CHAPTER 6

Further Programming Techniques

6.1 The Benefits of Considered Program Design and Structure

There are a number of challenges when tackling an embedded system design project. It is usually wise to first consider the software design structure, particularly with large projects and designs that have many modes of operation. It is often not possible to program all functionality into a single control loop, so the approach for structuring code and breaking it up into understandable features should be well thought out. In particular, it helps if the following can be achieved:

- That code is readable, structured, and well documented.
- That code can be tested for performance in a modular form.
- That development reuses existing code utilities to keep development time short.
- That code design supports multiple engineers working on a single project.
- That future upgrades to code can be implemented efficiently.

There are a number of C/C++ programming techniques which enable these design requirements to be considered, as discussed in this chapter.

6.2 Functions

In the C/C++ language, a *function* is a portion of code within a larger program. The function performs a specific task and is relatively independent of the main code. Functions can be used to manipulate data; this is particularly useful if a number of similar data manipulations are required in the program. We can input data values to the function and the function can return a result to the main program. Functions are particularly useful for coding mathematical algorithms, lookup tables, data conversions, and control features. It is also possible to use functions with no input or output data, simply to reduce code size and to improve readability of code. Fig. 6.1 illustrates a function call.

There are a number of advantages when using functions. Firstly, a function is written once and compiled into one area of memory, irrespective of the number of times that it is

Fast and Effective Embedded Systems Design. http://dx.doi.org/10.1016/B978-0-08-100880-5.00006-2

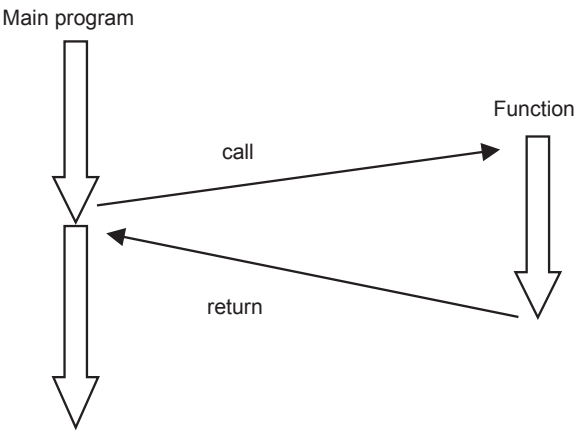

Figure 6.1
A function call.

called from the main program, so demands on program memory are reduced. Functions also enable clean and manageable code to be designed, allowing software to be well structured and readable. The use of functions also enables the practice of *modular coding*, especially useful when teams of software engineers are required to develop large and advanced applications. Writing code with functions allows one engineer to develop a particular software feature, while another engineer may take responsibility for something else.

Using functions is not always completely beneficial however. There is a small execution time overhead in storing program position data and jumping and returning from the function, but this should only be an issue for consideration in the most time critical systems. Furthermore, it is possible to "nest" functions within functions, which can sometimes make software challenging to follow. A limitation of C functions is that only a single value can be returned from the function, and arrays of data cannot be passed to or from a function (only single value variables can be used). Working with functions and modular techniques therefore requires a considered software structure to be designed and evaluated before programming is started.

6.3 Program Design

6.3.1 Using Flow Charts to Define Code Structure

It is often useful to use a *flowchart* (also called a *flow diagram*) to indicate the operation of program flow and the use of functions. We can design code flow using a flowchart prior to coding. Fig. 6.2 shows some of the flowchart symbols that are used.

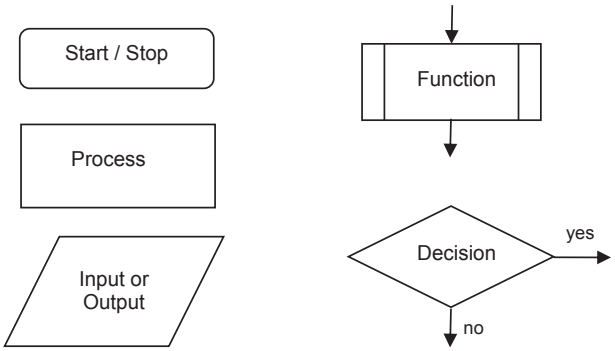

Figure 6.2
Example flow chart symbols.

For example, take the following software design specification:

Design a program to increment continuously the output of a seven-segment numerical LED display, as shown in Fig. 6.3. This should step through the numbers 0—9, then reset back to 0 to continue counting. This includes the following:

- Use a function to convert a hexadecimal counter byte A to the relevant seven-segment LED output byte B.
- Output the LED output byte to light the correct segment LEDs.
- If the count value is greater than 9, then reset to 0.
- Delay for 500 ms to ensure that the LED output counts up at a rate that is easily visible.

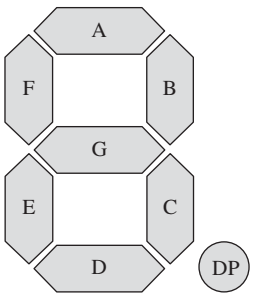

Figure 6.3
Seven-segment display.

The control of the seven-segment display has already been discussed in Section 3.6.3. Drawing on this, a feasible software design is shown in Fig. 6.4.

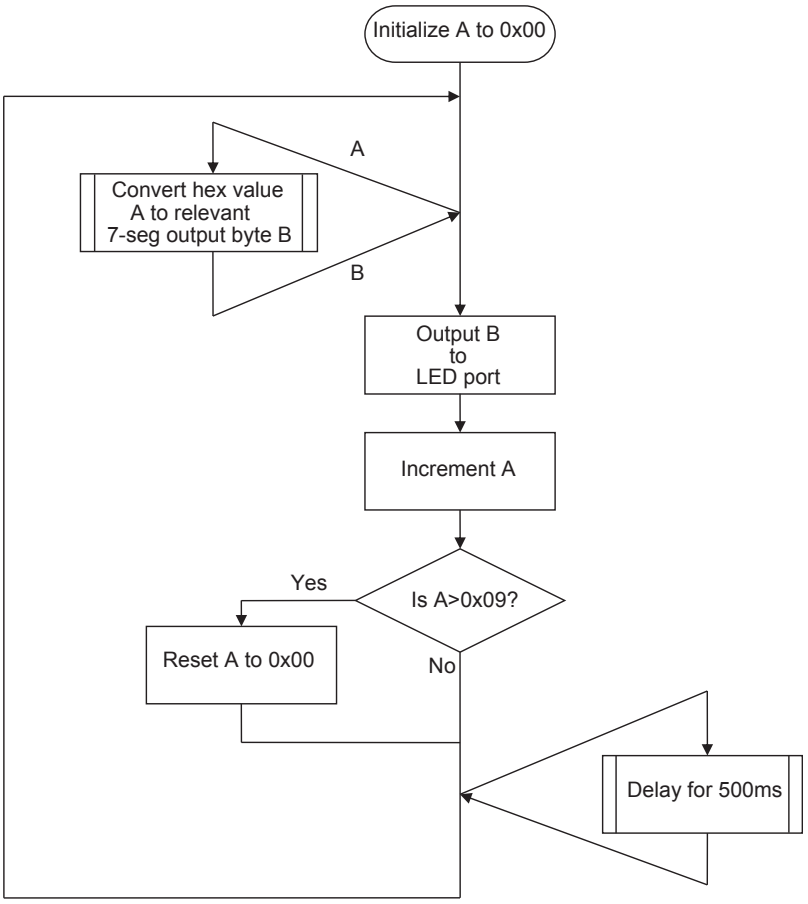

Figure 6.4
Example flowchart design for a seven-segment display counter.

