Chapter 6:
Further Programming Techniques

rt rev. 12.9.16

www.embedded-knowhow.co.uk

The benefits of considered program design

In a typical embedded program design, it's not usually possible to program everything into a single control loop, so the code will need to be broken up into smaller elements. In particular it helps when:

- code is readable, structured and documented;

- code can be tested in a modular form;

- development reuses existing code utilities to keep development time short;

- code design supports multiple engineers working on a single project;

- future upgrades to code can be implemented efficiently;

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

Functions

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 the result to the main program. It is also possible to use functions with no input or output data.

Main program

Function

call

return
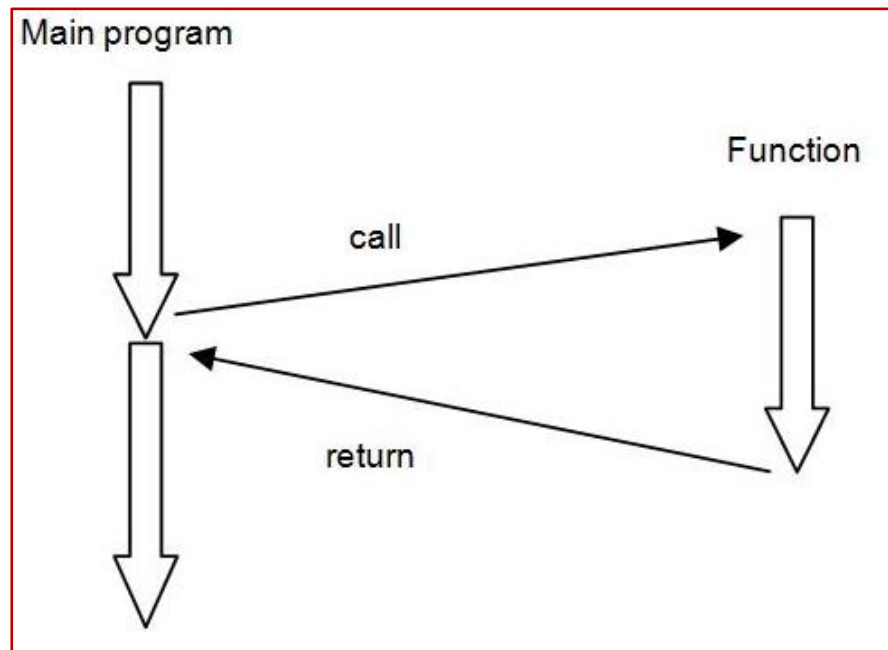
A function call

Program design

It is often useful to use a flowchart to indicate the operation of program flow and the use of functions. We can design code flow using a flowchart prior to coding. Figure 6.2 shows some of the flowchart symbols that are used.
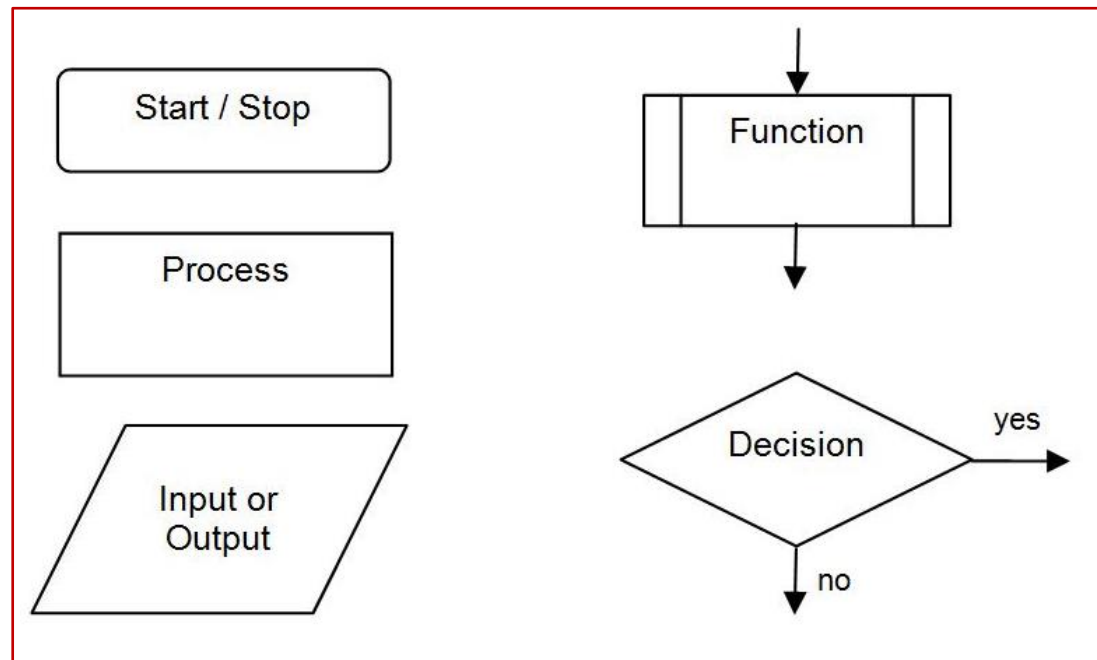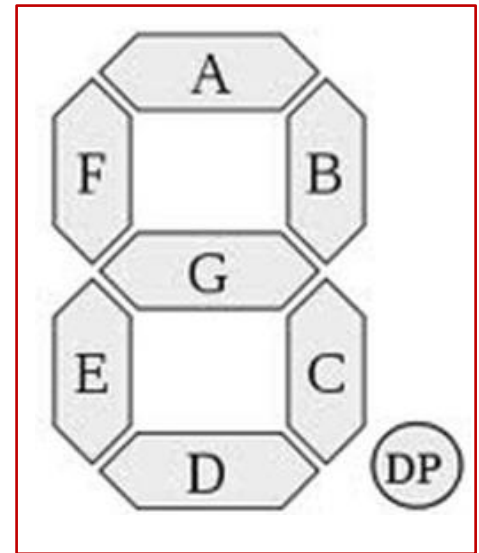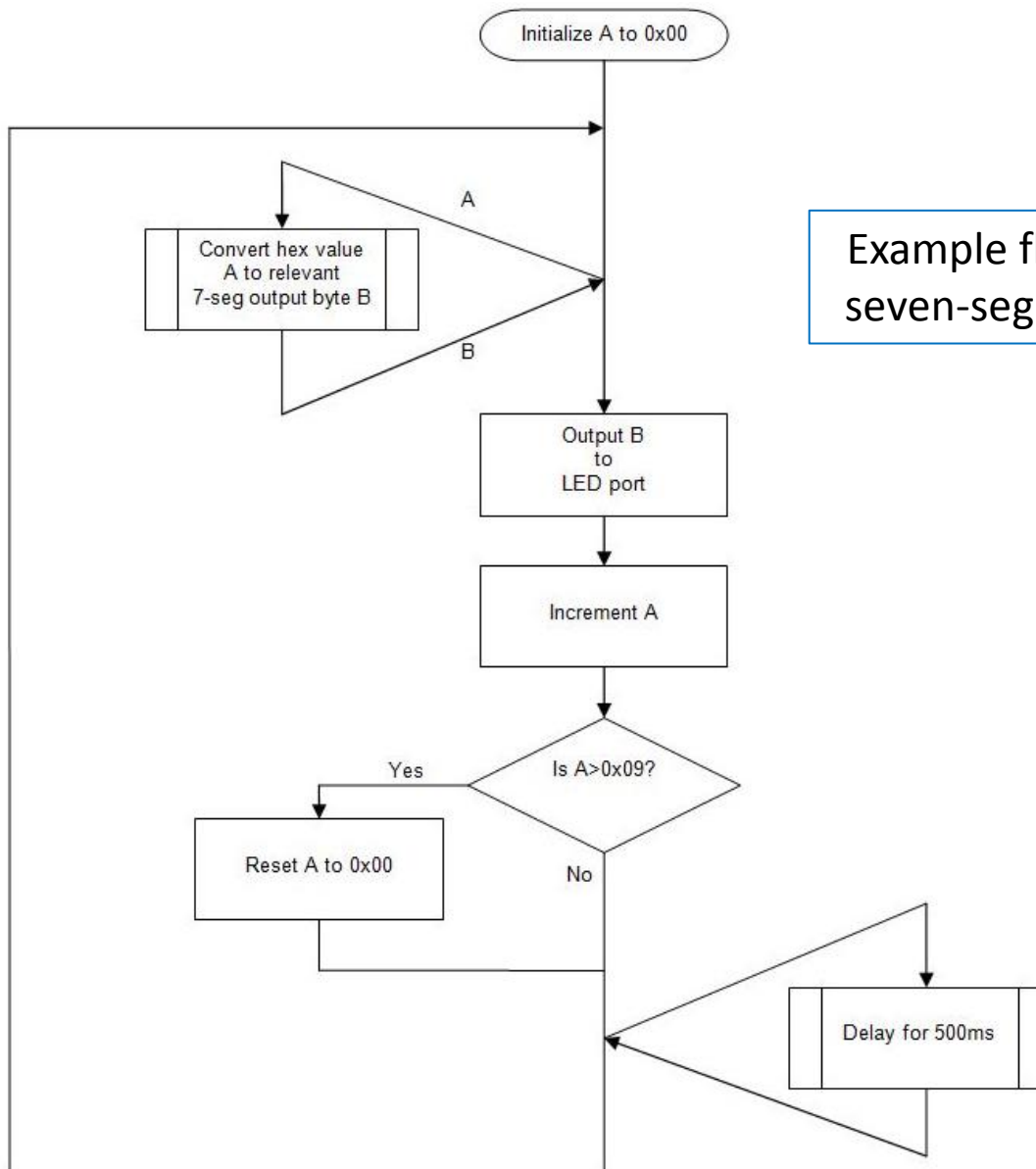

