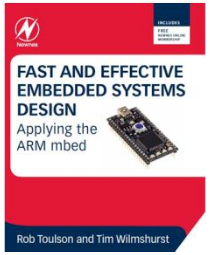
Embedded Systems Design Course

Applying the mbed microcontroller

Modular design and programming techniques

These course notes are written by R.Toulson (Anglia Ruskin University) and T.Wilmshurst (University of Derby). (c) ARM 2012

These course notes accompany the textbook "Fast and effective embedded system design : Applying the ARM mbed"


FAST AND EFFECTIVE EMBEDDED SYSTEMS DESIGN
Applying the ARM mbed
Rob Toulson and Tim Wilmshurst

Modular design and programming techniques

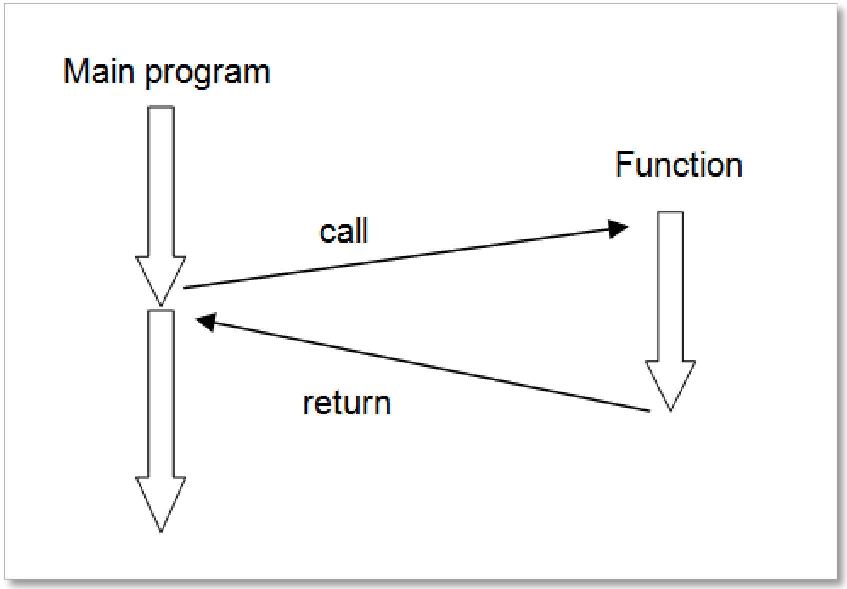
- Developing advanced embedded systems
- Functions and subroutines
- Working with 7-segment displays
- Building mbed projects with functions
- Modular programming
- Using header files
- Creating a modular program

Developing advanced embedded systems

- When working on large, multi-functional projects it is particularly important to consider the design and structure of the software. It is not possible to program all functionality into a single loop!
- The techniques used in large multifunctional projects also bring benefits to individuals who work on a number of similar design projects and regularly reuse code.
- Good practice for embedded system design includes:
  - Code that is readable, structured and documented
  - Code that can be tested in a modular form
  - Reuses existing code utilities to keep development time short
  - Support multiple engineers working on a single project
  - Allow for future upgrades to be implemented
- There are a number of C/C++ programming techniques which enable these design requirements to be considered…

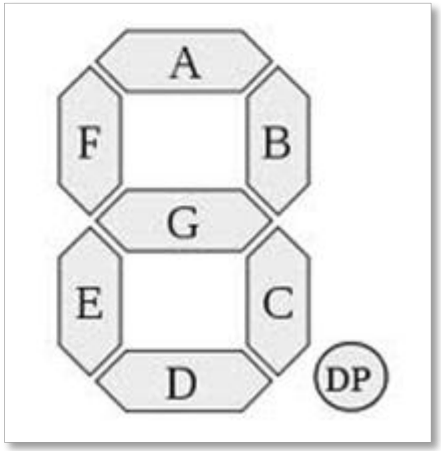
Functions and subroutines

- A function (sometimes called a subroutine) is a portion of code within a larger program
- A function performs a specific task and is relatively independent of the main code



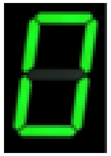
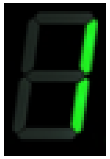
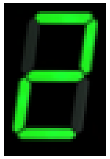
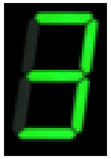
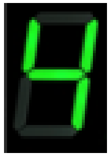
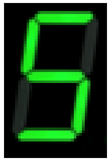
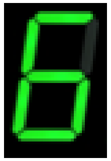
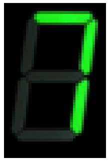
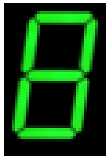
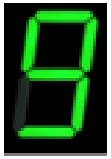
Functions and subroutines

To demonstrate the use of functions, we will look at examples using a standard 7-segment LED display.