Flow charts allow us to visualize the order of operations of code and to make judgments on what sections of a program may require the most attention or take the most effort to develop. They also help with communicating a potential design with nonengineers, which may hold the key to designing a system which meets a very detailed specification.

6.3.2 Pseudocode

Pseudocode consists of short, English phrases used to explain specific tasks within a program. Ideally, pseudocode should not include keywords in any specific computer language. Pseudocode should be written as a list of consecutive phrases, we can even draw arrows to show looping processes. Indentation can be used to show the logical program flow in pseudocode also.

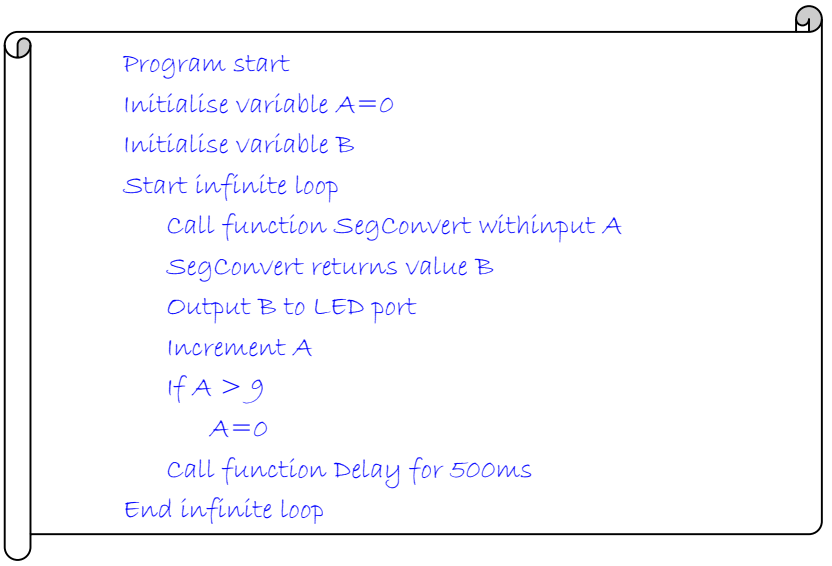

Figure 6.5
Example pseudocode for seven-segment display counter.

Writing pseudocode saves time later during the coding and testing stage of a program's development and also helps communication between designers, coders, and project managers. Some projects may use pseudocode for design, others may use flow charts, and some a combination of both.

The software design shown by the flowchart in Fig. 6.4 could also be described in pseudocode as shown in Fig. 6.5.

Note that the functions **SegConvert( )** and **Delay( )** are defined elsewhere, for example, in a separate "utilities" file, authored by a different engineer. Function **SegConvert( )** could implement a simple lookup table or number of **if** statements that assigns the suitable value to B.

6.4 Working With Functions on the mbed

C code feature

The implementation and syntax of functions in C/C++ are reviewed in Section B4 of Appendix B with some simple examples. It is important to remember that, as with variables, all functions (except the **main( )** function) must be declared at the start of a program. The declaration statements for functions are called *prototypes*. So each function in the code must have an associated prototype for it to compile and run. Many data values can be passed to a function, but only one data value can be returned. Such data elements, called *arguments*, must be of a defined data type and be declared in the prototype. The function prototype takes the form

```
return_type function_name (var1_type var1_name, var2_type var2_name,...)
```

The data type of the return value is given first, followed by the function name. Then, in brackets, input data types may be listed, along with the argument names. It is possible to have functions with no input or output arguments, in which case the term **void** should be used in place of the argument data type.

The actual function code needs defining within a C/C++ file also, in order for it to be called from within the main code. This is done by specifying the function in the same way as the prototype, followed by the actual function code. We will see a number of examples during this chapter, so in each case make a point of identifying the function prototype and the actual function definition. A further point to note is that if a function is defined in code prior to the **main( )** C function, then its definition serves as its prototype also.

6.4.1 Implementing a Seven-Segment Display Counter

Program Example 6.1 shows a program which implements the designs described by the flowchart in Fig. 6.4 and the pseudocode shown in Fig. 6.5. It applies some of the seven-segment display techniques first used in Program Example 3.7, but goes beyond these. The main design requirement is that a seven-segment display is used to count continuously from 0 to 9 and loop back to 0. Declarations for the **BusOut** object and the A and B variables, as well as the **SegConvert( )** function prototype, appear early in the program. It can be seen that the **main( )** program function is followed by the **SegConvert( )** function, which is called regularly from within the main code. Notice in the line

```
B=SegConvert(A);                // Call function to return B
```

that **B** immediately takes on the return value of the **SegConvert( )** function. The return value is a variable that is used, calculated, or deduced within the function and returned to the main program that called the function.

Notice in the **SegConvert( )** function the final line immediately below, which applies the **return** keyword:

```
return SegByte;
```

This line causes program execution to return to the point from which the function was called, carrying the value **SegByte** as its return value. It's an important technique to use once you start writing functions which provide return values. We notice of course that **SegByte** has been declared as a **char** data type within the function, and this data type matches that expected and defined within the function prototype, which is seen early in the program listing.

```
/* Program Example 6.1: seven-segment display counter
                                                              */
#include "mbed.h"
BusOut Seg1(p5,p6,p7,p8,p9,p10,p11,p12); // A,B,C,D,E,F,G,DP
char SegConvert(char SegValue);          // function prototype
char A=0;                                // declare variables A and B
char B;

int main() {                             // main program
  while (1) {                            // infinite loop
    B=SegConvert(A);                     // Call function to return B
    Seg1=B;                              // Output B
    A++;                                 // increment A
    if (A>0x09){                         // if A > 9 reset to zero
       A=0;
    }
    wait(0.5);                           // delay 500 milliseconds
  }
}

char SegConvert(char SegValue) {         // function 'SegConvert'
  char SegByte=0x00;
  switch (SegValue) {                    //DP G F E D C B A
    case 0 : SegByte = 0x3F;break;       // 0 0 1 1 1 1 1 1 binary
    case 1 : SegByte = 0x06;break;       // 0 0 0 0 0 1 1 0 binary
    case 2 : SegByte = 0x5B;break;       // 0 1 0 1 1 0 1 1 binary
    case 3 : SegByte = 0x4F;break;       // 0 1 0 0 1 1 1 1 binary
    case 4 : SegByte = 0x66;break;       // 0 1 1 0 0 1 1 0 binary
    case 5 : SegByte = 0x6D;break;       // 0 1 1 0 1 1 0 1 binary
    case 6 : SegByte = 0x7D;break;       // 0 1 1 1 1 1 0 1 binary
    case 7 : SegByte = 0x07;break;       // 0 0 0 0 0 1 1 1 binary
    case 8 : SegByte = 0x7F;break;       // 0 1 1 1 1 1 1 1 binary
    case 9 : SegByte = 0x6F;break;       // 0 1 1 0 1 1 1 1 binary
  }
    return SegByte;
}
```

Program Example 6.1: Seven-segment display counter

The function **SegConvert( )** in Program Example 6.1 is essentially a *lookup table*, i.e., a table of values that converts one value directly to another value. In this example the input value 0 is converted to an output value 0x3F, while an input value of 6 is converted to an output value of 0x7D. This lookup table has a direct mapping between input and output values, as described by the **switch( )** statement. Lookup tables can, however, be large arrays of data representing mathematical functions (such as logarithms and trigonometric values) and can include calculations for interpolating and extrapolating between and beyond data points.

