"C-less"

Reference Manual

C language essentials: A concise subset of the C programming language.

Michael J. Bauer, B.E.(Elec.)
Swinburne University of Technology
Melbourne, Australia

Foreword

This is pure C, less a whole raft of esoteric constructs and complexity, aimed at beginners learning to develop programs for embedded microcontroller applications.

The language subset is appropriate for a first course in embedded programming. "C-less" provides a solid foundation on which to build further knowledge and skills in the art.

Program structure and functions

A C program consists of one or more source code files (with ".c" extension), optionally including "header" files (with ".h" extension) containing various definitions and declarations.

A C source code file may contain any or all of the following components:

Comment block with information about the program

Compiler Pre-processor directives

Function prototype declarations

Global (external) variable declarations

Private (static) variable declarations

Function definitions

A C program must have one function named "main". Execution of the program begins with the main( ) function. In embedded microcontroller applications, the main function is typically void of "arguments". (For details, see section on Function Definitions, below.)

Some of the program components listed above may be fragmented and may appear in any order in a source code file, but it is good practice to follow the order shown. In particular, there should be no further #include directives, function prototype declarations, global or private data declarations beyond the first function definition appearing in the source file.

Following is a very simple example of a complete C program. Its purpose is to output a text string "Hello, world!" to a peripheral device, which may be a serial port or LCD screen, etc.

The main function calls a sub-function named "putstr" which is not defined in the source code here, so it must be defined elsewhere. Assume the function code exists in a "library module", which is a pre-compiled object code file. Library modules are linked to the main program (by the compiler "tool chain") before being loaded into the target processor.

However, it is necessary to provide information in the main program about library functions, so the compiler knows how they are to be called. The essential information about putstr( ) is contained in a “header” file named “iodev.h” in this example.

```
/* ==================================================================
 *
 *   File:  hello.c
 *
 *   This program will output the text "Hello, world!".
 */
#include <iodev.h>

int  main( void )
{
    putstr("Hello, world!");
}
```

The components of this program are described in detail below.

Comment block

A “comment block” (also termed a “block comment” or “banner”) may be any text, on one or more lines, beginning with the delimiter “/*” and ending with the delimiter “*/” (without the quotes). Comment blocks should be placed at the top of each source file and above each function definition to describe the function’s purpose, operation and usage.

Example:

```
/* ===================================================================================
 * File:  avr_voltmeter.c
 *
 * This program uses the ATmega88 on-chip ADC to measure the voltage on an input
 * pin (PC4) wired to the potentiometer on the "AVR-Pad" board.
 *
 * Source code from "LCD_Display_v2.c" is used as a framework for this application.
 *
 * Author:      Mike Bauer
 * Originated:  April 2019
 * Version:     1.0
 */
```

Block comment delimiters are also useful for “commenting out” parts of executable code, temporarily, for the purpose of “debugging” (fault-finding) a program. The compiler ignores comments.

Comment line

The two-character sequence “//” introduces a comment on a single line. The comment ends at the end of the line. There may be executable code on the same line, before the comment starts.

Example:

```
    counter = 0;    // initialize a counter
```

Compiler pre-processor directives

Pre-processor directives are commands issued to the compiler. Most do not generate executable code for the target processor, although some may, indirectly.

Pre-processor directives are keywords prefixed by the hash (#) symbol, which must be the first character on a new line. Directives are normally terminated by the end-of-line code (LF), but may be extended onto the next line by placing a back-slash '\' at the end of the line.

The C-less subset allows the following compiler directives:

#include <*system_file*> | *"app_header_file"*

The #include directive is used to include the contents of another file in the compilation, typically a header file containing definitions and declarations required by the program. Note that it is considered poor programming practice to #include another C source file (*.c) containing executable code.

Standard C library header files, intrinsic to the compiler tool-chain, are normally identified using angle brackets to delimit the file-name. Examples:

```
#include <stdlib.h>   // definitions for standard library functions

#include <string.h>   // definitions for standard string functions

#include <math.h>     // definitions for standard math functions
```

Application-specific header files developed for the program are identified using double quotes to delimit the file-name, for example:

```
#include "project_defs.h"   // definitions unique to the project
```

#define *MACRO_NAME [macro_definition]*

The #define directive is used to define a "macro", which is a text string comprising any recognisable C code. Wherever the compiler (pre-processor pass) finds a macro reference (*MACRO_NAME*), it substitutes the associated text (*macro_definition*).