Example flow chart symbols

Program design example 1

"Design a program to increment continuously the output of a seven-segment numerical LED display through the numbers 0 to 9, then reset back to 0 to continue counting. This includes:

- 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 zero;

- Delay for 500ms to ensure that the LED output counts up at a rate that is easily visible.
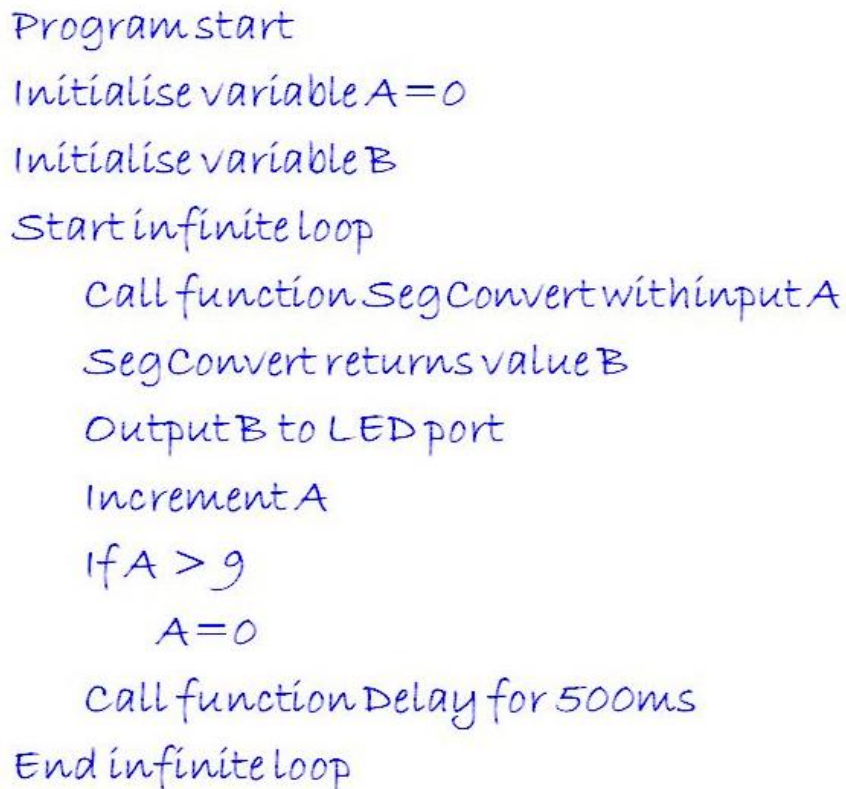
Program design example 2

Example flowchart design for a seven-segment display counter

Pseudocode

Pseudocode consists of short, English phrases used to define specific actions within a program.

```
Program start
Initialise variable A = 0
Initialise variable B
Start infinite loop
    Call function Seg Convert with input A
    Seg Convert returns value B
    Output B to LED port
    Increment A
    If A > 9
        A = 0
    Call function Delay for 500ms
End infinite loop
```

