Chapter 3 - Programming in C

Since the heart of an embedded control system is a microprocessor, we need to be able to develop a program of instructions for the microprocessor to use while it controls the system in which it is embedded. When programs are developed on the same type of computer system on which they will be run, as is most commonly done, it is called *native platform development.* An example of native platform development is the use of *Borland's C/C++* to develop a program on an *Intel* PentiumII-based computer such as an *IBM*-compatible PC, and subsequently running the program on the same computer.

However, the type of program development which you will be doing in this course is known as *cross platform development*, where the laboratory computer (one platform) is used to develop programs which are targeted to run on the *Motorola* EVB (another platform). Thus even before such a program can be tested, it must be transmitted (downloaded) from the computer to the EVB (see *Introl-CODE on page 10*).

For this embedded microprocessor, we will be using a programming language called 'C'. C is extremely flexible, and allows programmers to perform many low-level functions which are not easily accessible in languages like FORTRAN or Pascal. Unfortunately, the flexibility of C also makes it easier for the programmer to make mistakes and potentially introduce errors into their program. To avoid this, you should be very careful to organize your program so that it is easy to follow, with many comments so that you and others can find mistakes quickly. The example programs were written with this in mind, so that you get an idea of what well structured programs look like. In order to get you started in C programming, this chapter will explain the basics which you will need to begin. If you would like more examples, they can be found in the *LITEC Tutorials* under *Software: C Programming*.

Brief Overview

As stated above, the C language allows many things which other languages do not, making it very easy to make errors. For this reason, it is suggested that you study this entire unit, including the tutorials if necessary, and become familiar with the specifics before beginning the labs. Although this may seem like a lot of work before the first lab, it will be worth your time and will pay off quickly. If there is a question you have regarding the C language that is not included in the manual, or in the tutorials, you will probably find the answer in a C reference text.

A Simple Program in C

The following program is similar to the first programming example used in most C programming books and illustrates the most basic elements of a C program.

```
#include <stdio.h>  /* include file */

main()      /* begin program here */
{           /* begin program block */

      printf("Hello World\n");

            /* send Hello World
               to the terminal */

}           /* end the program block */
```

The first line instructs the compiler to include the *header file* for the standard input/output functions. This line indicates that some of the functions used in the file (such as printf) are not defined in the file, but instead are in an external library (stdio.h is a standard library header file). This line also illustrates the use of comments in C. Comments begin with the two character sequence "`/*`" and end with the sequence "`*/`". Everything between is ignored and treated as comments by the compiler. Nested comments are not allowed.

The second line in the program is `main()`. Every C program contains a function called `main()` which is the function that executes first. The next line is a curly bracket. Paired curly brackets are used in C to indicate a *block* of code. In the case above, the block belongs to the `main()` statement preceding it.

The `printf` line is the only statement inside the program. In C, programs are broken up into *functions*. The printf function sends text to the terminal. In our case, the 68HC11 will send this text over the serial port to the computer terminal, where we can view it. This line also illustrates that every statement in C is followed by a semicolon. The compiler interprets the semicolon as the end of one statement, and then allows a new statement to begin.

You may also notice that the comment after the printf statement continues over more than one line. It is important to remember that *everything* is ignored by the compiler between the comment markers. (see *Specifics of the Introl Compiler on page 24*)

The last line is the closing curly bracket which ends the block belonging to the main function. More examples can be found in the tutorials under the C examples section.

Syntax Specifics

There are many syntax rules in C, but there is neither room nor time here to discuss everything in this manual. Instead, we explain the basics, along with the specifics of the *Introl-CODE* compiler which are not in your textbook. Additional information about the C language can be found in the tutorials, and in any C reference text.