Connect a seven-segment display to the mbed and implement Program 6.1. Apply the wiring diagram in Fig. 3.12 of Chapter 3. Verify that the display output continuously

counts from 0 to 9 and then resets back to 0. Ensure that you understand how the program works by cross referencing with the flowchart and pseudocode designs shown previously.

Exercise 6.1

Change Program Example 6.1 so that the display counts up in hexadecimal from 0 to F. You will need to work out the display patterns for A to F using the seven-segment display. For a few, you will need lower case, for others, upper case.

■

6.4.2 Function Reuse

Now that we have a function to convert a decimal value to a seven-segment display byte, we can build projects using multiple seven-segment displays with little extra effort. For example, we can implement a second seven-segment display (see Fig. 6.6) by simply defining its mbed **BusOut** declaration and calling the same **SegConvert( )** function as before.

It is possible to implement a counter program which counts from 00 to 99 by simply modifying the main program code to that shown in Program Example 6.2. Note that the **SegConvert( )** function previously defined in Program Example 6.1 is also required to be

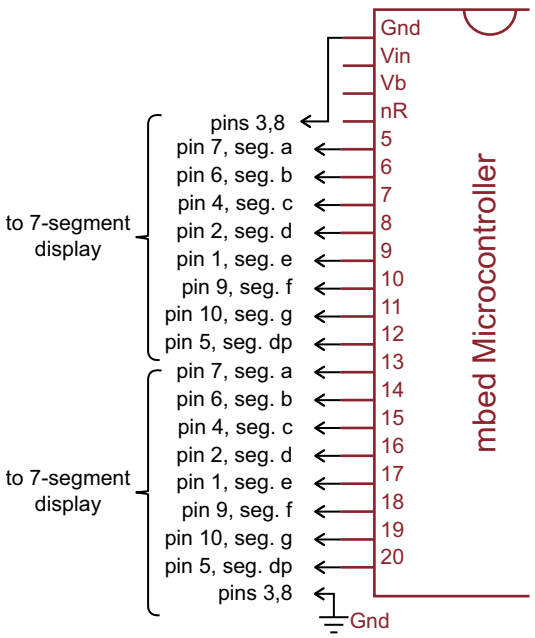

Figure 6.6
Two seven-segment display control with the mbed.

copied (reused) in this example. Note also that a slightly different programming approach is used; here we use two **for** loops to count each of the tens and units values.

```
/* Program Example 6.2: Display counter for 0-99
                                                                    */
#include "mbed.h"
BusOut Seg1(p5,p6,p7,p8,p9,p10,p11,p12); // A,B,C,D,E,F,G,DP
BusOut Seg2(p13,p14,p15,p16,p17,p18,p19,p20);

char SegConvert(char SegValue);     // function prototype
int main() {                        // main program
  while (1) {                       // infinite loop
    for (char j=0;j<10;j++) {       // counter loop 1
      Seg2=SegConvert(j);           // tens column
      for (char i=0;i<10;i++) {     // counter loop 2
          Seg1=SegConvert(i);       // units column
          wait(0.2);
      }
    }
  }
}
// add SegConvert function here...
```

Program Example 6.2: Two digit seven-segment display counter.

Using two seven-segment displays, with pin connections shown in Fig. 6.6, implement Program Example 6.2 and verify that the display output counts continuously from 00 to 99 and then resets back to 0. Review the program design and familiarize yourself with the method used to count the tens and units digits each from 0 to 9.

Exercise 6.2

Write and test mbed programs to perform the following action with a dual seven-segment display setup:

1. Count 0 to FF, in hexadecimal format
2. Create a 1-minute timer—count 0—59, with a precise increment rate of 1 s. Flash an LED every time the count overflows back to zero (i.e., every minute)
3. Introduce two LEDs to make a simple "1" digit, and count 0 to 199. This is called a two and a half digit display.

In each case, set the count rate to a different speed and ensure that the program loops back to 0x00, so that the counter operates continuously.

■

6.4.3 A More Complex Program Using Functions

A more advanced program could read two numerical values from a host terminal application and display these on two seven-segment displays connected to the mbed. The

program can therefore display any integer number between 00 and 99, as required by user key presses.

The example program below uses four functions to implement the host terminal output on seven-segment displays. The four functions are as follows:

- **SegInit( )**—to set up and initialize the seven-segment displays
- **HostInit( )**—to set up and initialize the host terminal communication
- **GetKeyInput( )**—to get keyboard data from the terminal application
- **SegConvert( )**—to convert a decimal integer to a seven-segment display data byte

We will use the mbed serial USB interface to communicate with the host PC, as we did in Section 5.3, and two seven-segment displays, as in the previous exercise.

For the first time now we come across a method for communicating keyboard data and display characters; using *ASCII* codes. The term ASCII refers to the American Standard Code for Information Interchange method for defining alphanumeric characters as 8-bit values. Each alphabet character (lower and upper case), number (0 to 9), and a selection of punctuation characters are all described by a unique identification byte, i.e., "the ASCII value." This coding is widely used; for example, when a key is pressed on a computer keyboard, its ASCII byte is communicated to the PC. The same applies when communicating with displays. We will develop the use of ASCII characters further in Chapter 8.