Example pseudocode for seven-segment display counter

Implementation of program design example

```c
/* 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;
}
```
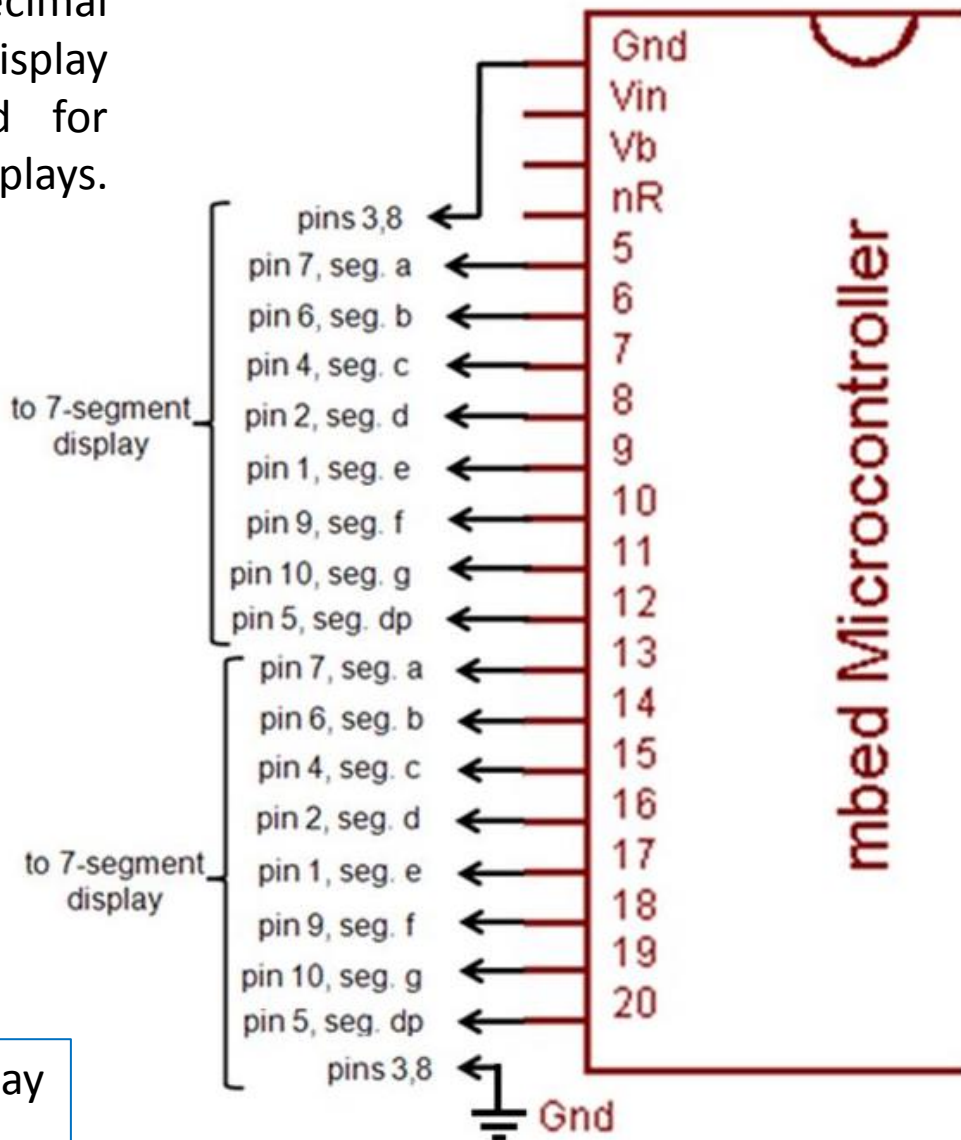
This program realises both the previous pseudocode and the flow diagram.

Function reuse

The function to convert a decimal value to a seven-segment display byte, can now be adapted for multiple seven-segment displays. See Program Example 6.2.