In its simplest and most often used form, a macro is used to assign a symbol (i.e. the macro name) to a numeric constant or expression. These so-called "symbolic constants" help to make source code easier to read and avoid mistakes when a constant value needs to be replaced in multiple occurrences in a program. Examples:

```
#define BUFFER_SIZE  1024

#define NUL  0

#define NEW_LINE  10   // or 0x0A or '\n'

#define UNSIGNED_SHORT_MAX  (256 * 256 - 1)
```

Macro names may include upper-case and lower-case letters, digits (0..9) and the underscore character (_). It is a widely adopted convention to use only upper-case letters in macro names, so that they are easily distinguishable from function names.

Macros can take “arguments” in much the same way as functions can. When the compiler processes a macro reference, it substitutes C expressions, identifiers (variable names) or constants in place of the “dummy” arguments it finds. For example…

The following two macros evaluate the high-order byte and low-order byte (resp.) of an unsigned short (16-bit) integer, represented by the argument (w):

```
#define HI_BYTE(w)  ((w) >> 8))    // shift (w) 8 bit places right

#define LO_BYTE(w)  ((w) & 0xFF)   // keep the low-order 8 bits
```

Wherever the compiler finds `HI_BYTE(x)` in a program, where x is any integer expression, variable or constant, it will substitute the code from the macro definition `((x) >> 8)` thereby giving the value of the argument (x) shifted right 8 bit places.

Note: If x is a signed integer with a negative value, the macro might not evaluate to the expected result. This is because the “bitwise shift right” operator (>>) behaves differently for signed and unsigned integers.

Yet another use for a macro is to define a symbol without any specific value. Such symbols may be used in conjunction with the directives #ifdef and #ifndef to direct the compiler to compile (or to disregard) certain sections of code. Example:

```
#define BUILD_FOR_REV_A_PROTOTYPE
```

(See below descriptions of #ifdef and #ifndef directives for details.)

#if *condition*

Directives #if, #else, #endif are intended for “conditional compilation”. Lines following a #if *condition* directive, up to a corresponding #else or #endif directive, are processed by the compiler only if the condition is “true” (non-zero); otherwise the lines are ignored. Lines in-between the #else and #endif directives are processed by the compiler only if the condition is “false” (zero).

#if … #endif constructs may be nested.

The “condition” is a Boolean expression which may include symbols (defined prior, e.g. by a #define directive), constants and Boolean operators (e.g. ==, !=, >, <, …).

Example:

`#if HARDWARE_REVISION == 2`

[*directives*]

[*executable statements*]

`#endif`

#else

See above description of the #if directive.

The #else clause is optional in the #if … #endif construct.

`#endif`

For each #if directive, there must be a matching #endif directive.

`#ifdef` ***symbol***

The directive construct #ifdef … #endif is also intended for conditional compilation. Lines following a #ifdef *symbol* directive, up to a corresponding #else or #endif directive, are processed by the compiler only if the symbol has been defined prior; otherwise the lines are ignored. For each #ifdef directive, there must be a matching #endif directive.

`#ifndef` ***symbol***

Lines following a #ifndef *symbol* directive, up to a corresponding #else or #endif directive, are processed by the compiler only if the symbol has ***not*** been defined prior.

Function Prototype Declarations

Before a function can be called in a program, the compiler needs to know some of the function's attributes, such as its name, its type (i.e. the data type of its return value, if any), also the ordering and types of its "arguments", if any.

An "argument" (*abbr.* "arg", also known as a "parameter") is a numeric value passed into the function and taking the form of a local variable which may be used in expressions within the function. A special type of argument is a "pointer", which is simply a variable holding the address of a data object in memory, e.g. a text string or array.

The general form of a function prototype declaration is:

```
qualifier  type   function_name( type1 [arg1], type2 [arg2], . . . );
```

… where *type* is the data type of the value returned by the function, e.g. "char", "int", "float"; or "void" if there is no value returned. Data types may have one or more "qualifiers" – these will be explained in a later section. A function name may comprise upper-case letters, lower-case letters, decimal digits (0..9) and the underscore character '_'. Note that a prototype declaration must be terminated with a semicolon.