The ASCII byte for numerical characters has the higher four bits set to value 0x3 and the lower four bits represent the value of the numerical key which is pressed (0x0 to 0x9). Numbers 0—9 are therefore represented in ASCII as 0x30 to 0x39. To convert the ASCII byte returned by the keyboard to a regular decimal digit, the higher four bits need to be removed. We do this by logically ANDing the ASCII code with a bitmask, a number with bits set to 1 where we want to keep a bit in the ASCII, and set to 0 where we want to force the bit to 0. In this case we apply a bitmask of 0x0F. The logical AND applies the operator **&** from Table B.5 and appears in the line

```
return (c&0x0F);         // apply bit mask to convert to decimal, and return
```

Example functions and program code are shown in Program Example 6.3. Once again, the function **SegConvert( )**, as shown in Program Example 6.1, should be added to complete the program.

```
/* Program Example 6.3: Host keypress to 7-seg display
                                                                */
#include "mbed.h"
Serial pc(USBTX, USBRX);                        // comms to host PC
BusOut Seg1(p5,p6,p7,p8,p9,p10,p11,p12);        // A,B,C,D,E,F,G,DP
BusOut Seg2(p13,p14,p15,p16,p17,p18,p19,p20);   // A,B,C,D,E,F,G,DP

void SegInit(void);                          // function prototype
void HostInit(void);                         // function prototype
char GetKeyInput(void);                      // function prototype
```

```
char SegConvert(char SegValue);             // function prototype
char data1, data2;                          // variable declarations

int main() {         // main program
  SegInit();         // call function to initialise the 7-seg displays
  HostInit();        // call function to initialise the host terminal
  while (1) {                 // infinite loop
    data2 = GetKeyInput();    // call function to get 1st key press
    Seg2=SegConvert(data2);   // call function to convert and output
    data1 = GetKeyInput();    // call function to get 2nd key press
    Seg1=SegConvert(data1);   // call function to convert and output
    pc.printf(" ");           // display spaces between numbers
  }
}
// functions
void SegInit(void) {
  Seg1=SegConvert(0);         // initialise to zero
  Seg2=SegConvert(0);         // initialise to zero
}

void HostInit(void) {
  pc.printf("\n\rType two digit numbers to be displayed\n\r");
}

char GetKeyInput(void) {
  char c = pc.getc();      // get keyboard data (ascii 0x30-0x39)
  pc.printf("%c",c);       // print ascii value to host PC terminal
  return (c&0x0F);         // apply bit mask to convert to decimal, and return
}
// copy SegConvert function here too...
```

Program Example 6.3: Two digit seven-segment display based on host key presses

Implement Program Example 6.3 and verify that numerical keyboard presses are displayed on the seven-segment displays. Familiarize yourself with the program design and understand the input and output features of each program function.

6.5 Using Multiple Files in C/C++

Large embedded projects in C/C++ benefit from being split into a number of different files, usually, so that a number of engineers can take responsibility for different parts of the code. This approach also improves readability and maintenance. For example, the code for a processor in a vending machine might have one C/C++ file for the control of the actuators delivering the items and a different file for controlling the user input and LCD display. It doesn't make sense to combine these two code features in the same source file as they each relate to different peripheral hardware. Furthermore, if a new batch of vending machines are to be built with an updated keypad and display, only that piece of the code needs to be modified. All the other source files can be carried over without change.

Modular coding uses *header files* to join multiple files together. In general, we use a main C/C++ file (**main.c** or **main.cpp**) to contain our high level code, but all functions and *global variables* (variables that are available to all functions) are defined in feature-specific C files. It is good practice therefore for each C/C++ feature file to have an associated header file (with a **.h** extension). Header files typically include declarations only, for example, compiler directives, variable declarations, and function prototypes.

Section B9.2 describes the C standard library, which contain a very wide range of functions available to extend the capability of C/C++. These are linked via a number of header files, which can be used for more advanced data manipulation or arithmetic. For example, Program Example 4.3 uses the standard **sin( )** function, linked through the **math.h** header file; Program Example 5.4 uses the standard **printf( )** function, linked through **stdio.h.** Both **math.h** and **stdio.h** are invoked automatically by **mbed.h**. Links to other standard library C/C++ header files are made in the **mbed.h** header, they are also discussed in Section B9 of Appendix B.

6.5.1 Summary of the C/C++ Program Compilation Process

To further understand the design approach to modular coding, it helps to understand the way programs are preprocessed, compiled, and linked to create a binary execution file for the microprocessor. A simplified version of this process is shown in Fig. 6.7 and described in detail with an example in Section 6.6.

In summary, first a *preprocessor* looks at a particular source file and implements any defined preprocessor rules (known as *preprocessor directives*). The preprocessor also identifies and prepares the header files that will be utilized by the C++ program files. The compiler then combines each header and C/C++ file to generate a number of *object files* for the program. In doing so the compiler ensures that the source files do not contain any syntax errors and that the object and library files are formatted correctly for the linker. Note, of course, that a program can have no syntax errors, but still be quite useless.

The *linker* manages the allocation of memory for the microprocessor application and ensures that all object and library files are linked to each other correctly. The linker generates a single executable binary (**.bin**) file, which can be downloaded to the microprocessor. In undertaking the task, the linker may uncover programming faults associated with memory allocation and capacity.

6.5.2 Using #define, #include, #ifndef, and #endif Directives

C code feature

As mentioned previously, the C/C++ preprocessor prepares code before the program is compiled. Preprocessor directives are denoted with a **#** symbol, as described in Section B2 of Appendix B. These may also be called *compiler directives* as the preprocessor is essentially a subprocess of the compiler.

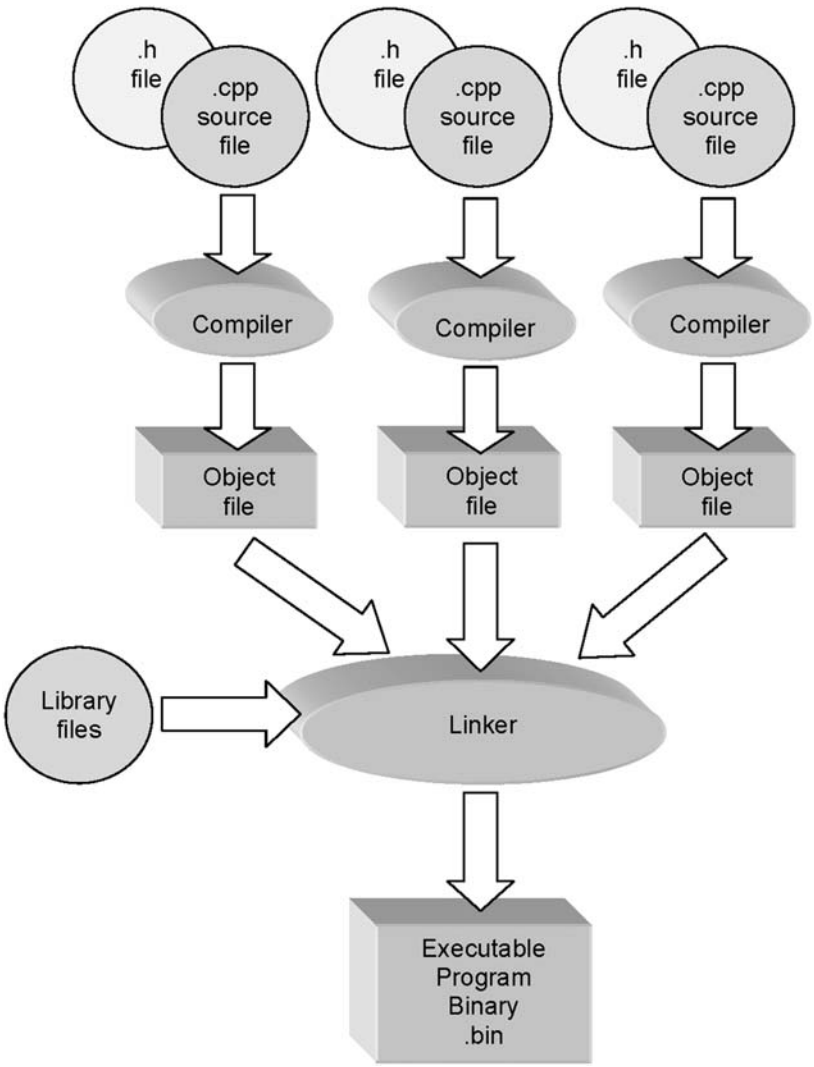

Figure 6.7
C program compile and link process.