Declarations

One thing which was distinctly missing from the first example program was a variable. The type of variables available with the *Introl-CODE* compiler on the 68HC11 and their declaration types are listed below in *Table 3.1*:

Table 3.1 - Introl-CODE C variable types

Type[a]	Size (bytes)	Smallest Value	Largest Value
integer			
(unsigned) char	1	0	255
signed char	1	-128	127
(signed) short	2	-32768	32767
unsigned short	2	0	65535
(signed) int	2	-32768	32767
unsigned int	2	0	65535
(signed) long	4	-2147483648	2147483647
unsigned long	4	0	4294967295
floating point			
float	4	1.2×10^{-38}	1.2×10^{38}
long float	4	1.2×10^{-38}	1.2×10^{38}
double	4	1.2×10^{-38}	1.2×10^{38}
long double	8	2.2×10^{-308}	18×10^{307}

a. The items in parentheses are not required, but are implied by the definition. We recommend that you state these definitions explicitly to avoid errors due to misdefinition.

The format for declaring a variable in a C program is as follows:

```
<type> variablename;
```

For example, the line

```
int i;
```

would declare an integer variable `i`. Although there are a large variety of variable types available, it is important to realize that the larger the size of the data type, the more time will be required by the 68HC11 to make the calculations. Increased calculation time is also an important consideration when using floating-point variables. It is suggested that in the interest of keeping programs small and efficient, you should not use floating point numbers unless absolutely necessary.

Repetitive Structures

Computers are very useful when repeating a specific task, and almost every program utilizes this capability. The repetitive structures `for`, `while`, and `do..while` are all offered by C.

for Loops

The most common of looping structures is the `for` loop, which looks like this

```
for(initialize_statement; condition; increment){
   ...
}
```

In the example above, the `for` loop will perform the "`initialize_statement`" one time before commencing the loop. The "`condition`" will then be checked to make sure that it is true (non-zero). As long as the "`condition`" is true the statements within the loop block will be performed. After each iteration of the loop, the increment statement is performed. For example:

```
for(i=0; i<10; i++) {
   display(i);
}
```

The statement above will initially set i equal to zero, and then call a user-defined function named `display()`10 times. Each time through the loop, the value of `i` is incremented by one. After the tenth time through, `i` is set to 10, and the for loop is ended since `i` is not less than ten.

while Loops

Another frequently used loop structure is the `while` loop, which follows this format

```
while(condition){
   ...
}
```

When a `while` loop is encountered, the condition given in parenthesis is evaluated. If the condition is true (evaluates to non-zero), the statements inside the braces are executed. After the last of these statements is executed, the condition is evaluated again, and the process is repeated. When the condition is false (evaluates to zero), the statements inside the braces are skipped over, and execution continues after the closing brace. As an example, consider the following:

```
i = 0;
while (i<10) {
   i++;
   display(i);
}
```

The above `while` loop will give the same results as the preceding example given with the `for` loop. The variable `i` is first initialized to zero. When the `while` is encountered in the next line, the computer checks to see if `i` is less than 10. Since `i` begins the loop with the value 0 (which is less than ten), the statements inside the braces will be executed. The first line in the loop, `i++,` increments `i` by 1 and is equivalent to `i=i+1`. The second line calls a function named `display` with the current value of `i` passed as a parameter. After `display` is called, the computer returns to the `while` statement and checks the condition `(i < 10)`. Since after the first iteration of the loop the value of `i` is 1, the condition `(i < 10)` evaluates to logical *TRUE* or equivalently "1", the loop will again be executed. The looping will continue until `i` equals 10 when the condition `(i < 10)` will evaluate as being false. The program will then skip over all the statements within the braces of the `while` construct, and proceed to execute the next statement following the closing brace "`}`".

