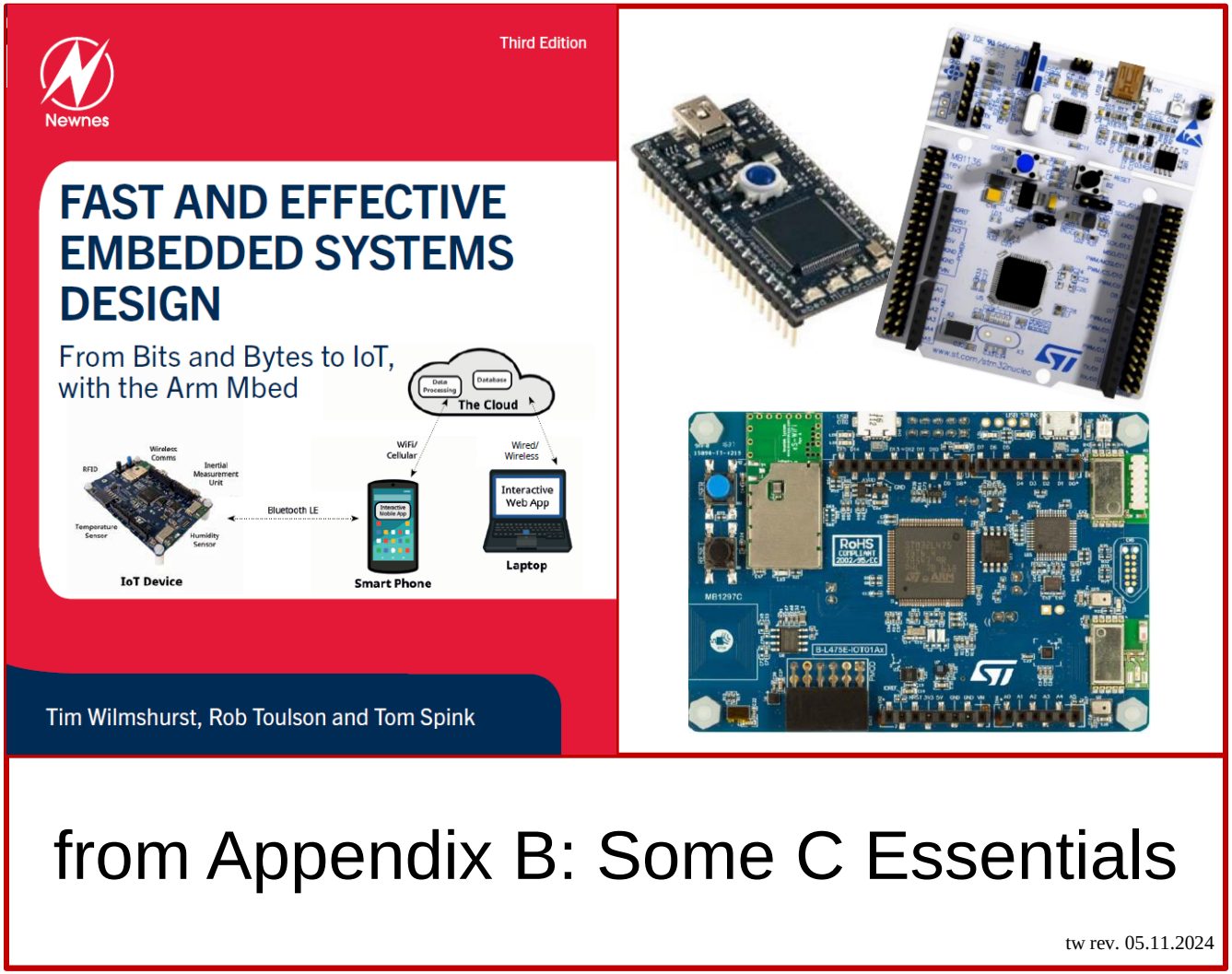


from Appendix B: Some C Essentials

tw rev. 05.11.2024

If you use or reference these slides or the associated textbook, please cite the original authors' work as follows:

Wilmshurst, T., Spink, T. & Toulson, R. (2024). Fast and Effective Embedded Systems Design – From Bits and Bytes to IoT, with the ARM Mbed (3rd edition), Newnes, Oxford, ISBN: 978-0323951975.