A simple preprocessor directive is **#define** (usually referred to as "hash-define"), which allows us to use meaningful names for specific constants. Here are some examples

```
#define    SAMPLEFREQUENCY  44100
#define    PI               3.141592
#define    MAX_SIZE         255
```

The preprocessor replaces **#define** with the actual value associated with that name, so using **#define** statements doesn't actually increase the memory size of the program or the load on the microprocessor.

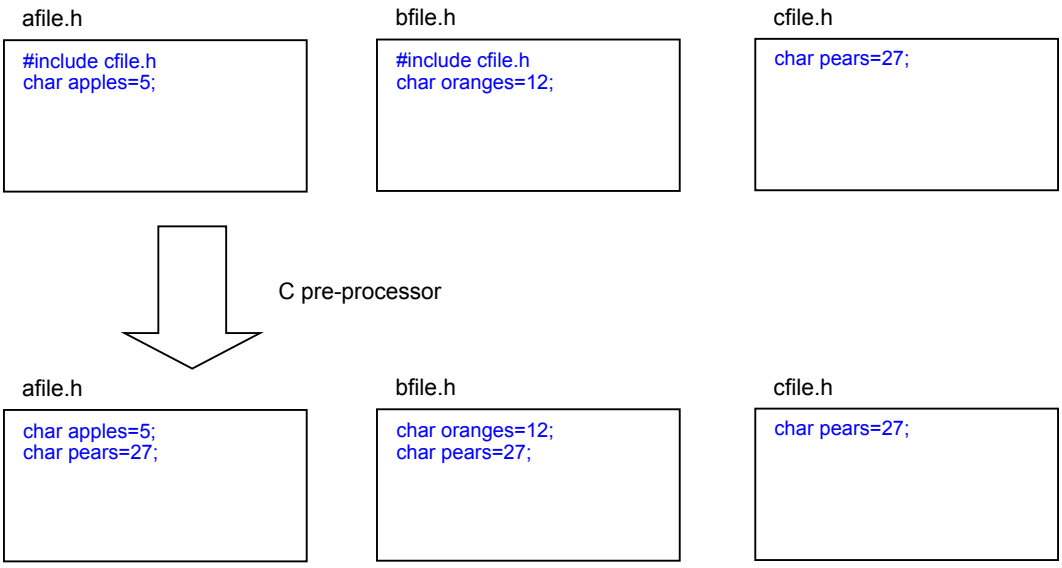

Figure 6.8
C preprocessor example with multiple declaration error for variable "pears".

The **#include** (usually referred to as "hash-include") directive is commonly used to tell the preprocessor to include any code or statements contained within an external header file. Indeed we have seen this ubiquitous **#include** statement in every complete program example so far in this book, as this is used to connect our programs with the core mbed libraries.

Fig. 6.8 shows three header files to explain how the **#include** statement works and highlights a common issue with using header files. It is important to know that **#include** essentially just acts as a cut and paste feature. Therefore, if we compile **afile.h** and **bfile.h** where both files also **#include cfile.h**, we will have two copies of the contents of **cfile.h** (hence variable **pears** will be defined twice). The compiler will thus highlight an error, as multiple declarations of the same variables or function prototypes are not allowed.

C code feature

Fig. 6.8 highlights a problem with using **#include** statements to link header files, in that it is common for the preprocessor to attempt to declare a single variable, function, or object more than once, which is not allowed by the compiler. The **#ifndef** directive, which means "if not defined," can be used to provide a solution to the problem. When using header files it is possible (and indeed good practice) to use a conditional statement to define variables and function prototypes only if they have not previously been defined. The **#ifndef** directive provides a solution as it allows a conditional statement based on the existence of a **#define** value. If the **#define** value has not previously been defined then that value and all of the header file's variables and

prototypes are defined. If the **#define** value has previously been declared then the header file's contents are not implemented by the preprocessor (as they must certainly have already been implemented). This ensures that all header file declarations are only added once to the project. The example code shown in Program Example 6.4 represents a template header file structure using the **#ifndef** condition, avoiding the error highlighted in Fig. 6.8.

```
/* Program Example 6.4: Template for .h header file
                                                                */
#ifndef VARIABLE_H    // if VARIABLE_H has not previously been defined
#define VARIABLE_H    // define it now
// header declarations here...
#endif                // end of the if directive
```

Program Example 6.4: Example header file template

Note in this example, we don't actually give **VARIABLE_H** a value. This is ok; we can declare "**#defines**" without actually needing to assign a value to them, much in the same way as C/C++ variables can be defined before they are given a value. The **#endif** directive is used to indicate the end of the **#ifndef** conditional, as seen in Program Example 6.4.

6.5.3 Using mbed Objects Globally

All mbed objects must be defined in an "owner" source file. However, we may wish to use those objects from within other source files in the project, i.e., "globally." This can be done by also defining the mbed object in the associated owner's header file. When an mbed object is defined for global use, the **extern** specifier should be used. For example, a file called **my_functions.cpp** may define and use a **DigitalOut** object called "**RedLed**" as follows:

```
DigitalOut RedLed(p5);
```

If any other source files need to manipulate **RedLed**, the object must also be declared in the **functions.h** header file using the **extern** specifier, as follows:

```
extern DigitalOut RedLed;
```

Note that the specific mbed pins don't need to be redefined in the header file, as these will have already been specified in the object declaration in **my_functions.cpp**.

6.6 Modular Program Example

A modular program example can now be built from the nonmodular code given in Program Example 6.3. Here we separate the functional features to different source and

header files. We therefore create a keyboard-controlled seven-segment display project with multiple source files as follows:

- **main.cpp**—contains the main program function
- **HostIO.cpp**—contains functions and objects for host terminal control
- **SegDisplay.cpp**—contains functions and objects for seven-segment display output

The following associated header files are also required:

- **HostIO.h**
- **SegDisplay.h**

The program file structure in the mbed compiler should be similar to that shown in Fig. 6.9. Note that modular files can be created by right clicking on the project name and selecting "New File".

