APPENDIX B

Some C Essentials

B.1 A Word About C

This appendix aims to summarize the main features of the C language as used in this book, though not of the language as a whole. It is intended to be just adequate for the purpose of supporting the book. With care and experimentation, it can be used as an adequate introduction to the main features of the language. If you are a C novice, however, it's well worth having another reference source available, particularly as you consider the more advanced features. Refs. [1] and [2] are both good.

As you progress through the book, you will find yourself jumping around within the material of this appendix. Don't feel you need to read it sequentially. Instead, read the different sections as they're referenced from the book.

B.2 Elements of a C Program

B.2.1 Keywords

C has a number of keywords whose use is defined. A programmer cannot use a keyword for any other purpose, for example, as a data name. Keywords are summarized in Tables B.1–B.3.

B.2.2 Program Features and Layout

Simply speaking, a C program is made up of the following:

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;
```

Table B.1: C keywords associated with data type and structure definition.

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

Table B.2: C keywords associated with program flow.

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 labeled 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

Table B.3: C keywords associated with data storage class.

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

declare a variable called **exchange_rate** as a floating-point number, and another variable called **new_value** as an integer. The data types are keywords seen in the preceding tables.

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 (see below) 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 optimize the program layout.

As an example, the program that the mbed compiler always starts up with, shown as Program Example 2.1, *could* be written as shown here. It wouldn't be easy to read, however. It's the semicolons at the end of each statement, and the brackets, which in reality define much of the program structure.

```
#include "mbed.h"
DigitalOut myled(LED1); int main() {while(1) {myled = 1; wait(0.2); myled = 0;
wait(0.2);}}
```

Comments

Two ways of commenting are used. One is to place the comment between the markers **/*** and ***/**. This is useful for a block of text information running over several lines.

Alternatively, when two forward slash symbols (//) are used, the compiler ignores any text which follows on that line only, which can then be used for comment.

For example:

```
/*A program which flashes mbed LED1 on and off,
Demonstrating use of digital output and wait functions. */
#include "mbed.h"     //include the med 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.

B.2.3 Compiler Directives

Compiler directives are messages to the compiler and do not directly lead to program code. Compiler directives all start with a hash, `#`. Two examples follow.

#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. Angled brackets (<>) are used to enclose files held in a directory different from the current working directory, hence often for library files not written by the current author. Quotation marks are used to contain a file located within the current working directory, hence often user defined.

For example:

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

#define

The **#define** directive allows use of names for specific constants. For example, to use the number $\pi = 3.141592$ in the program, we could create a `#define` for the name "PI" and assign that number to it, as shown:

```
#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.

B.3 Variables and Data

B.3.1 Declaring, Naming, and Initializing

Variables must be named, and their data type defined, before they can be used in a program. Keywords from Table B.1 are used for this. For example:

```
int     MyVariable;
```

defines "MyVariable" as a data type int (integer).

It is possible to initialize the variable at the same time as declaration, for example:

```
int     MyVariable = 25;
```

initializes **MyVariable** and sets it to an initial value of 25.

It is possible to give variables meaningful names, while still avoiding excessive length, for example, "Height", "InputFile", "Area". Variable names must start with a letter or underscore; no other punctuation marks are allowed. Variable names are case sensitive.

B.3.2 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 of the link between data type, number range, and memory size are shown in Table B.4. It is interesting to compare these with information on number types given in , Appendix A. Note that the actual memory size applied to data types can vary between compilers. A full listing for the mbed compiler can be found in Ref. [3].

Table B.4: Example C data types, as implemented by the mbed compiler.

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 65535
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 $1.79769313486231571 \times 10^{+308}$

B.3.3 Working With Data

In C we can work with numbers in binary, fixed or floating-point decimal, or hexadecimal format, depending on what is most convenient, and what number type and range is required. For time critical applications it is important to remember that floating-point calculations can take much longer than fixed point. In general, it's easiest to work in decimal, but if a variable represents a register bit field or a port address, then it's usually more appropriate to manipulate the data in hexadecimal. When writing numbers in a program, the default radix (number base) for integers is decimal, with no leading 0 (zero). Octal numbers are identified with a leading 0. Hexadecimal numbers are prefixed with 0x.

For example, if a variable **MyVariable** is of type **char** we can perform the following examples to assign a number to that variable:

```
MyVariable = 15;      //a decimal example
MyVariable = 0x0E;    //a hexadecimal example
```

The value for both is the same.

B.3.4 Changing Data Type: Casting

Data can be changed from one data type to another by *type casting*. This is done using the cast operator, seen at the bottom of Table B.5. For example, in the line that follows, **size** has been declared as **char**, and **sum** as **int**. However, their division will not yield an integer. Therefore the result is cast to floating point.