Arrays

It may be necessary to store a list or table of values of the same type that you want to associate in some way. An array can be used to do this. An array is a group of memory locations all of the same name and data type. Each memory location in the array is called an element and is numbered with the first element as number "0". Note: Be aware, though, that arrays can quickly use up the available variable space, and the compiler does not necessarily check this potential problem. The array is declared with the type of data to be stored in the array as follows:

```
<type> arrayname[maxsize];
```

For example, the lines

```
int values[10];
float timer[60];
```

would declare an array named values that can store up to ten integers (values[0] .... values[9]) and an array named timer that can store up to sixty floating point values (timer[0] .... timer[59]). The array can be initially filled with values when it is declared, or it can later be filled with data by the program as follows:

```
int c[5]={23, 10, 35, 2, 17};   /* c[0]...c[4] is filled with listed values */
int f[5]={0};                    /* f[0]...f[4] is filled with zeros */
for (i=0;i<=4;i++)
   f[i]=i;                    /* fills the elements of f[0]..f[4] with 0..4 */
```

Arrays can also have multiple dimensions. A simple example is an array with multiple rows and columns. These arrays can also be initialized and filled as follows:

```
int data[3][10]={ {0},{0} );     /* initialized data to have three rows and ten
                                    columns filled with zeros */
data[0][5]=26;        /* puts the value 26 into the element at row 0, column 5 */
data[2][9]=5;        /* puts the value 5 into the element at row 2, column 9 */
```

Operators

In addition to a full complement of keywords and functions, C also includes a full range of operators. Operators usually have two arguments, and the symbol between them performs an operation on the two arguments, replacing them with the new value. You are probably most familiar with the mathematical operators such as + and -, but you may not be familiar with the bitwise and logical operators which are used in C. *Table 3.2 - Table 3.6* list some of the different types of operators available. The operators are also listed in the order of precedence in *Table 3.7*. Just as in algebra where multiplication precedes addition, all C operators obey a precedence which is summarized in table *Table 3.7*.

Mathematical

The symbols used for many of the C mathematical operators are identical to the symbols for standard mathematical operators, e.g., add "+", subtract "-", and divide "/". *Table 3.2* lists the mathematical operators.

Table 3.2 - Mathematical operators

operator	description
*	multiplication
/	division
%	mod (remainder)
+	addition
-	subtraction

Relational, Equality, and Logical

The C language offers a full range of control structures including `if..else`, `while`, `do..while`, and `switch`. Most of these structures should be familiar from previous computing classes, so the concepts are left to a reference text on C. In C, remember that any non-zero value is true, and a value of zero is false. Relational, equality, and logical operators are used for tests in control structures, and are shown in *Table 3.3*. All operators in this list have two arguments (one on each side of the operator).

Table 3.3 - Relational, equality, and logical operators

operator	description
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

operator	description
==	equal to
!=	not equal to
||	logical OR
&&	logical AND

Bitwise

C can perform some low-level operations such as bit-manipulation that are difficult with other programming languages. In fact, some of these bitwise functions are built into the language. *Table 3.4* summarizes the bitwise operations available in C.

Table 3.4 - Bitwise and shift operators

operator	description	example	result
&	bitwise AND	`0x88 & 0x0F`	`0x08`
^	bitwise XOR	`0x0F ^ 0xFF`	`0xF0`
|	bitwise OR	`0xCC | 0x0F`	`0xCF`
<<	left shift	`0x01 << 4`	`0x10`
>>	right shift	`0x80 >> 6`	`0x02`

Unary

Some C operators are meant to operate on one argument, usually the variable immediately following the operator. *Table 3.5* gives a list of those operators, along with some example for reference purposes.

Table 3.5 - Unary operators

operator	description	example	equivalent
++	post-increment	`j = i++;`	`j = i;` `i = i + 1;`
++	pre-increment	`j = ++i;`	`i = i + 1;` `j = i;`
--	post-decrement	`j = i--;`	`j = i;` `i = i - 1;`
--	pre-decrement	`j = --i;`	`i = i - 1;` `j = i;`
*	pointer dereference	`*ptr`	value at a memory location whose address is in `ptr`
&	reference (pointer) of	`&i`	the address of `i`
+	unary plus	`+i`	`i`
-	unary minus	`-i`	the negative of `i`
~	ones complement	`~0xFF`	`0x00`
!	logical negation	`!(0)`	`(1)`

Assignment

Most mathematical and bitwise operators in the C language can be combined with an equal sign to create an assignment operator. For example `a+=3;` is a statement which will add 3 to the current value of `a`. This is a very useful shorthand notation for `a=a+3;`. All of the assignment operators have the same precedence as equals, and are listed in the precedence table.