The **main.cpp** file holds the same main function code as before, but with **#include** directives to the new header files. Program Example 6.5 details the source code for **main.cpp**.

```
/* Program Example 6.5: main.cpp file for modular 7-seg keyboard controller
                                                                          */
#include "mbed.h"
#include "HostIO.h"
#include "SegDisplay.h"
char data1, data2;                    // variable declarations
int main() {                          // main program
  SegInit();                          // call init function
  HostInit();                         // call init function
  while (1) {                         // infinite loop
    data2 = GetKeyInput();            // call to get 1st key press
    Seg2 = SegConvert(data2);         // call to convert and output
    data1 = GetKeyInput();            // call to get 2nd key press
    Seg1 = SegConvert(data1);         // call to convert and output
    pc.printf(" ");                   // display spaces on host
  }
}
```

Program Example 6.5: Source code for main.cpp

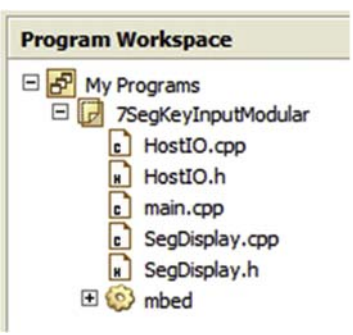

Figure 6.9
File structure for modular seven-segment display program.

The **SegInit( )** and **SegConvert( )** functions are to be "owned" by the **SegDisplay.cpp** source file, as are the **BusOut** objects named **Seg1** and **Seg2**. The resulting **SegDisplay.cpp** file is shown in Program Example 6.6.

```
/* Program Example 6.6: SegDisplay.cpp file for modular 7-seg keyboard
controller
                                                                    */
#include "SegDisplay.h"
BusOut Seg1(p5,p6,p7,p8,p9,p10,p11,p12);      // A,B,C,D,E,F,G,DP
BusOut Seg2(p13,p14,p15,p16,p17,p18,p19,p20); // A,B,C,D,E,F,G,DP
void SegInit(void) {
  Seg1=SegConvert(0);        // initialise to zero
  Seg2=SegConvert(0);        // initialise to zero
}
char SegConvert(char SegValue) {         // function 'SegConvert'
  char SegByte=0x00;
  switch (SegValue) {                  //DP G F E D C B A
    case 0 : SegByte = 0x3F; break;    // 0 0 1 1 1 1 1 1 binary
    case 1 : SegByte = 0x06; break;    // 0 0 0 0 0 1 1 0 binary
    case 2 : SegByte = 0x5B; break;    // 0 1 0 1 1 0 1 1 binary
    case 3 : SegByte = 0x4F; break;    // 0 1 0 0 1 1 1 1 binary
    case 4 : SegByte = 0x66; break;    // 0 1 1 0 0 1 1 0 binary
    case 5 : SegByte = 0x6D; break;    // 0 1 1 0 1 1 0 1 binary
    case 6 : SegByte = 0x7D; break;    // 0 1 1 1 1 1 0 1 binary
    case 7 : SegByte = 0x07; break;    // 0 0 0 0 0 1 1 1 binary
    case 8 : SegByte = 0x7F; break;    // 0 1 1 1 1 1 1 1 binary
    case 9 : SegByte = 0x6F; break;    // 0 1 1 0 1 1 1 1 binary
  }
  return SegByte;
}
```

Program Example 6.6: Source code for SegDisplay.cpp

Note that **SegDisplay.cpp** file has an **#include** directive to the **SegDisplay.h** header file. This is given in Program Example 6.7.

```
/* Program Example 6.7: SegDisplay.h file for modular 7-seg keyboard
controller
                                                                  */
#ifndef SEGDISPLAY_H
#define SEGDISPLAY_H
#include "mbed.h"
extern BusOut Seg1;     // allow Seg1 to be manipulated by other files
extern BusOut Seg2;     // allow Seg2 to be manipulated by other files
void SegInit(void);                 // function prototype
char SegConvert(char SegValue);     // function prototype
#endif
```

Program Example 6.7: Source code for SegDisplay.cpp

The **DisplaySet** and **GetKeyInput** functions are to be "owned" by the **HostIO.cpp** source file, as is the Serial USB interface object named "**pc**". The **HostIO.cpp** file should therefore be as shown in Program Example 6.8.

```
/* Program Example 6.8: HostIO.cpp code for modular 7-seg keyboard controller
                                                                          */
#include "HostIO.h"
Serial pc(USBTX, USBRX);       // communication to host PC
void HostInit(void) {
  pc.printf("\n\rType two digit numbers to be \n\r");
}
char GetKeyInput(void) {
  char c = pc.getc();    // get keyboard ascii data
  pc.printf("%c",c);     // print ascii value to host PC terminal
  return (c&0x0F);       // return value as non-ascii
}
```

Program Example 6.8: Source code for HostIO.cpp

```
/* Program Example 6.9: HostIO.h code for modular 7-seg keyboard controller
                                                                          */
#ifndef HOSTIO_H
#define HOSTIO_H
#include "mbed.h"
extern Serial pc;        // allow pc to be manipulated by other files
void HostInit(void);     // function prototype
char GetKeyInput(void);  // function prototype
#endif
```

Program Example 6.9: Source code for HostIO.h

The HostIO header file, **HostIO.h**, is shown in Program Example 6.9.

Create the modular seven-segment display project given by the Program Examples 6.5–6.9. You will need to create a new project in the mbed compiler and add the required modular files by right clicking on the project and selecting "New File". Hence create a file structure which replicates Fig. 6.9.

You should now be able to compile and run your modular program. Use the circuit of Fig. 6.6.

Exercise 6.3

Create an advanced modular project which uses a host terminal application and a servo.

The user inputs a value between 1 and 9 from the keyboard which moves the servo to a specified position. An input of 1 moves the servo to 90 degrees left and a user input

of 9 moves the servo to 90 degrees right. Numbers between 1 and 9 move the servo to a relative position, for example, the value 5 points the servo to the center.

You can reuse the **GetKeyInput( )** function from the previous examples.

You may also need to create a lookup table function to convert the numerical input value to a suitable PWM duty cycle value associated with the desired servo position. ■

We have seen that functions are useful for allowing us to write clean and readable code while allowing manipulation of data. This in turn has enabled us to create modular programs, which therefore enable large multifunctional projects to be programmed. The program development can also be managed through a team of engineers and with a mechanism that enables reuse of code and a simple approach to updating and upgrading software features.

6.7 Working With Bespoke Libraries

Every program we have seen so far has used just one library—the standard **mbed** library that is shown in the program folder tree (as seen in Fig. 6.9 earlier), which includes all the functions that allow us to exploit the mbed's features, including digital and analog inputs, digital and analog outputs, and the PWM. From hereon in this book, a number of other libraries will also be used. Many libraries exist for implementing additional mbed features as well as for connecting and communicating with peripheral devices such as liquid crystal displays, temperature sensors, and accelerometers. Some of these libraries are provided by the mbed official website, whereas others have been created by advanced developers who have allowed their code to be shared through the online mbed community. We call these the "bespoke" libraries.

When writing a program that accesses functions defined within a bespoke library file, it is necessary to import the library to the project through the mbed compiler. There are two ways to do this when using the mbed online compiler. Firstly, from the compiler, it is possible to right click on the project folder and select the Import Library Option. This allows the library to be loaded either through via the Import Wizard facility or directly by entering the library URL, if you know it. The menu for importing libraries to an mbed project is shown in Fig. 6.10.