The argument list (in parentheses after the function name) may be void (empty), or it may contain one or more argument specifiers, separated by commas. In a function prototype, only the *type* of each argument is mandatory. Optionally, an argument name (identifier) may be inserted after each type to help improve readability. Examples:

```
    bool   LCD_Init(void);              // LCD controller initialisation
    void   LCD_ClearScreen(void);       // Clear LCD screen memory
    void   LCD_Mode(char);              // Set pixel write mode (set, clear, flip)
    void   LCD_PosXY(int x, int y);     // Set graphics cursor position to (x, y)
    void   LCD_SetFont(BYTE font_ID);   // Set font for char or text display
    BYTE   LCD_GetFont();               // Get current font ID
    void   LCD_PutChar(char uc);        // Show ASCII char at (x, y)
    void   LCD_PutText(char *str);      // Show text string at (x, y)
    void   LCD_PutDigit(BYTE);          // Show hex/decimal digit value (1 char)
```

Note: The keyword "void" is optional in an argument list which is empty. In the above examples, the data type "BYTE" is not a C keyword and must be defined before the function prototypes. An asterisk (*) preceding an argument identifier indicates that the arg is a pointer, i.e. its value is the address of a data object. For example, the argument in the prototype declaration `LCD_PutText(char *str)` is a pointer to an array of chars, also called a "string". (See elsewhere for more detail on strings and pointer usage.)

Global (external) variable declarations

The C language allows a choice of the "scope" of variables, i.e. which source file(s) can access a variable and which function(s) within a source file can access other variables. The scope of a variable is determined by the place(s) where it is declared and/or by "qualifying" its data type in the declaration.

A variable is said to be "global" if it can be accessed by any function, anywhere in the program, across multiple source files if there is more than one. In C, a variable declared outside of all functions is global by default, except if it is qualified by the keyword "static".

Examples:

```
unsigned  g_ErrorCode;
int  g_PressureLevel_dB;
```

The (optional) prefix "`g_`" makes it obvious that a variable is global.

For a source file to have access to a global variable defined in another source file, the variable must be declared again in the file which requires access, qualified by the keyword "extern".

```
extern unsigned  g_ErrorCode;
```

In a program comprising multiple source files, it is good practice to place such "external" declarations in a common header file #included in the source files which need access.

Private (static) variable declarations

A variable (or function) is said to be "private" if its scope is limited to a particular source file or code "module". The keyword "static" is used to declare a private variable. Example:

```
static int  m_PreviousReading;
```

The (optional) prefix "`m_`" makes it obvious that a variable is private to the module.

The keyword "static" is also a data type qualifier specifying that a variable will be allocated permanent storage at a fixed address in data memory, as opposed to "automatic" variables which are allocated temporary storage (typically in a "LIFO" stack), only when needed. (Details on "automatic" and "local" variables may be found in the following section.)

Function definitions

A C function definition takes the general form:

```
qualifier  type   function_name( type1 [arg1], type2 [arg2], . . . )
{
    data declarations;

    executable statements;
}
```

Not all of the above parts of a function are mandatory. The simplest function definition is:

```
type  function_name( )
{
    executable statement(s);
}
```

The simplest function doesn't even need to have any executable statements.

The mandatory "type" before the function name defines the data type of the return value. A function having no return value must have its type defined as "void". In this case, the function does not need to have a "return" statement – it will exit if and when execution reaches the closing brace. Otherwise, a return statement is mandatory.

Functions are "public" by default, meaning that they are accessible to all code modules comprising the program, assuming that a prototype declaration has been made. A function can be made "private", i.e. accessible only within the source file in which it is defined, by qualifying it as "static" in the type definition. In the author's opinion, this constitutes an overuse of the keyword "static" in the design of the C language. It is recommended to define a macro "PRIVATE" synonymous with "static" to make private function declarations read more sensibly.

```
    #define PRIVATE  static
```

Function names may comprise upper-case letters, lower-case letters, decimal digits (0..9) and the underscore character '_'. (Refer to your compiler manual for specific naming rules.)

Immediately following the function name is a list of "arguments" in parentheses.

An "argument" (*abbr.* "arg", also known as a "parameter") is a numeric value passed into the function and taking the form of a local variable which may be used in expressions within the function. A special type of argument is a "pointer", which is simply the address of some data object in memory, e.g. a text string or array. A function which has no arguments may leave the list empty or put the keyword "void" there instead.

Arguments and variables defined within a function body have their scope restricted to that function. These are termed "local" variables. Note that it is permitted to have two or more local variables with the same name in different functions, but these will be unrelated of course.

The scope of a local variable can be further restricted to just one part of a function called a "code block", which comprises one or more statements delimited by matching braces {…}. The author disapproves of this coding practice, because it can undermine readability and conceal hard-to-find bugs. It is strongly recommended to keep data declarations and executable statements segregated, at the top and bottom of the function body (respectively).

Variables and data type declarations

Basic data types:

`char`	Character (byte), size 8 bits (*typically* unsigned by default)
`int`	Integer, signed by default, may be qualified as unsigned (size is compiler dependent, *typ*. 16 or 32 bits*)
`short`	Short integer, signed by default (*typically* 16 bits*)
`long`	Long integer, signed by default (*typically* 32 bits*)
`float`	Floating-point, always signed (*typ*. 32-bit IEEE format)

*Note: Typical data sizes given are representative of compilers designed for microcontroller applications, as opposed to “high-end” processor applications (e.g. computers and mobile devices running 64-bit operating systems, e.g. Windows, Linux, Android, iOS, etc.)