```
mean=(float)sum/size;
```

Some type conversions may be done by the compiler implicitly. It is, however, better programming practice not to depend on this, but to do it explicitly.

B.4 Functions

A function is a section of code which can be called from another part of the program. So if a particular piece of code is to be used or duplicated many times, we can write it once as a function and then call that function whenever the specific operation is required. 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. Therefore, the function does not itself modify the value of the variable named. The impact of the function should thus be predictable and controlled.

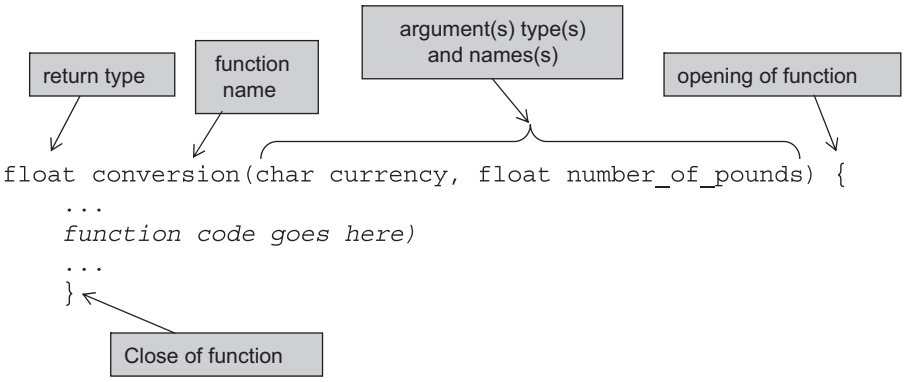

Figure B.1
Function example.

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

```
Return_type function_name (variable_type_1 variable_name_1, variable_type_2
                                                          variable_name_2,...)
```

An example is shown in Fig. B.1. The return type is given first. In this example, the keyword **float** is used. After the function name, in brackets, one or more data types may be listed, which identify the arguments which must be passed *to* the function. In this case, two arguments are sent, one of type **char** and one of type **float**. Following the function header, a pair of braces encloses the code which makes up the function itself. This could be anything from a single line to many pages. The final statement of the function may be a **return**, which will specify the value returned to the calling program. This is not essential if no return value is required.

B.4.1 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( )**. It must follow the structure just described. However, as **main( )** contains the central program, one expects to send nothing to it, nor receive anything from it. Therefore 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( )**.

B.4.2 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 run. 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.

B.4.3 Function Definitions

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);
```

The structure of this function is explained in Section B.6.2.

B.4.4 Using the static Storage Class With Functions

The static data type is useful for defining variables within functions, where the data inside the function must be remembered between function calls. For example, if a function within a real time system is used to calculate a digital filter output, the function should

always remember its previous data values. In this case, data values inside the function should be defined as static, for example, as shown below.

```
float movingaveragefilter(float data_in) {
  static float data_array[10];       // define static float data array
  for (int i=8;i>=0;i--) {
    data_array[i+1]=data_array[i];  // shift each data value along
  }                                  // (the oldest data value is discarded)
  data_array[0]=data_in;             // place new data at index 0
  float sum=0;
  for (int i=0;i<=9;i++) {
      sum=sum+data_array[i];         // calculate sum of data array
  }
  return sum/10;                     // return average value of array
}
```

B.5 Operators

C has a wide set of operators, shown in Table B.5. 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."

Operators have a certain order of precedence, shown in the table. The compiler applies this order when it evaluates a statement. If more than one operator at the same level of precedence occurs in a statement, then those operators are evaluated in turn, either left to right or right to left, as shown in the table. For example, the line

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

contains two operators. Table B.5 shows that the addition operator has precedence level 4, while all assign operators have precedence 14. The addition is therefore evaluated first, followed by the assign. In words we could say, the new value of the variable **counter** has been assigned the previous value of **counter**, plus one.