If using the mbed Import Wizard, it is possible to search for a library of a specific name and then select the one which you wish to import, as shown in Fig. 6.11. When using the Import Wizard the author of the library and their most recent modification date are given, which can be useful information when looking for libraries by specific developers or from

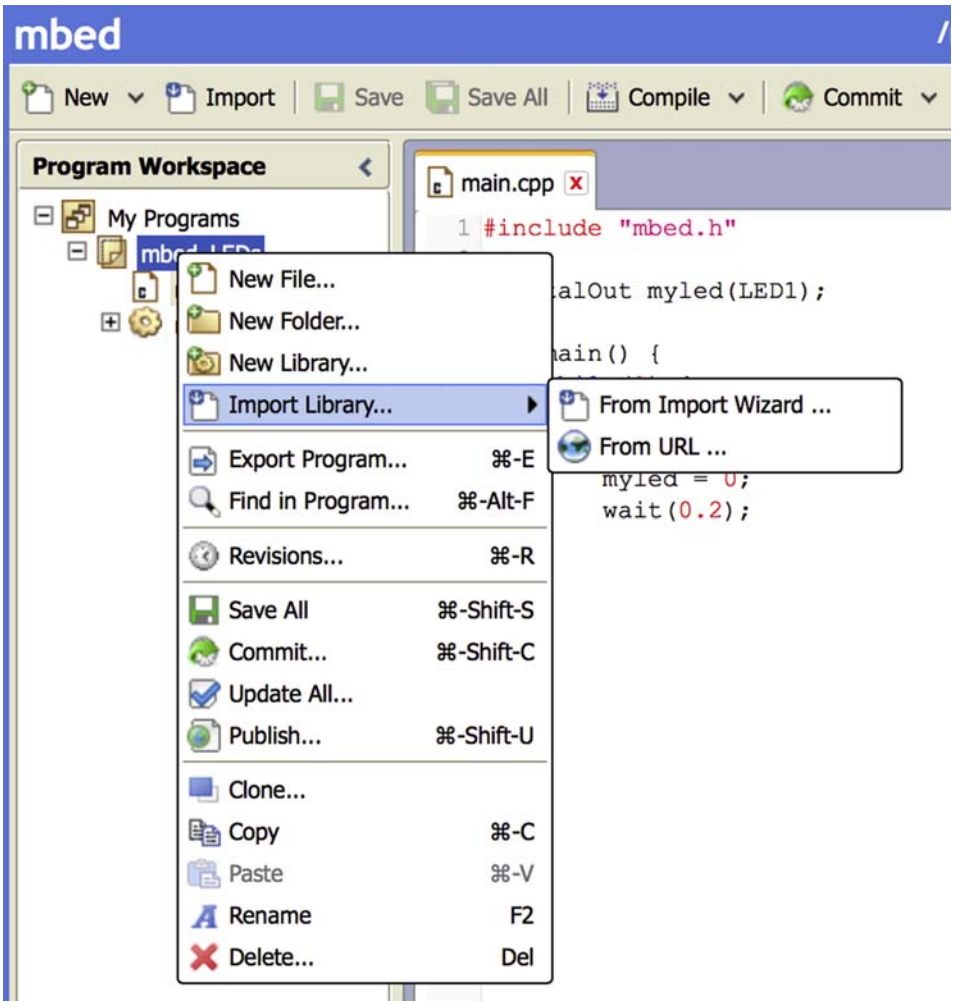

Figure 6.10
Importing a bespoke library to an mbed project.

a particular time period. In Fig. 6.11, the search term "USB" has been used to access a list of mbed libraries that have the search term in their name. It can be seen that the first two libraries in the list, **USBDevice** and **USBHost**, are official mbed libraries (and will be discussed later in the book), whereas all the libraries listed thereafter are authored by developers from the mbed community.

The second method for importing libraries is directly from the mbed website itself. It is possible to browse programs and library code on the mbed website—both those provided by mbed themselves and those made available and published by developers within the mbed community. For example, the mbed official **USBDevice** library can be found at the

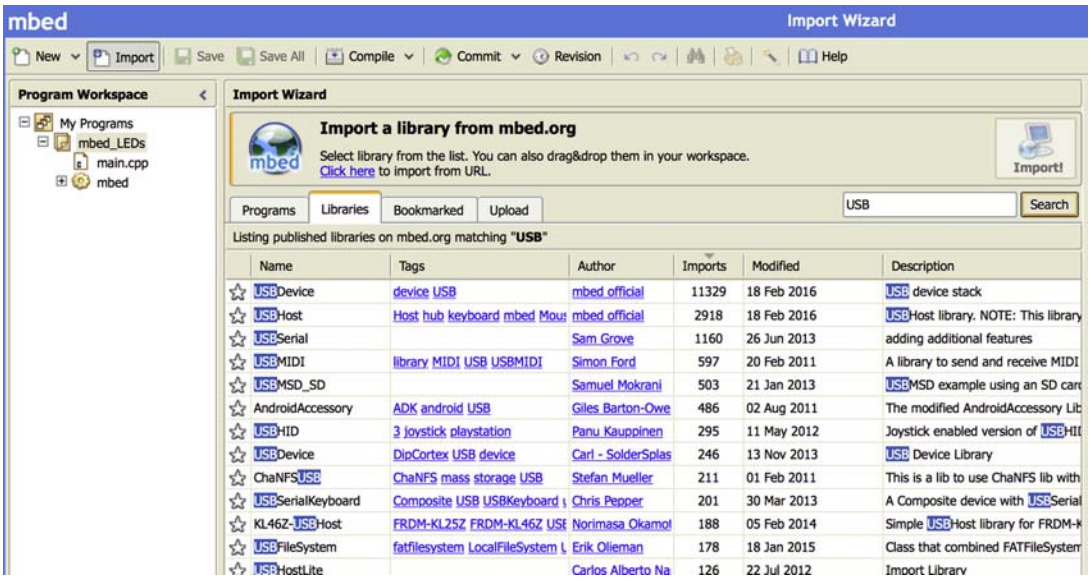

Figure 6.11
mbed Import Wizard.

web address shown below. The corresponding **USBDevice** library webpage is shown in Fig. 6.12.

Library	Library URL and import path
USBDevice	https://developer.mbed.org/users/mbed_official/code/USBDevice/

From the library URL webpage, it is possible to import the library directly into the mbed compiler by selecting the "Import this library" option on the right hand side. When the import option is selected a new window opens which asks for the destination program to be chosen, which then allows the library import to be completed.