Data type qualifiers:

`signed`	Qualifies int, short or long (redundant, for emphasis only)
`unsigned`	Qualifies char, int, short or long
`static`	Allocate permanent storage at a fixed address in data memory
`auto`	Automatic – allocate temporary storage while the function is executing. The scope of an “auto” variable is confined to the function in which it is defined. The qualifier “auto” is redundant. Variables are assumed “auto” by default if not qualified as “static”.
`const`	Constant – a “data object” declared as “const” must be initialised in the same statement which defines it. The compiler should complain (i.e. issue an error or warning) if any executable code attempts to modify the contents of any data object declared “const”.

Since “unsigned char” is a frequently used type, but somewhat cumbersome to write, it is recommended to define a type “BYTE” which is synonymous, as follows:

```
#define BYTE  unsigned char
```

Examples of variable declarations:

`char  c;`	ASCII character or control code
`BYTE  b;`	8 bit unsigned number (0 ~ 255)
`int   ival;`	signed integer
`unsigned count;`	unsigned integer (the keyword “int” is implied)
`long  sample;`	signed long integer
`float reading;`	floating-point variable

More than one variable of the same type may be defined on the same line, for example…

`int  i, j, k;`	3 signed integers

Initialisation of Variables

Automatic variables, i.e. non-static, should be initialised by code inside the function in which they appear, especially prior to appearing in a conditionally-executed statement, because the compiler does not implicitly initialise these. (Most compilers will issue a warning about "non-initialised variable which could result in unpredictable execution…" or similar.)

Global (external) and static variables are initialised to zero implicitly at run time, i.e. at the start of program execution, unless explicitly initialised with a non-zero value.

Static variables defined inside a function, i.e. local static variables, are initialised just once at the start of program execution. This applies regardless of whether the initialisation is implicit, or whether it is done explicitly in the variable definition. In other words, local static variables are not (re-) initialised each time the function is called.

Examples of local variable initialisation:

`static char  hit = 1;`	local static variable initialised to 1
`int   i = 0;`	local auto variable initialised to 0

In the above examples, the local auto variable, `i`, will be initialised to 0 every time the function is called. The local static variable, `hit`, will be initialised to 1 at the start of program execution, however. If a statement within the function modifies its value, the last value assigned will be retained for subsequent calls, if any.

Arrays

An array of variables is defined by appending its dimension(s) to its identifier, in square brackets, for example…

```
int  pulseReading[40];
```

This defines a linear (one-dimensional) array of 40 signed integers. A 2D array is defined in the next example, which creates a table of 100 rows by 8 columns (or vice-versa!).

```
float  calibrationTable[100][8];
```

Array elements are indexed in a similar manner. For example, `pulseReading[i]` identifies the `i'`th element of the above array. Note that in C, array indexing starts at zero [0], i.e. there is a "zero'th" element, so the maximum value of the index is (M – 1) where M is the array size.
Of course, you can ignore element 0 and start numbering from 1, if you prefer, but take care to add at least one to the array size defined. And beware of the infamous "out by one" error!

Array Initialisation

Static arrays may be initialised in the same statement that defines them. The general form is:

```
static type  array_name[SIZE] = { k1, k2, k3, ... };
```

… where SIZE is the number of elements reserved in memory for the array and k1, k2, k3, etc, are constants of the same type as the array declaration. The constants are copied into the array from the first element, array_name[0], onwards. If the array SIZE specified is not big enough to hold all of the constants listed, the compiler should complain.

The array size specifier (SIZE) is optional. If omitted, the compiler will allocate just enough storage to hold the supplied constants, i.e. it will determine the size for you. If the program needs to know how many elements an array has been allocated, this number can be found using the operator "sizeof" , which evaluates the number of bytes in its "argument". The following expression is commonly used to find the size (number of elements) of an array:

```
    (sizeof(array) / sizeof(array[0])
```

For example, if the above array has 10 elements allocated and its type is "long" (4 bytes), sizeof(array_name) = 40 and sizeof(array_name[0]) = 4, so the expression's value is 10.

