this is after the opening curly bracket.

Sometimes we need to change a variable type temporarily during a calculation. This process is called *type conversion* or *type casting*, and its form is

(cast-type) expression

where `cast-type` is one of the C data types. For example,

```
(int) 12.8 * 3.1
```

casts 12.8 to an `int`, which is 12 after truncation. A cast is a unary operation and follows the same precedence as any other unary operator. Because of this, only the 12.8 is cast to an `int` in the above expression. Parenthesis would be necessary to cast the result into an `int`.

4. Functions

In C, everything gets done in a function. This structure of C is very powerful. It allows you to compartmentalize specific operations, and to call these operations whenever they are needed. As you construct your program you should think carefully about creating a function anytime a segment of code begins to appear more than once.

Build up your programs by starting with the most primitive functions and moving on to the more complex functions. A disciplined approach to function development can save a lot of time "hacking" without any true understanding of the full implications of your changes.

As with the overall program, each function has the same form:

```
function_name (argument list, if any)
{
    declarations
    statements
}
```

You are encouraged to make good use of indentation to keep your program readable. I always have matching curly braces aligned vertically. This makes it easy for me to identify the code that they delimit, thereby simplifying my debugging.

5. Control Flow

There are many ways to control the flow of a function. Control applications often need to make decisions. In these cases, `if-else` statements are used. Their general form is:

```
if (expression)
    statement 1;
else
    statement 2;
```

`Expression` is evaluated and if it is true (nonzero) `statement 1` is executed; otherwise `statement 2` is executed. These constructions can be built into more complicated decisions by augmenting the `if-else` with some `else-if` statements:

```
if (expression 1)
    statement 1;
else if (expression 2)
    statement 2;
else
    statement 3;
```

The loop is ubiquitous in real-time control, where our program does something over and over again. In fact, most real-time control applications intentionally implement infinite loops that will continue to control our system for as long as power is applied. To create our infinite loop, we will routinely use the `while` loop. In

```
while (expression)
{
    statements
}
```

`expression` is evaluated. If it is true (nonzero), the `statements` are executed; the `statements` are not executed if `expression` is false.

The most common loop for making calculations is the `for` loop. The `for` loop has the structure

```
for (expression 1; expression 2; expression 3)
{
    statements
}
```

This structure is equivalent to

```
expression 1
while (expression 2)
{
    statements
    expression 3;
}
```

Grammatically, the three components of a `for` loop are expressions. `Expressions 1` and `3` are generally assignment statements or function calls. `Expression 2` is generally a relational statement. Any of the statements can be omitted, although the semicolons must remain. If `expression 2` is omitted, the test is considered permanently true and the loop becomes infinite, presumably to be broken by either a `break` or `return` statement within the scope of the `for` loop.

The C programming language has other loop structures that are variants of the `while` and `for` loops. Consult a more complete C reference for details on `do-while` loops, the `break` statement and the `continue` statement.

6. Code and Pseudo-Code

It is generally easiest to plan your route on a trip if you first express your trip objectives in words. In the Embedded Control course, our trip is the computer program and the pseudo-code is the words that describe our program. Pseudo-code is the exercise of breaking your overall task down into sufficiently small pieces that your algorithm becomes clear and relatively straightforward to code. Well-written pseudo-code should be able to serve as the comments of your final program. In class we will develop pseudo-code for virtually all of your laboratory exercises.

The following is an example of pseudo-code and the resulting C code.

Code	Pseudo-Code
<pre>#include <stdio.h> #include <math.h> #include <EVB.h> #define TRUE 1 #define FALSE 0</pre>	<p>Compiler directives</p> <p>Global variable declarations</p>
<pre>int sensor (void);</pre>	<p>Function prototypes</p>
<pre>void main (void) { _H11DDRC = 0x00; while (TRUE) { if (sensor()) { _H11PORTB = 0x01; printf ("\r Sensor activated! "); } else { _H11PORTB = 0x00; printf ("\r No sensor activated! "); } } }</pre>	<p>Main function</p> <p>Set port C for input</p> <p>While (TRUE)</p> <p> If sensor is activated</p> <p> Light LED at PB0 Output to display</p> <p> If sensor is not activated</p> <p> Turn off LEDs at Port B Output to display</p> <p>End main function</p>
<pre>int sensor (void) { unsigned char value; value = _H11PORTC & 0x01; if (value == 0) return FALSE; else return TRUE; }</pre>	<p>Function sensor</p> <p> Local declarations</p> <p> Read PORT C Isolate bit 0 If bit 0 is 0, sensor is not activated If bit 0 is 1, sensor is activated</p> <p>End function sensor</p>