B.6 Flow Control: Conditional Branching

Flow Control is the title which covers the different forms of branching and looping available in C. As branching and looping can lead to programming errors, C provides clear structures to improve programming reliability.

B.6.1 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

Table B.5: C operators.

Precedence and Order	Operation	Symbol	Precedence and Order	Operation	Symbol
Parentheses 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
Logical Operators					
11, L to R	AND (1 if both X and Y are not 0)	X&&Y	2, R to L	NOT (1 if X=0)	!X
12, L to R	OR (1 if either X or Y are not 0)	X||Y			
Bitwise Operators					
8, L to R	Bitwise AND	X&Y	2, L to R	Ones complement (bitwise NOT)	~X
10, L to R	Bitwise OR	X|Y	5, L to R	Right shift. X is shifted right Y times	X>>Y
9, L to R	Bitwise XOR	X^Y	5, L to R	Left shift. X is shifted left Y times	X<<Y
Assignment Operators					
14, R to L	Assignment	X=Y	14, R to L	Bitwise AND assign	X&=Y
14, R to L	Add assign	X+=Y	14, R to L	Bitwise inclusive OR assign	X|=Y
14, R to L	Subtract assign	X−=Y	14, R to L	Bitwise exclusive OR assign	X^=Y
14, R to L	Multiply assign	X*=Y	14, R to L	Right shift assign	X>>=Y
14, R to L	Divide assign	X/=Y	14, R to L	Left shift assign	X<<=Y
14, R to L	Remainder assign	X%=Y			
Increment and Decrement Operators					
2, R to L	Preincrement	++X	2, R to L	Postincrement	X++
2, R to L	Predecrement	− −X	2, R to L	Postdecrement	X− −
Conditional Operators					
13, R to L	Evaluate *either* X (if Z≠0) *or* Y (if Z=0)	Z?X:Y	15, L to R	Evaluate X first, followed by Y	X,Y
"Data Interpretation" Operators					
2, R to L	The object or function pointed to by X	*X	2, R to L	The address of X	&X
2, R to L	Cast—the value of X, with (scalar) type specified	(*type*) X	2, R to L	The size of X, in bytes	sizeof X

not satisfied, then the code is not executed. There may or may not also be following **else** or **else if** statements.

Syntax:

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

The **if** and **else** statements are evaluated in sequence, i.e.,

> **else if** statements are only evaluated if all previous **if** and **else** conditions have failed; **else** statements are only executed if all previous conditions have failed.

For example, in the above example, the `else if (Condition2)` will only be executed if Condition1 has failed.

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;
}
```

B.6.2 Switch *Statements and Using* break

The **switch** statement allows a selection to be made of one out of several actions, based on the value of a variable or expression given in the statement. An example of this structure has already appeared, in the example function in Section B.4.3. The structure uses no less than four C keywords. Selection is made from a list of **case** statements, each with an associated label—note that a colon following a text word defines it as a label. If the label equals the **switch** expression then the action associated with that **case** is executed. The **default** action (which is optional) occurs if none of the **case** statements are satisfied. The **break** keyword, which terminates each **case** condition, can be used to exit from any loop. It causes program execution to continue after the **switch** code block.

B.7 Flow Control: Program Loops

B.7.1 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.

B.7.2 for Loops

The **for** loop allows a different form of looping, in that the dependent variable is updated automatically every time the loop is repeated. It defines an initialized 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.

B.7.3 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 of the following loops:

```
while(1) {
  … continuously called C statements here
}
```

Or

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

B.7.4 Exiting Loops With break

The **break** keyword can also be used to exit from a **for** or **while** loop, at any time within the loop. For example:

```
while(i>5) {
    ... C statements here
    if (fred == 1)
      break;
    ... C statements here
}                          //end of while
//execution continues here loop completion, or on break
```

B.8 Derived Data Types

In addition to the fundamental data types, there are further data types which can be derived from them. Example types that we use are described in this section.

B.8.1 Arrays and Strings

An array is a set of data elements, each of which has 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; it is recognized by the use of the square brackets which follow the name. The number of elements and their value can also be specified. For example, the declaration

```
unsigned 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 initialized 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 **message [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.

Importantly, 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.

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
```

B.8.2 Pointers

Instead of specifying a variable by name, we can specify its address. In C terminology, such an address is 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**.

Doing things the other way round, 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**.

We can also use pointers with arrays, because an array is really just a number of data values stored at consecutive memory locations. So 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
```

So array searching can be done by moving the pointer value to the correct array offset. Pointers are required for a number of reasons, but one simple reason is because the C standard does not allow us to pass arrays of data to and from functions, so we must use pointers instead to get around this.