Strings

The term "string" in C has more than one connotation. In broad terms, a string is a set of characters (bytes). There are variable strings, more correctly termed "array of characters", which are held in data memory. A string variable may be defined thus:

`char  mytext[80];` Array of up to 80 characters (bytes)

There may be also constant strings, which is what is normally implied when we talk about a "string" unqualified. A string constant is written as a series of ASCII characters, delimited by double quotes, for example: `"Hello, world!"`. A string constant may be used to initialise a string variable, like this:

```
    char  mytext[] = "Hello, world!";
```

Note that the array size specifier is optional. If omitted, the compiler will do the work of counting the number of characters in the string, adding 1 more for the NUL terminator.

String constants are usually stored in the (Flash) program memory of a microcontroller, which is typically much bigger than the data memory (SRAM).

The compiler automatically appends an ASCII 'NUL' character (code = 0x00) to the end of each string constant defined, not to be confused with the printable digit zero (code = 0x30). A string thus formed is said to be "NUL-terminated". Care should be taken by the programmer to ensure that string variables (arrays) are also NUL-terminated, especially where strings will be manipulated using functions in the standard string library. (See Appendix B.)

There is yet another context in which the term "string" is used. A "string pointer" is a variable holding the address of a string in memory. A pointer is defined by placing an asterisk (*) in front of its identifier. The general form of a pointer definition is:

```
    type  *pointer;
```

… where *type* specifies the data type of an object addressed by *pointer*. Example:

```
    char  *pc;
```

This defines a pointer (pc) which addresses objects of type char, including character arrays, also known as strings. A pointer definition can be extended to initialise it with the address of a data object, for example a string constant, as in the following example:

```
    char  *line1 = "G'day mate!";
```

Note the similarity of the definition to the initialised character array, `mytext`, above. The array identifier (mytext) and the pointer identifier (line1) each have a numeric value associated and those values are memory addresses. The essential difference between them is that the array address is a constant (fixed by the compiler/linker), whereas the pointer is a variable which may be modified by program code execution.

Passing strings as arguments to a function

A string constant or an array of characters can be "passed" to a function as an argument. However, the function does not inherit its own private copy of the string or array. The argument value passed is simply the address of the string or array. The function can use this address to access characters in the (external) string. Examples:

1. `strcpy(mytext, line1);`

2. `strcpy(mytext, "G'day mate!");`

Example 1 shows a standard C library function strcpy( ) being used to copy characters from one string (line1) to another (mytext). Note that the destination "string" (first argument) must be variable, i.e. defined as an array of characters in data memory.

Example 2 shows the same function being used to copy a string constant "G'day mate!" directly to a string variable (mytext).

In both examples, pre-existing data in the array (mytext) is over-written.

Numeric constants

(Note: String constants are described in the foregoing sections.)

C-less allows the following forms of numeric constants, where…

'd' is a decimal digit (0..9), "dd" is one or more decimal digits,
'h' is a hexadecimal digit (0..9, a..f, A..F), "hh" is one or more hex digits,
'1' is a binary digit (i.e. bit, 0 or 1), 'a' is a printable ASCII character.

`dd`	positive decimal integer (e.g. 0, 1, 5, 10, 50, 100, 9999)
`−dd`	negative decimal integer (e.g. −1, −5, −10, −999)
`0..9, 0xhh, 0Xhh`	hexadecimal number (e.g. 0, 9, 0x00, 0xCF00, 0X8080)
`0, 1, 0b11, 0B11`	binary number (e.g. 0, 1, 0b101, 0B0101, 0b11001100)
`dd.dd`	floating-point, double (e.g. 0F, 1. 5, 100., 0.1, 100F) *
`'a'`	ASCII character (e.g. 'a', 'Z', '0', '9', '$', '?', '#', '+', …)
`'\?'`	ASCII control code (e.g. '\0' = NUL, '\n' = LF, '\r' = CR)

* Note: The precision of floating-point operations is compiler-dependent. In many compilers, floating-point arithmetic uses double-precision (64 bits) to evaluate expressions containing a constant with a decimal point. Single-precision can be forced by appending the suffix 'F' to floating-point constants, or by using the "cast" (float) in front. There may be a compiler option to force all floating-point arithmetic to be single-precision (32 bits), thereby reducing object code size and speeding up floating-point operations.

An integer constant is forced to be “long” (32 bits) by adding the suffix ‘L’.
An integer constant is forced to be “unsigned” by adding the suffix ‘U’.
An integer constant is forced to be “unsigned long” by adding the suffix ‘UL’.