C Keywords

C has 32 reserved "keywords". These are used for program flow control, and declaration of data types. They cannot be used by the programmer as names. A selection are shown below.

Word	Summary meaning	Word	Summary meaning
char	A single character, usually 8-bit	**signed**	A qualifier applied to char or int (default for char and int is signed)
const	Data that will not be modified	**sizeof**	Returns the size in bytes of a specified item, which may be variable, expression or array
double	A ‘double precision’ floating-point number	**struct**	Allows definition of a data structure
enum	Defines variables that can only take certain integer values	**typedef**	Creates new name for existing data type
float	A ‘single precision’ floating-point number	**union**	A memory block shared by two or more variables, of any data type
int	An integer value	**unsigned**	A qualifier applied to char or int (default for char and int is signed)
long	An extended integer value; if used alone, integer is implied	**void**	No value or type
short	A short integer value; if used alone, integer is implied	**volatile**	A variable which can be changed by factors other than the program code

C keywords associated with data type and structure definition

C Keywords

Word	Summary meaning	Word	Summary meaning
break	Causes exit from a loop	**for**	Defines a repeated loop – loop is executed as long as condition associated with for remains true
case	Identifies options for selection within a switch expression	**goto**	Program execution moves to labelled statement
continue	Allows a program to skip to the end of a for, while or do statement	**if**	Starts conditional statement; if condition is true, associated statement or code block is executed
default	Identifies default option in a switch expression, if no matches found	**return**	Returns program execution to calling routine, causing also return of any data value specified by function
do	Used with while to create loop, in which statement or code block following do is repeated as long as while condition is true	**switch**	Used with case to allow selection of a number of alternatives; switch has an associated expression which is tested against a number of case options
else	Used with if, and precedes alternative statement or code block to be executed if if condition is not true	**while**	Defines a repeated loop – loop is executed as long as condition associated with while remains true

C keywords associated with program flow

Word	Summary meaning	Word	Summary meaning
auto	Variable exists only within block within which it is defined. This is the default class	**register**	Variable to be stored in a CPU register; thus, address operator (&) has no effect
extern	Declares data defined elsewhere	**static**	Declares variable which exists throughout program execution; the location of its declaration affects in what part of the program it can be referenced

C keywords associated with data storage class

Program Features and Layout

C is a so-called free-form programming language. That means that there is not a strict layout format.

Simply speaking, a C program is made up of:

- *declarations* – which set the scene and initialise things.
- *statements* – where the programming action takes place.
- *space* – which provides essential gaps between the words and symbols used, and is also used to improve clarity through the way the code is laid out.
- *comments* – which provide a commentary to the human reader on what is going on.

Declarations

All variables in C must be declared before they can be applied, giving as a minimum variable name and its data type. A declaration is terminated with a semicolon. In simple programs, declarations appear as one of the first things in the program. They can also occur within the program, with significance attached to the location of the declaration.

For example:

```
float exchange_rate;
int new_value;
```

declare a variable called **exchange_rate** as a floating point number, and another variable called **new_value** as an integer.

Statements

Statements are where the action of the program takes place. They perform mathematical or logical operations, and establish program flow. Every statement which is not a block ends with a semicolon. Statements are executed in the sequence they appear in the program, except where program branches take place.

For example, this line is a statement:

```
counter = counter + 1;
```

Space and layout

There is not a strict layout format to which C programs must adhere. The way the program is laid out, and the use of space, are both used to enhance clarity. Blank lines and indents in lines for example are ignored by the compiler, but used by the programmer to optimise the program layout.

As an example, the Mbed “blinky” program could be written as below. It wouldn’t be easy to read however. It’s the semi-colons at the end of each statement, and the brackets, which in reality define much of the program structure.

```
int main(){DigitalOut led(LED1);while(true){led=!led;ThisThread::sleep_for(500ms);}}
```

Comments

Two ways of commenting are used:

- place the comment between the markers ***/*** and ******* **/**. This is useful for a block of text information running over several lines.
- two forward slash symbols (***//***), the compiler ignores any text which follows on that line only.