In many examples hereon in this book, bespoke libraries will be discussed and used. In each case the author who has developed the library code will be credited and the library webpage and import path will be included in the chapter's final References list. Each time it is important to remember to import any necessary bespoke libraries to allow the program examples to compile successfully. Note that every effort has been made to give correct links to reliable libraries, however, webpages do sometimes become readdressed over time and authors may choose to remove or update their code. Equally, bespoke libraries cannot be 100% guaranteed to work correctly in every detail, because there is limited quality control over their design and programming. However, there are many valuable resources, which do function correctly, and can be used to both speed up code writing and to easily connect with advanced peripheral devices. In all the examples in this

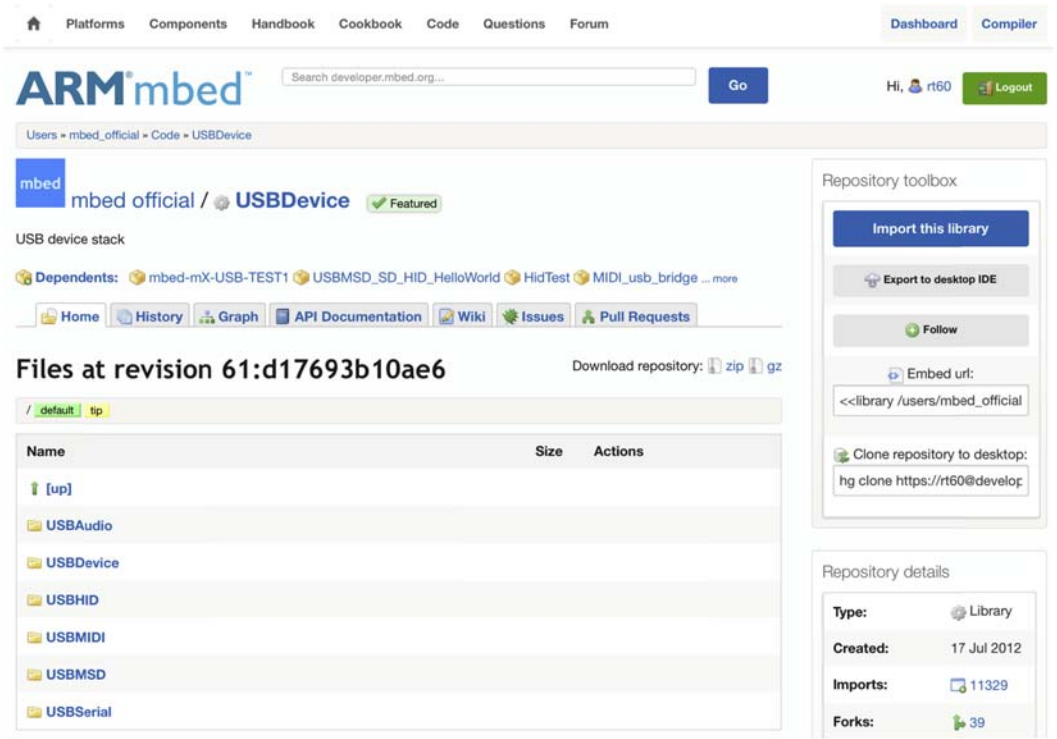

Figure 6.12
USBDevice library webpage.

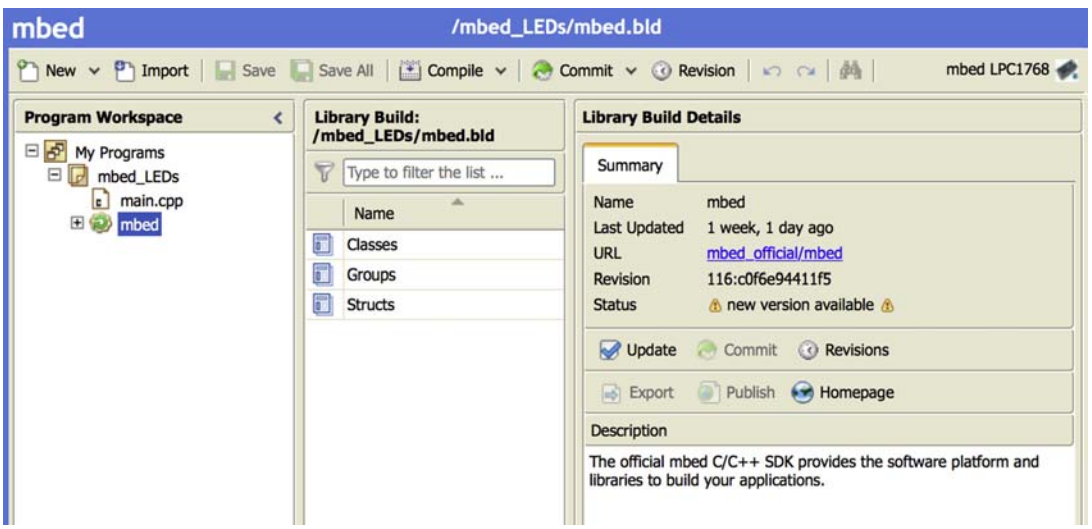

Figure 16.13
mbed Library Build Details showing that and updated version is available.

book, which use bespoke libraries, care has been taken to ensure that examples are functional and reliable at the time of writing.

Since libraries (including the mbed official ones) are often "works in progress," you may sometimes need to update libraries to the most recent version. The easiest way to do this is to select the library in your program folder and look at its current status on the right hand side of the compiler, as shown in Fig. 6.13. This Figure highlights the Library Build Details and shows that a new version is available, i.e., it has been updated since the user's program **mbed_LEDs** was created. If the user wishes to update the library (which is usually, but not always, recommended), they can simply select the Update button shown in the Summary panel. Note also that the mbed library in the Program Workspace shows a small green arrow on its icon—this arrow indicates also that an updated library is available.

It is not always necessary to update libraries just because a new version is available. If your code compiles and functions perfectly then there is no need to update the library at all. If you are creating a new program then we suggest you always use the most recent library versions, however, if revisiting a program from the past, you may wish to leave it exactly as it is.

Chapter Review

- It is essential to plan program design with care.
- We can use flow charts and pseudocode to assist program design.
- We use functions to allow code to be reusable and easier to read.
- Functions can take input data values and return a single data value as output; however, it is not possible to pass arrays of data to or from a function.
- The technique of modular programming involves designing a complete program as a number of source files and associated header files. Source files hold the function definitions whereas header files hold function and variable declarations.
- The C/C++ compilation process compiles all source and header files and links those together with predefined library files to generate an executable program binary file.
- Preprocessor directives are required to ensure that compilation errors owing to multiple variable declarations are avoided.
- Modular programming enables a number of engineers to work on a single project, each taking responsibility for a particular code feature.
- Bespoke libraries, developed and shared by mbed staff, or programmers in the mbed community, can be used in mbed projects to allow advanced peripherals and mbed features to be implemented quickly and reasonably reliably.

Quiz

1. List the advantages of using functions in a C program.
2. What are the limitations associated with using functions in a C program?
3. What is pseudocode and how is it useful at the software design stage?
4. What is a function "prototype" and where can it be found in a C program?
5. How much data can be input to and output from a function?
6. What is the purpose of the preprocessor in the C program compilation process?
7. At what stage in the program compilation process are predefined library files implemented?
8. When would it be necessary to use the **extern** storage class specifier in an mbed C program?
9. Why is the **#ifndef** preprocessor directive commonly used in modular program header files?
10. Draw a program flow chart which describes a program that continuously reads an analog temperature sensor output once per second and displays the temperature in degrees Celsius on a 3-digit seven-segment display.

References*

* No external sources are referenced for this chapter. See, however, Appendix B references for further support information.