Boolean (logical) constants

C does not provide intrinsic values for Boolean constants, e.g. “TRUE” or “FALSE”, but it does provide for conditional expressions which evaluate to zero or non-zero. It is common practice to define two symbolic constants, TRUE and FALSE, in a header file, as follows:

```
#ifndef FALSE  // if not defined elsewhere...
#define FALSE  0
#define TRUE   (!FALSE)
#endif
```

The above directives prevent `TRUE` and `FALSE` from being redefined in case they are already defined somewhere else, perhaps in a library header file #included in the program. Note that `TRUE` is not given a specific value, e.g. 1 or −1, because it may be *any non-zero value*. All we know is that `TRUE` is not `FALSE`. Let the compiler deal with it.

Executable statements

Assignment statements

General form:

```
variable = expression;
```

The value of the *expression* is first computed, then assigned (copied) to the *variable*.

An “expression” is a combination of function calls, variables, constants and “operators”. The simplest expression is a constant. This example assignment statement copies a constant value to a variable:

```
max = 1000;
```

Arithmetic expressions may comprise one or more operands and operators, as in the following example assignment statements. (See Appendix A for the full set of operators.)

```
average = (a + b) / 2;

count = count + 1;

reading = ADC_Read(input);

voltage_mV = (ADC_Read(input) * 3300) / 1024;

portbits = PORTA & bitmask;
```

The compiler will convert the value of the expression to the data type of the assigned variable, except where the expression evaluates to an incompatible data type. C is very permissive, but unforgiving, in this regard. Beware of the possibility of truncation of value or other unwanted aberration when assigning an expression to a variable of different type, in particular a smaller size, e.g. assigning an `int` to a `char`. The compiler will not complain, unless it has its “warning” options set to a very sensitive level. In this example, `byte1` is a `char` and `ival` is an `int`, so the expression evaluates to int, which could exceed the size of `byte1` (8 bits):

```
    byte1 = ival * 10;
```

Expressions involving operands of different types are permitted in C. The compiler will "escalate" (i.e. change) the type of an operand where two operands of different types are mixed in an expression. For example, if an integer and floating-point operand are involved, the integer will be "escalated" to floating-point and the resulting type will be `float`.

Comparison (Boolean) expressions

An expression may be a comparison between two operands. This form of expression evaluates to zero if the result is "false"; non-zero if the result is "true". Examples:

`(a == b)`	Test for equality (a equal to b)
`(a != b)`	Test for inequality (a NOT equal to b)
`(i >= 0)`	Test for positive value
`(a > b)`	Test for "`a` is greater than `b`"
`(flag == 0)`	Test for zero (or FALSE)

The Boolean "equality" operator (**`==`**) is not to be confused with the "assign" operator '**`=`**'. Take care not to use the latter in a conditional expression unintentionally. It's a trap for beginners. The compiler will not complain because an assignment statement may be used in place of a conditional expression. It's not good programming practice, but perfectly valid.

By the way, avoid using this expression, because `TRUE` may be any non-zero value:

```
   if (whatever == TRUE) ...
```

Instead, use the following expression which will always evaluate as intended:

```
   if (whatever) ...    // correct test for "whatever is TRUE"
```

Conversely, there is nothing wrong with using the expression:

```
   if (whatever == FALSE) ...
```

Although the preferred equivalent expression is simply:

```
   if (!whatever) ...   // preferred test for "whatever is FALSE"
```

Loop constructs (while, for)

A "while" loop takes the general form:

```
    while (condition)
    {
        statement(s);
    }
```

The statement(s) within the loop body, i.e. between the braces {...}, will be executed repetitively while the *condition* (expression inside round brackets) evaluates to non-zero.

The *condition* expression is tested at the start of each iteration of the loop, so if it is already zero on the first test, the loop body will not be executed at all. When the condition evaluates to "false" (zero), the loop exits and execution follows on from the closing brace. A "break" statement may be placed anywhere in the loop body to provide an additional exit condition.

Example:

```
while (count > 0)
{
    total = total + readValue();   // add some quantity to total
    count--;                       // decrement the loop count
}
```

Statements within the loop body should be indented as shown. Loops may be nested, i.e. one or more loops may be placed (suitably indented) inside the body of another. Although there is no limit to the number of levels of nesting of loops, it's good practice to confine it to 2 or 3 levels. Where there's a lot happening inside a loop, it's best to call functions within the loop to keep it simple and to improve readability.

A "for" loop takes the general form:

```
for (init_statement ; condition ; next_statement)
{
    statement(s);
}
```

The first statement (*init_statement*) is executed just once at the start of the loop, before any statement within the loop body. The statement(s) within the loop body, between the braces, will be executed repetitively while the middle expression (*condition*) evaluates to non-zero. The statement "*next_statement*" is executed at the completion of each iteration of the loop body, i.e. every time around the loop.