Two seven-segment display control with the mbed

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. A simplified version of the compilation process is shown here.
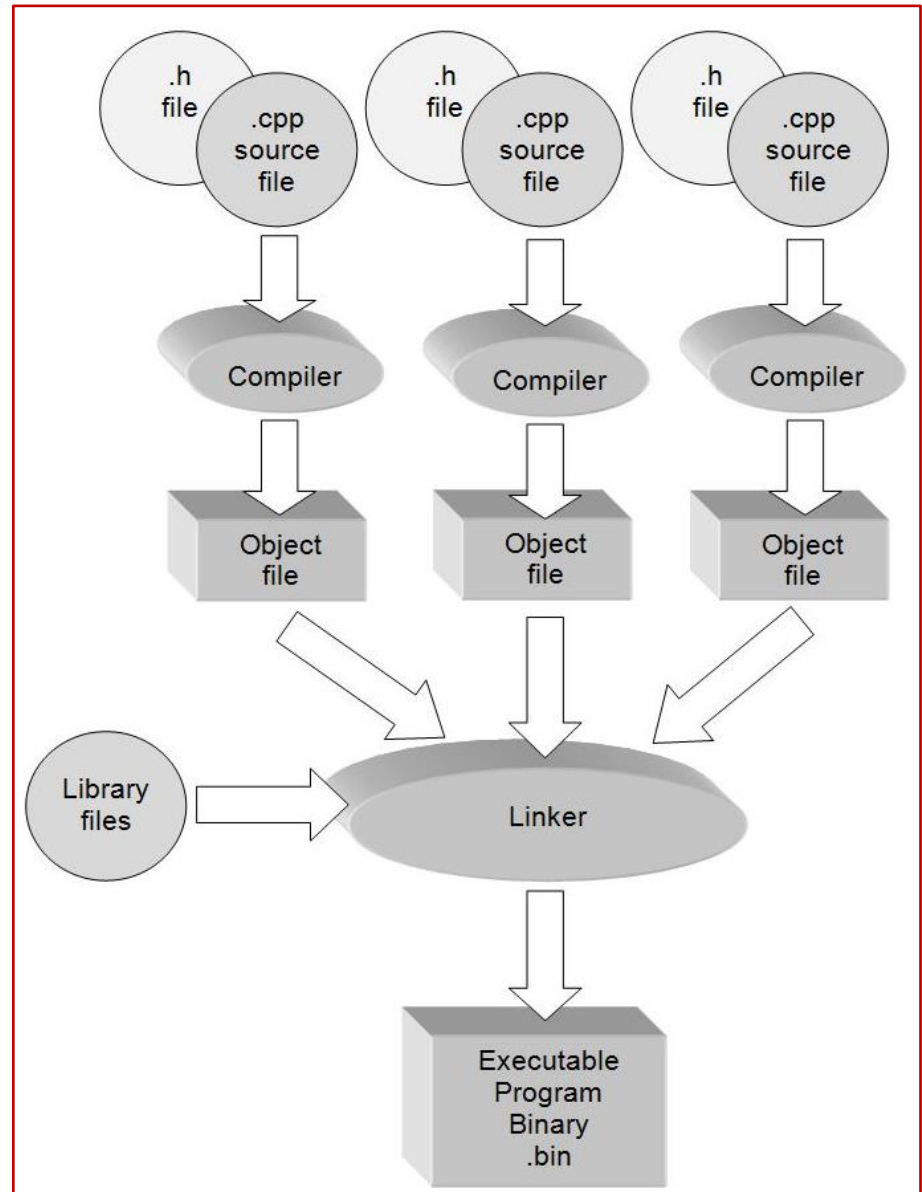
Using #define, #include, #ifndef and #endif directives

The C/C++ pre-processor modifies code before the program is compiled. Pre-processor directives are denoted with a "#" symbol.

The **#include** directive is used to tell the pre-processor to include any code or statements contained within an external header file; **#include** essentially just acts as a copy and paste feature.

The **#define** directive allows use of meaningful names for specific numerical constants, for example:
```
#define    SAMPLEFREQUENCY   44100
#define    PI                3.141592
```

The **#ifndef** directive means "if not defined", and helps to avoid multiple definition of a variable in different files, for example:
```
#ifndef VARIABLE_H    // if VARIABLE_H has not previously been defined
#define VARIABLE_H    // define it now
```

The **#endif** directive is used to indicate the end of the **#ifndef** conditional

Using mbed objects globally

All mbed objects must be defined in an "owner" source file. But we may want to use those objects in other files in the project, i.e. "globally". This can be done by defining the mbed object in the owner's header file. When an mbed object is defined for global use, the **extern** specifier should be used. For example, in Program Example 6.6, the file **SegDisplay.cpp** defines **Seg1** as follows:

```
BusOut Seg1(p5,p6,p7,p8,p9,p10,p11,p12);
```

As other source files need to manipulate **Seg1**, it is also declared in the **SegDisplay.h** header file (Program Example 6.7) using the **extern** specifier, as follows:

```
extern BusOut Seg1;
```

The specific mbed pins don't need to be redefined in the header file, as these will have already been specified in the original object declaration.