B.8.3 Structures and Unions

Structures and *unions* are both sets of related variables, defined through the C keywords **struct** and **union**. In a way they are like arrays, but in both cases they can be of data elements of *different* types.

Structure elements, called *members*, are arranged sequentially, with the members occupying successive locations in memory. A structure is declared by invoking the **struct** keyword, followed by an optional name (called the structure *tag*), followed by a list of the structure members, each of these itself forming a declaration. For example:

```
struct resistor {int val; char pow; char tol;};
```

declares a structure with tag **resistor**, which holds the value (**val**), power rating (**pow**), and tolerance (**tol**) of a resistor. The tag may come before or after the braces holding the list of structure members.

Structure elements are identified by specifying the name of the variable and the name of the member, separated by a full stop (period). Therefore, **resistor.val** identifies the first member of the example structure above.

Like a structure, a union can hold different types of data. Unlike the structure, union elements all begin at the same address. Hence the union can represent only one of its members at any one time, and the size of the union is the size of the largest element. It is up to the programmer to track which type is currently stored! Unions are declared in a format similar to that of structures.

Unions, structures, and arrays can occur within each other.

B.9 C Libraries and Standard Functions

B.9.1 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 files combining with standard 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. 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 almost every program in the book. Note also that the **.c** file where the function declarations appear must also include the header file.

B.9.2 Libraries and the C Standard Library

Because C is a simple language, much of its functionality derives from standard functions and macros which are available in the libraries accompanying any compiler. A C library is a set of precompiled functions which can be linked into the application. These may be supplied with a compiler, available in-company, or be public domain. Notably 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.

B.9.3 Using printf

This versatile function provides formatted output, typically for sending display data to a PC screen. Text, data, formatting, and control formatting can be specified. Only summary information is provided here, a full statement can be found in [1]. Examples below are taken from chapters in this book. In each case the function appears in the form **pc.printf( )**, indicating that **printf( )** is being used as a member function of a C++ class **pc** created in the example program. This doesn't affect the format applied.

Simple text messages

```
pc.printf("ADC Data Values...\n\r");   \\send an opening text message
```

This prints the text string "ADC Data Values…" to screen and uses control characters \n and \r to force a new line and carriage return respectively.

Data messages

```
pc.printf("%1.3f",ADCdata);
```

This prints the value of the **float** variable **ADCdata.** A *conversion specifier*, initiated by the % character, defines the format. Within this the "f" specifies floating point, and the .3 causes output to three decimal places.

```
pc.printf("%1.3f \n\r",ADCdata);   \\send the data to the terminal
```

As above, but includes \n and \r to force a new line and carriage return.

Combination of text and data

```
pc.printf("random number is %i\n\r", r_delay);
```

This prints a text message, followed by the value of **int** variable (indicated by the "i" specifier) **r_delay**.

```
pc.printf("Time taken was %f seconds\n", t.read());  //print timed value to pc
```

This prints a text message, followed by the return value of function **t.read( ),** which is of type **float.**

B.10 File Access Operations

B.10.1 Overview

In C we can open files, read and write data, and also scan through files to specific locations, even searching for particular types of data. The commands for input and output operations are all defined by the C stdio library, already mentioned in connection with **printf( )**.

The stdio library uses the concept of *streams* to manage the data flow, where a stream is a sequence of bytes. All streams have similar properties, even though the actual implementation of their use is varied, depending on things like the source and destination of their flow. Streams are represented in the stdio library as pointers to FILE objects, which uniquely identify the stream. We can store data in files (as chars) or we can store words and strings (as character arrays). A summary of useful stdio file access library functions is given in Table B.6.

B.10.2 Opening and Closing Files

A file can be opened with the following command:

```
FILE* pFile = fopen("datafile.txt","w");
```

This assigns a pointer with name **pFile** to the file at the specific location given in the **fopen** statement. In this example the *access mode* is specified with a "w", and a number of other file open access modes, and their specific meanings, are shown in Table B.7. When "w" is the access mode (meaning *write access*), if the file doesn't already exist, then the **fopen** command will automatically create it in the specified location.