The *condition* expression is tested at the start of each iteration of the loop, so if it is already zero on the first test, the loop body will not be executed at all. When the *condition* evaluates to "false" (zero), the loop exits and execution follows on from the closing brace. A "break" statement may be placed anywhere in the loop body to provide an additional exit condition.

Example:

```
for (i = 0 ; i < 100 ; i++)
{
    Buffer[i] = 0;
}
```

In this example, an array of 100 elements is cleared. The index variable (i) will increment from 0 to 99 inside the loop, and again on exit, so after the loop is done the index value will be 100.

Statements within a loop body should be indented as shown in the examples. Loops may be nested, i.e. one or more loops may be placed (suitably indented) inside the body of another. It is good practice to restrain nesting of loops to 2 or 3 levels. Where there's a lot happening inside a loop, it's best to call functions within the loop to keep it uncluttered.

Conditional execution (the "if" construct)

There are several forms of the "if" construct to be considered.

Here is the "short" form, where the statement is short enough to fit on a single line:

```
if (condition) short_statement;
```

The short statement following the *condition* (expression inside the brackets) is executed only if the condition is "true", i.e. evaluates to a non-zero value. The short statement, as with all "simple" statements in C, must be terminated with a semi-colon.

Example:

```
if (reading > 1000)  reading = 1000;   // Cap reading at 1000
```

Optionally, an "else" clause may be added where another simple statement is to be executed in case the condition expression is "false", i.e. it evaluates to zero. Here is the general form:

```
if (condition) short_statement1;
else  short_statement2;
```

The short statement following "else" is executed only if the condition is false (0).

A more complex form of the "if" statement allows a "compound statement" to be executed conditionally. Here is the general form:

```
if (condition)
{
    statement1;
    statement2;
        :
}
```

A compound statement comprises two or more simple statements enclosed within braces. Here, the compound statement body will be executed if the condition is true (non-zero). Similarly to the simple form, an "else" clause may be added, thus:

```
if (condition)
{
    statement01;
    statement02;
        :
}
else
{
    statement11;
    statement12;
        :
}
```

As with loops, "if" statements may be nested, i.e. a statement within an "if" construct may be another "if" statement.

Example:

```
if (reading < 0)
{
    if (retryCount > 3)  flagError = TRUE;
}
```

An equivalent alternative coding of this example, which is more compact, is this:

```
if (reading < 0 && retryCount > 3)  flagError = TRUE;
```

Here, the operator (&&) performs a "logical AND" and so the error flag will be set true if the reading is negative and the "retry count" is greater than 3. Note that the "less than" (<) and "greater than" (>) relational operators have higher precedence than the AND operator (&&), so the relational expressions are evaluated before the logical AND operation.

A complete list of C operators is provided in Appendix A.

Lastly, there is a particularly useful "nested if" construct which warrants a mention. This construct tests a number of conditions in succession and executes the statement corresponding to the first condition which evaluates "true" (non-zero), if any. Regardless of whether any further conditions may be true, no further statements will be executed, because that's the way the "else" clause works. Here's the general form:

```
if (condition1)  statement1;
else if (condition2)  statement2;
else if (condition3)  statement3;
   :         :             :
else if (conditionK)  statementK;
else  default_statement;
```

Note that these statements may be simple or compound. In the latter case, the semicolons must be omitted. If not, the compiler should complain, because a construct can't begin with "else".

The final "else" clause may be omitted, but it is recommended to provide a "default" statement in case none of the preceding conditions is "true".

Control of program flow (return, break)

The **return** statement, when executed, causes a function to exit. Program flow continues from the statement following the function call. There may be more than one return statement in a function, but it is good practice to have just one, placed at the bottom of the function body.

A function declared "void" does not need to have a return statement, but if it does, it cannot return a value. If there is no "return" statement, the function will exit when program flow reaches the closing brace.

Functions not declared "void" must have at least one, preferably only one, return statement and it must specify a value to be returned. In this case, the syntax of the return statement is:

```
return expression;
```

where *expression* evaluates to the same data type as in the function declaration.

The body of a loop construct (`while` or `for`) may contain one or more **break** statements, the syntax of which is simply:

```
break;
```

Whenever a break is executed, the loop exits. Program flow resumes at the next statement following the closing brace of the loop. Most often, a break is conditional, i.e. it is incorporated into an "if" statement. For example:

```
while (count > 0)
{
    total = total + readValue();
    if (total > 10000)  break;     // total exceeds maximum .. exit
    count--;
}
if (count != 0)  return ERROR;     // bail from function
```

The `break` statement is also used in the **`switch...case`** construct, which is beyond the scope of this subset of the C language.

References

For examples of complete, practical C programs created for entry-level microcontroller projects, please refer to the companion document, written by the same author:

"**C-less: Tutorial Introduction**".