Miscellaneous

Many of the operators in C are specific to the syntax of the C language, and bear other meanings depending on their operands. *Table 3.6* below is a list of some miscellaneous operators which are specific to the C language. This table has been included only as a reference, and you may wish to refer to a C reference manual for complete descriptions of these operators.

Table 3.6 - Miscellaneous operators

operator	description
()	function call
[]	array
->	pointer to structure member access
.	structure member access
(*type*)	type cast
sizeof	size of type (in bytes)
?:	if ? then: else
,	combination statement

Precedence and Associativity

All operators have a precedence and an associativity. *Table 3.7* illustrates the precedence of all operators in the language*. Operators on the same row have equal precedence, and precedence decreases as you move down the table.

* Kernighan and Ritchie, *The C Programming Language*

Table 3.7 - Operator precedence and associativity

operators[a]	associativity
() [] ->.	left to right
! ~ ++ -- * & (type) sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= %= &= ^= |= <<= >>=	right to left
.	left to right

a. This table is from Kernighan and Ritchie, *The C Programming Language.*

Programming Structure Hints

The C programs you develop should be written in a style that is easy for yourself and others to read and maintain (modify). Since issues pertaining to programming style are the topic of entire text books, the following are a few helpful guidelines and hints which are generally regarded as hallmarks of good programming style.

- Strictly follow the C style convention for the indentation of blocks of code.
- Select identifier names for C variables and functions which implicitly describe their functionality.
- Keep the scope of all variables as *local* within functions unless absolutely necessary to globalize their scope.
- As much as possible, encapsulate the functionality of chunks of code into C functions, i.e. adopt a **modular** programming style. It is especially important to avoid having functionally equivalent copies of code dispersed throughout a project comprising one or more files. If the functionality of similar chunks of code can be encapsulated into a

single C function, not only will this result in a reduction of the number of lines of code, but more of the code maintenance can be isolated to *solitary* functions.

– Use comments liberally throughout a program. A good rule of thumb is to include a descriptive comment for about every three lines of code.

Specifics of the *Introl* Compiler

Even though there is an ANSI standard for the C language, not all compilers conform to these standards, and most companies add features to the standard to differentiate their product and to take advantage of the specific hardware for which it compiles. In general, the *Introl-CODE* compiler conforms to the ANSI standard. The following section explains the important differences between the *Introl-CODE* compiler.

Library Functions

As seen in the prior example programs, most of the things done in C, with the exception of low-level functions, involve using library functions. All compilers have their own library functions, but they are usually very close to those defined by the ANSI standard. The *Introl-CODE* compiler is no exception since it includes most of the ANSI functions and contains some extensions specifically for the 68HC11. A list of useful functions is included in *Appendix A - Programming Information* for your reference.

Assembly Code

In-line assembly can be handled by the *Introl-CODE* compiler via the `asm` function if there is a need to optimize a certain recurring segment of code. The compiler then places the enclosed assembly code directly into the C program. The syntax for the `asm` function can be found in the *Introl-CODE* Reference Manual.

Comments

In accordance with the ANSI standard, nested comments (comments within comments) are not allowed.

Definition of an Interrupt Handler Function

Introl-CODE will generate a function as an interrupt handler if it is defined with the `__mod2__` type qualifier. The `__mod2__` type qualifier is not portable to most other compilers. If you wish to test compile your code on another compiler, such as *Borland C/C++* or *gcc* on RCS, you will need to remove `__mod2__` from your function prototypes.