When we have finished using a file for reading or writing it should be closed, for example, with

```
fclose(pFile);
```

Table B.6: Useful stdio library functions.

Function	Format	Summary Action
fclose	`int fclose ( FILE * stream );`	Closes a file
fgetc	`int fgetc ( FILE * stream );`	Gets a character from a stream
fgets	`char * fgets ( char * str, int num, FILE * stream );`	Gets a string from a stream
fopen	`FILE * fopen ( const char * filename, const char * mode);`	Opens the file of type FILE and name filename
fprintf	`int fprintf ( FILE * stream, const char * format, ... );`	Writes formatted data to a file
fputc	`int fputc ( int character, FILE * stream );`	Writes a character to a stream
fputs	`int fputs ( const char * str, FILE * stream );`	Writes a string to a stream
fseek	`int fseek ( FILE * stream, long int offset, int origin );`	Moves the file pointer to the specified location

str: An array containing the null-terminated sequence of characters to be written.
stream: Pointer to a FILE object that identifies the stream where the string is to be written.

Table B.7: Access modes for fopen.

Access Mode	Action
"r"	Open an existing file for reading.
"w"	Create a new empty file for writing. If a file of the same name already exists, it will be deleted and replaced with a blank file.
"a"	Append to a file. Write operations result in data being appended to the end of the file. If the file does not exist, a new blank file will be created.
"r+"	Open an existing file for both reading and writing.
"w+"	Create a new empty file for both reading and writing. If a file of the same name already exists, it will be deleted and replaced with a blank file.
"a+"	Open a file for appending or reading. Write operations result in data being appended to the end of the file. If the file does not exist, a new blank file will be created.

B.10.3 Writing and Reading File Data

If the intention is to store numerical data, this can be done in a simple way by storing individual 8-bit data values. The **fputc** command allows this, as follows:

```
char write_var=0x0F;
fputc(write_var, pFile);
```

This stores the 8-bit variable **write_var** to the data file. The data can also be read from a file to a variable as follows:

```
read_var = fgetc(pFile);
```

Table B.8: fseek origin values.

Origin Value	Description
SEEK_SET	Beginning of file
SEEK_CUR	Current position of the file pointer
SEEK_END	End of file

Using the stdio.h commands, it is also possible to read and write words and strings with **fgets( ), fputs( )** and to write formatted data with **fprintf( )**.

It is also possible to search or move through files looking for particular data elements. When reading data from a file, the file pointer can be moved with the fseek command. For example, the following command will reposition the file pointer to the eighth byte in the text file:

```
fseek (pFile , 8 , SEEK_SET );  // move file pointer to 8 bytes from the start
```

The first term of the **fseek( )** function is the stream (the file pointer name), the second term is the pointer offset value, and the third term is the origin where the offset should be applied. There are three possible values for the origin as shown in Table B.8. The origin values have predefined names as shown.

Further details on the stdio commands and syntax can also be found in Refs. [1] and [2].

B.11 Toward Professional Practice

The readability of a C program is much enhanced by good layout on the page or screen. This helps to produce good and error-free code, and enhances the ability to understand, maintain, and upgrade it. Many companies impose a "house style" on C code written for them; such guides can be found online.

The very simple style guide we have adopted in this book should be exemplified in any of the programs reproduced. It includes the following:

- Courier New font applied,
- opening header text block, giving overview of program action,
- blank lines used to separate major code sections,
- extensive commenting,
- opening brace of any code block placed on line which initiates it,
- code within any code block indented two or four spaces compared with code immediately outside block,
- closing brace stands alone, indented to align with line initiating code block.

A useful set of guidelines which relate to the development of the mbed SDK is given in [4].

Version control is essential practice in any professional environment. For clarity, we have not displayed version control information within the programs as reproduced in the book, although this was done in the development process. Minimal version control within the source code (placed within the header text block) would include date of origin, name of original author, name of person making most recent revision, and date.

References

[1] P. Prinz, U. Kirch-Prinz, C Pocket Reference, O'Reilly, 2003, ISBN 0-596-00436-2.
[2] The C++ Resources Network, www.cplusplus.com.
[3] RealView® Compilation Tools. Version 4.0. Compiler Reference Guide. ARM (December 2010). DUI 0348C (ID101213).
[4] mbed SDK Coding Style, https://developer.mbed.org/teams/SDK-Development/wiki/mbed-sdk-coding-style.