For example:

```
/*A program which flashes mbed LED1 on and off,
Demonstrating use of digital output and wait functions. */

#include "mbed.h"     //include the mbed header file as part of this program
...
```

Code blocks

Declarations and statements can be grouped together into *blocks*. A block is contained within braces, i.e. { and }. Blocks can and are written within other blocks, each within its own pair of braces. Keeping track of these pairs of braces is an important pastime in C programming, as in a complex piece of software there can be numerous ones nested within each other.

Compiler Directives

Compiler directives are messages to the compiler, and do not directly lead to program code. They start with a hash, `#`. Examples include:

#include

The **#include** directive directly inserts another file into the file that invokes the directive. This provides a feature for combining a number of files as if they were one large file. For example:

```
#include "mbed.h"
```

#define

This allows use of names for specific constants. For example:

```
#define    PI          3.141592
```

The name ‘PI’ is then used in the code whenever the number is needed. When compiling, the compiler replaces the name in the **#define** with the value that has been specified.

Data Types

When a data declaration is made, the compiler reserves for it a section of memory, whose size depends on the type invoked. Examples are shown.

Note that the actual memory size applied to data types can vary between compilers.

Data type	Description	Length (bytes)	Range
char	Character	1	0 to 255
signed char	Character	1	-128 to +127
unsigned char	Character	1	0 to 255
short	Integer	2	-32768 to +32767
unsigned short	Integer	2	0 to 65 535
int	Integer	4	–2147483648 to + 2147483647
long	Integer	4	-2147483648 to + 2147483647
unsigned long	Integer	4	0 to 4294967295
float	Floating point		$1.17549435 \times 10^{-38}$ to $3.40282347 \times 10^{+38}$
double	Floating point, double precision		$2.22507385850720138 \times 10^{-308}$ to $1.79769313486231571 \times 10^{+308}$

Example C data types, as implemented by the Mbed compiler

Working with Data

When writing numbers in a C program, the default radix (number base) for integers is decimal, with no leading 0 (zero). We can also work with numbers in floating point decimal, octal or hexadecimal format, depending on what is most convenient, and what number type and range is required.

Often it's convenient to work in decimal, but if a variable represents a register bit field or a port address, then it can work better to manipulate the data in hexadecimal.

Octal numbers are identified with a leading 0. Hexadecimal numbers are prefixed with 0x.

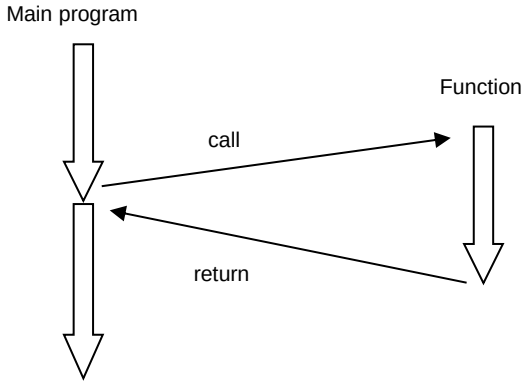
For time critical applications it is important to remember that floating point calculations can take much longer than fixed point.

For example if a variable **MyVariable** is of type **char** we can perform the following examples to assign a number to that variable. The value for each of the following is the same.

```
MyVariable = 14;      //a decimal example
MyVariable = 0x0E;    //a hexadecimal example
MyVariable = 016;     //an octal example
```

Functions

A function is a section of code which can be called from another part of the program, usually if it needs to be used more than once. Using functions saves coding time and improves readability by making the code neater.