Modular program example

Complex programs can now be built up from multiple files. Actual program code for this example can be found as Program Example 6.5 to 6.9, in the book or on the support web site.
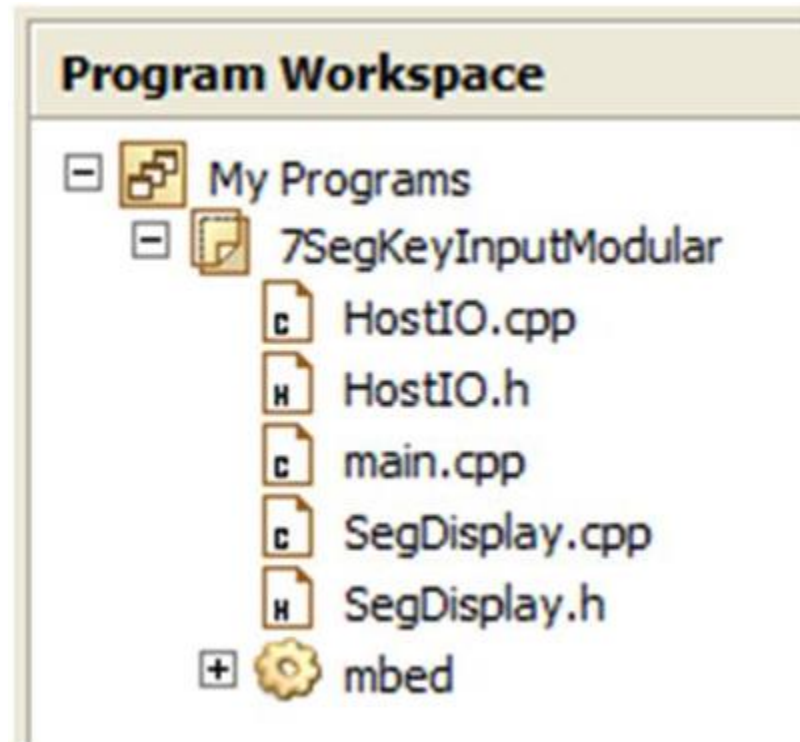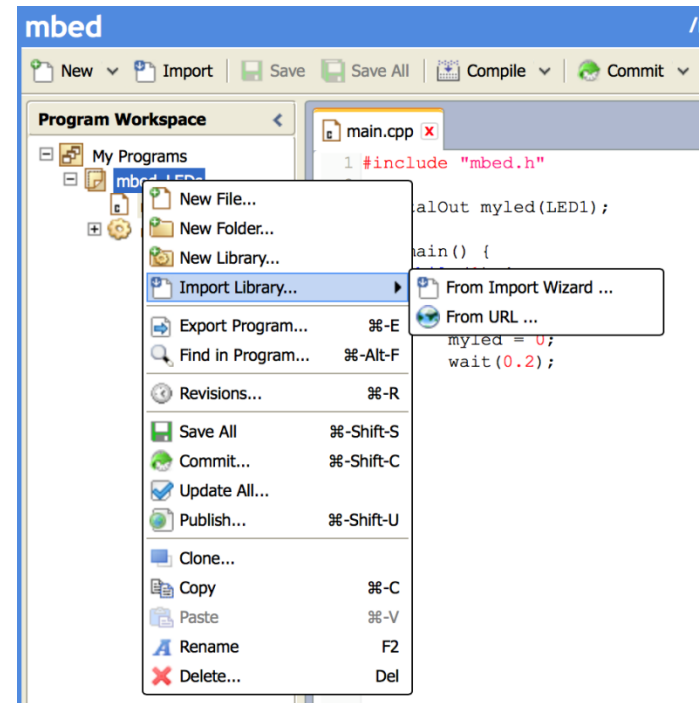
Working with bespoke libraries

Many libraries exist for implementing additional mbed features. 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 and select the Import Wizard (see image).

Working with bespoke libraries

The second method for importing libraries is directly from the mbed website itself with the library's own URL.

Select 'Import this library' option and you will then be asked to choose the mbed program which you want to import the library in to.

Updating libraries

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.

If a new version is available the 'Update' selection will be active.

Chapter quiz questions

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 pre-processor 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** pre-processor 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.

Chapter review

- 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.

- We can use flow charts and pseudo-code to assist program design.

- 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.

- Pre-processor 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 can be used in mbed projects to allow advanced peripherals and mbed features to be implemented quickly and reasonably reliably.