Also by this author is a guide to good programming practices and coding style, entitled:

"**Embedded C Coding Standard**".

These publications and many other resources for embedded software development can be found on the author's website, here:

http://www.mjbauer.biz/mjb_resources.htm

Contact: `mjbauer@iprimus.com.au`

Copyright Notice

Appendix A -- Operators in order of precedence (highest at the top)

Operator	Operation
`( )`	brackets, round
`[ ]`	brackets, square
`++`	increment (prefix or postfix)
`--`	decrement (prefix or postfix)
(*type*)	type cast (prefix) – converts data type of operand
`*`	data at pointer (prefix)
`&`	address of data object (prefix)
`-`	unary minus (prefix)
`~`	bitwise complement, invert bits (prefix)
`!`	logical complement, Boolean NOT (prefix)
`*`	multiply
`/`	divide
`%`	modulo (remainder), integer operands
`+`	add
`-`	subtract
`>>`	shift right *n* bit places (*n = op2*) [see note]
`<<`	shift left *n* bit places (*n = op2*)
`>`	is greater than
`>=`	is greater than or equal to
`<=`	is less than or equal to
`<`	is less than
`==`	is equal to
`!=`	is not equal to
`&`	bitwise AND
`^`	bitwise XOR (exclusive OR)
`|`	bitwise OR (inclusive OR)
`&&`	logical AND
`||`	logical OR
`=`	assign (*variable = expression*)
`+=`	add then assign back (*variable += expression*)
`-=`	subtract then assign back (*variable −= expression*)
`*=`	multiply then assign back (*variable *= expression*)

Note: The “bitwise shift right” operator (>>) behaves differently for signed and unsigned integers. With unsigned, for each bit place shifted, the most significant bit is zeroed. This is termed a “logical shift right”. With signed integers, the most significant bit (sign bit) is preserved. This is termed “arithmetic shift right” and this is commonly used for fast integer division by a power of two (2, 4, 8, 16, …).

Appendix B -- Standard C Library Functions

A function sub-set selected for low-level embedded microcontroller applications.

String functions -- defined in `<string.h>`

`strlen(s)`	Returns the length of string (s), excluding the NUL terminator.
`strcpy(s1, s2)`	Copies a constant string (s2) to a variable string (s1).
`strcat(s1, s2)`	Concatenates (i.e. appends) one string (s2) to another (s1).
`strcmp(s1, s2)`	Compares one string (s2) to another (s1) alpha-numerically; returns zero if the strings are identical; returns positive (+1) if `s2` is higher than `s1` in the "sort order"; otherwise returns negative (−1).
`strncmp(s1, s2, n)`	Same as `strcmp( )`, but compares only the first `n` characters.

Utility functions -- defined in `<stdlib.h>`

`abs(i)`	Returns the absolute value (magnitude) of an integer.
`labs(n)`	Returns the absolute value (magnitude) of a long integer.
`rand()`	Returns a "random" number (int) between 0 and `RAND_MAX`.
`srand(u)`	Initialises (seeds) the PRNG sequence with a value (u).
`itoa(i, abuf, n)`	Converts a signed integer (`i`) to an ASCII numeric string; the 2nd arg (`abuf`) is a pointer to the result string (array); the 3rd arg (n) is the number base (typically 10 for decimal); leading zeros are removed. (Use `strlen(abuf)` to find length.)
`atoi(str)`	Converts a string (`str`) to a signed integer and returns its value; the string may contain ASCII digits ('0' to '9') and a minus sign.
`atol(str)`	Same as `atoi( )`, except returns a signed long integer.

Floating-point math functions -- defined in `<math.h>`

Arguments and return values may be double (64 bit) or single (32 bit) precision, depending on the compiler default and option settings.

`sin(a), asin(x)`	Sine and inverse sine functions, (a) is in radians.
`cos(a), acos(x)`	Cosine and inverse cosine functions, (a) is in radians.
`tan(a), atan(x)`	Tangent and inverse tan functions, (a) is in radians.
`exp(x)`	Natural exponential function; returns **e** to the power **x**.
`log(x)`	Natural logarithm function; returns log of **x** in base **e**.
`log10(x)`	Base-10 logarithm function; returns log of **x** in base 10.
`pow(x, y)`	Returns **x** raised to the power **y**.
`sqrt(x)`	Returns the square root of **x**.

Note: Inclusion of any floating-point math function in a program will result in a substantial increase in the object code file size. For very "low-end" microcontroller applications, and/or where computational speed is a requirement, floating-point arithmetic and math library functions should be avoided.