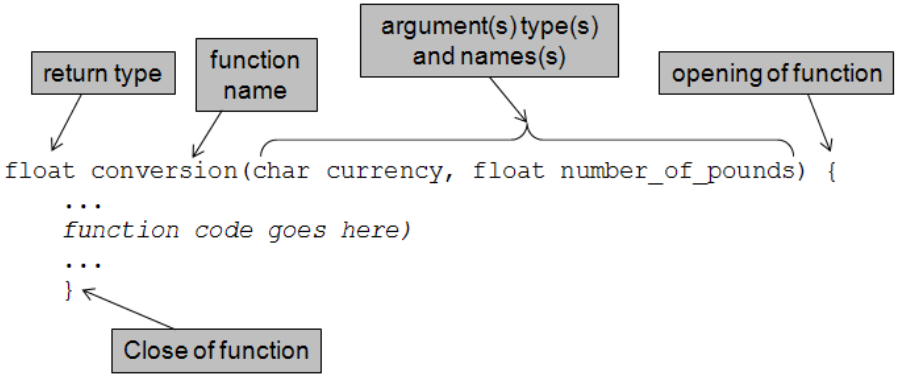


Data can be passed to functions, and returned from them. Such data elements, called *arguments*, must be of a type which is declared in advance.

Only one return variable is allowed, whose type must also be declared.

The data passed to the variable is a *copy* of the original. The function does not itself modify the value of the variable named.

A function is defined in a program by a block of code having particular characteristics. Its first line forms the function header, with example format:



The *main* function

The core code of any C program is contained within its ‘main’ function. Other functions may be written outside **main( )**, and called from within it. Program execution starts at the beginning of **main( ).**

As **main( )** contains the central program, one expects to send nothing to it, nor receive anything from it. Usual patterns for **main( )** are:

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

The keyword **void** indicates that no data is specified. The mbed **main( )** function applies the third option, as in C++ **int** is the return type specified for **main( )**.

Function Prototypes

Just like variables, functions must be declared at the start of a program, before the main function. The declaration statements for functions are called prototypes. Each function in the code must have an associated prototype for it to be located by the compiler, unless it is defined in full before its first use. The format is the same as for the function header.

For example, the following function prototype applies to the function header seen above:

```
float conversion(char currency, float number_of_pounds);
```

This describes a function that takes inputs of a character value for the selected currency and a floating point (decimal) value for the number of pounds to be converted. The function returns the decimal monetary value in the specified currency.

Writing Functions

The actual function code is called the *function definition.* For example:

```
float conversion(char currency, float number_of_pounds) {
  float exchange_rate;
  switch(currency) {
    case 'U': exchange_rate = 1.50;            // US Dollars
      break;
    case 'E': exchange_rate = 1.12 );          // Euros
      break;
    case 'Y': exchange_rate = 135.4);          // Japan Yen
      break;
    default: exchange_rate = 1);
  }
  exchange_value=number_of_pounds*exchange_rate;
  return(exchange_value);
}
```

This function can be called any number of times from within the main C program, or from another function, for example in this statement:

```
ten_pounds_in_yen=conversion('Y',10.45);
```

C Operators

C has a wide set of operators, examples are shown. The symbols used are familiar, but their application is *not* always the same as in conventional algebra. For example, a single ‘equals’ symbol, ‘=’, is used to assign a value to a variable. A double equals sign, ‘= =’, is used to represent the conventional ‘equal to’.

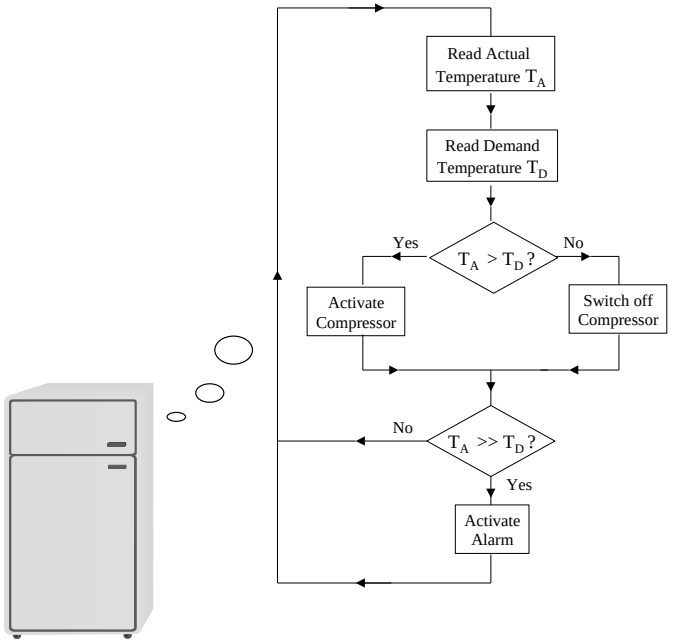
Prec. and order	**Operation**	**Symbol**	**Prec. and order**	**Operation**	**Symbol**
Parethenses and array access operators					
1, L to R	Function calls	( )	1, L to R	Point at member	X–>Y
1, L to R	Subscript	[ ]	1, L to R	Select member	X.Y
Arithmetic operators					
4, L to R	Add	X+Y	3, L to R	Multiply	X*Y
4, L to R	Subtract	X–Y	3, L to R	Divide	X/Y
2, R to L	Unary plus	+X	3, L to R	Modulus	%
2, R to L	Unary minus	–X			
Relational operators					
6, L to R	Greater than	X>Y	6, L to R	Less than or equal to	X<=Y
6, L to R	Greater than or equal to	X>=Y	7, L to R	Equal to	X= =Y
6, L to R	Less than	X<Y	7, L to R	Not equal to	X!=Y

Example C Operators

Flow control: *if* and *else*

If statements always start with use of the **if** keyword, followed by a logical condition. If the condition is satisfied, then the code block which follows is executed. If the condition is not satisfied, then the code is not executed. There may or may not also be following **else** or **else if** statements.

```
if (Condition1){
        …C statements here
}else if (Condition2){
        …C statements here
}else if (Condition3){
        …C statements here
}else{
        … C statements here
}
```



Example:

```
if (data > 10){
  data += 5;                  //If we reach this point, data must be > 10
  }
  else if(data > 5){          //If we reach this point, data must be <= 10
  data -= 3;
  }
  else{                      //If we reach this point, data must be <= 5
  nVal = 0;
  }
```

Flow control: *while* loops

A **while** loop is a simple mechanism for repeating a section of code, until a certain condition is satisfied. The condition is stated in brackets after the word **while**, with the conditional code block following. For example:

```
i=1
while (i<10) {
  … C statements here
  i++             //increment i
}
```

Here the value of **i** is defined outside the loop; it is then updated within the loop. Eventually **i** increments to 10, at which point the loop terminates. The condition associated with the while statement is evaluated at the start of each loop iteration; the loop then only runs if the condition is found to be true.

Flow control: *for* loops

The **for** loop allows looping, with the dependent variable updated automatically every time the loop is repeated. It defines an initialised variable, a condition for looping, and an update statement. Note that the update takes place at the end of each loop iteration. If the updated variable is no longer true for the loop condition, the loop stops and program flow continues. For example:

```
for(j=0; j<10; j++) {
        … C statements here
}
```

Here the initial condition is `j=0` and the update value is `j++` , i.e. **j** is incremented. This means that **j** increments with each loop. When **j** becomes 10 (i.e. after 10 loops), the condition `j<10` is no longer satisfied, so the loop does not continue any further.

Flow control: Infinite loops

We often require a program to loop forever, particularly in a super-loop program structure. An infinite loop can be implemented by either:

```
while(1) {              //while(true) may also be used
  … continuously called C statements here
}
```

Or

```
for(;;) {
      … continuously called C statements here
 }
```

Arrays

An array is a series of data elements, each of the same type. Any data type can be used. Array elements are stored in consecutive memory locations. An array is declared with its name and the data type of its elements, recognised by the use of the square brackets. The number of elements, and their value can also be specified. For example, the declaration

```
char message1[8];
```

defines an array called **message1**, containing eight characters. Alternatively it can be left to the compiler to deduce the array length, as seen in the two examples here:

```
char item1[] = "Apple";
int nTemp[] = {5,15,20,25};
```

In each of these the array is initialised as it is declared.

Elements within an array can be accessed with an index, starting with value 0. Therefore, for the first example above, **message1[0]** selects the first element and **message1[7]** the last. An access to **message1[8]** would be outside the boundary of the array, and would give invalid data. The index can be replaced by any variable which represents the required value.

The name of an array is set equal to the address of the initial element. Therefore, when an array name is passed in a function, what is passed is this address.

Strings

A string is a special array of type **char** that is ended by the NULL (\0) character. The null character allows code to search for the end of a string.

The size of the string array must therefore be one byte greater than the string itself, to contain this character.

For example a 20 character string could be declared:

```
char MyString[21];          // 20 characters plus null
```

Pointers (1)

Instead of specifying a variable by name, we can specify its address, called a *pointer.* A pointer can be loaded with the address of a variable by using the unary operator '&', like this:

```
my_pointer = &fred;
```

This loads the variable **my_pointer** with the *address* of the variable **fred**; **my_pointer** is then said to *point* to **fred**.

Or, the value of the variable pointed to by a pointer can be specified by prefixing the pointer with the '*' operator. For example, ***my_pointer** can be read as 'the value pointed to by **my_pointer**'. The * operator, used in this way, is sometimes called the *dereferencing* or *indirection* operator. The indirect value of a pointer, for example ***my_pointer**, can be used in an expression just like any other variable.

A pointer is declared by the data type it points to. Thus,

```
int *my_pointer;
```

indicates that **my_pointer** points to a variable of type **int**.

Pointers (2)

We can also use pointers with arrays. If the following is defined:

```
int dataarray[]={3,4,6,2,8,9,1,4,6};  // define an array of arbitrary values
int *ptr;                             // define a pointer
ptr = &dataarray[0];                  // assign pointer to the address of
                                      // the first element of the data array
```

Given the previous declarations, the following statements will therefore be true:

```
*ptr == 3;                  // the first element of the array pointed to
*(ptr+1) == 4;              // the second element of the array pointed to
*(ptr+2) == 6;              // the third element of the array pointed to
```

Libraries, and the C standard library

Because C is a simple language, much of its functionality derives from standard functions and macros, available in the libraries accompanying any compiler.

A C library is a set of precompiled functions which can be linked in to the application. These may be supplied with a compiler, available in-company, or be public domain.

There is a *Standard Library,* defined in the C ANSI standard. There are a number of standard header files, used for different groups of functions within the standard library. For example the <math.h> header file is used for a range of mathematical functions (including all trigonometric functions), while <stdio.h> contains the standard input and output functions, including for example the **printf( )** function.

stdio.h: I/O functions:

getchar() returns the next character typed on the keyboard.
putchar() outputs a single character to the screen.
printf() as previously described
scanf() as previously described

math.h: Mathematics functions

acos() returns arc cosine of arg
asin() returns arc sine of arg
atan() returns arc tangent of arg
cos() returns cosine of arg
exp() returns natural logarithim e
fabs() returns absolute value of num
sqrt() returns square root of num

Example Standard Library Functions

Header files

All but the simplest of C programs are made up of more than one file. Generally, there are many files which are combined together in the process of compiling, for example original source file combining with library files. To aid this process, a key section of any library file is detached and created as a separate *header* file.

Header file names end in **.h**; the file typically includes declarations of constants and function prototypes, and links on to other library files.

The function definitions themselves stay in the associated **.c** or **.cpp** files. In order to use the features of the header file and the file(s) it invokes, it must be included within any program accessing it, using **#include**. We see **mbed.h** being included in every program in the book.

The Compilation Process

A simplified version of the compilation process is shown. First a *pre-processor* looks at a particular source file and implements any defined pre-processor directives. It also identifies and prepares the header files that will be used by the C/C++ program files, and prepares the output that will be sent to the compiler.

The compiler takes each pre-processed C/C++ file and compiles, to generate a number of *object files* for the program. It ensures that the source files do not contain any syntax errors and that the object and library files are formatted correctly for the linker.

The *linker* manages the allocation of memory and ensures that all object and library files are linked to each other correctly.

The linker may uncover and report programming faults associated with memory allocation and capacity.

The linker generates a single executable binary (**.bin**) file, which can be downloaded to the microprocessor memory.