Note that the 8-bit byte for controlling the individual LED's on/off state is configured as:

(MSB) DP G F E D C B A (LSB)

A	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09
B (MSB) (LSB)	0011 1111	0000 0110	0101 1011	0100 1111	0110 0110	0110 1101	0111 1101	0000 0111	0111 1111	0110 1111
7-seg										

Working with 7-segment displays

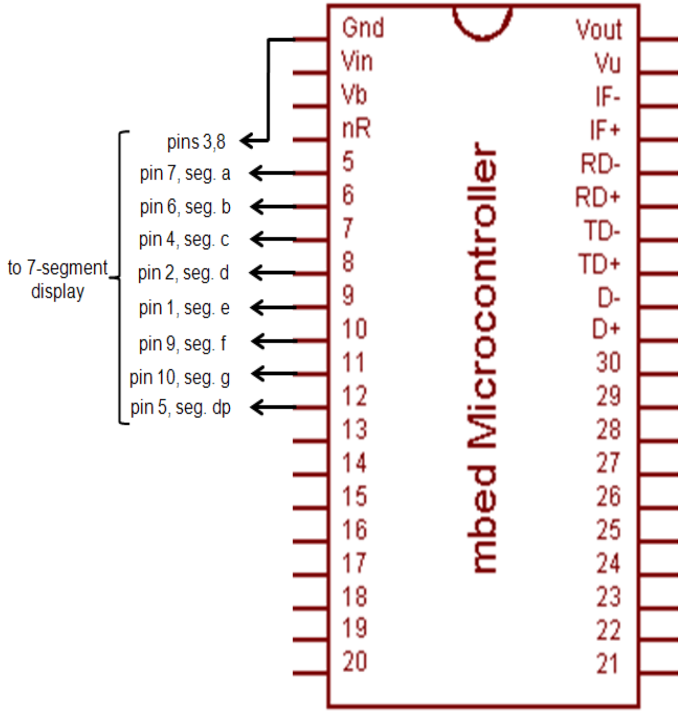
- A 7-segment display is actually just 8 LEDs in a single package. We can therefore connect each LED pin to an mbed pin to display a chosen number.

- For example, the code here uses a digital output for each LED segment and switches the correct LEDs on and off to display the number 3.

```
#include "mbed.h"

DigitalOut A(p5);
DigitalOut B(p6);
DigitalOut C(p7);
DigitalOut D(p8);
DigitalOut E(p9);
DigitalOut F(p10);
DigitalOut G(p11);
DigitalOut DP(p12);

int main() {
    A=1;
    B=1;
    C=1;
    D=1;
    E=0;
    F=0;
    G=1;
    DP=0;
}
```



Working with 7-segment displays

- You may see that this code can get a little intensive; it is a very simple operation but:
  - we have defined a large number of digital outputs
  - we have to set each output to a very specific value in order to display a single number

- There is an issue with scalability here:
  - What if we want to change the number regularly? Or output a number based on the result of a calculation?
  - What if we need more digits, i.e. decimal places or numbers with tens, hundreds, thousands etc?

```
#include "mbed.h"

DigitalOut A(p5);
DigitalOut B(p6);
DigitalOut C(p7);
DigitalOut D(p8);
DigitalOut E(p9);
DigitalOut F(p10);
DigitalOut G(p11);
DigitalOut DP(p12);

int main() {
    A=1;
    B=1;
    C=1;
    D=1;
    E=0;
    F=0;
    G=1;
    DP=0;
}
```

- If we want our LED display to continuously count from 0-9, our code might look something like this →

- <u>Exercise 1</u>: connect a 7-segment display to the mbed and verify that the code example continuously counts from 0-9

```
// program code for Exercise 1
#include "mbed.h"
DigitalOut A(p5);       // segment A
DigitalOut B(p6);       // segment B
DigitalOut C(p7);       // segment C
DigitalOut D(p8);       // segment D
DigitalOut E(p9);       // segment E
DigitalOut F(p10);      // segment F
DigitalOut G(p11);      // segment G
DigitalOut DP(p12);     // segment DP

int main() {
    while (1) {                                  // infinite loop
        A=1; B=1; C=1; D=1; E=1; F=1; G=0; DP=0;  // set LEDs '0'
        wait(0.2);
        A=0; B=1; C=1; D=0; E=0; F=0; G=0; DP=0;  // set LEDs '1'
        wait(0.2);
        A=1; B=1; C=0; D=1; E=1; F=0; G=1; DP=0;  // set LEDs '2'
        wait(0.2);
        A=1; B=1; C=1; D=1; E=0; F=0; G=1; DP=0;  // set LEDs '3'
        wait(0.2);
        A=0; B=1; C=1; D=0; E=0; F=1; G=1; DP=0;  // set LEDs '4'
        wait(0.2);
        A=1; B=0; C=1; D=1; E=0; F=1; G=1; DP=0;  // set LEDs '5'
        wait(0.2);
        A=1; B=0; C=1; D=1; E=1; F=1; G=1; DP=0;  // set LEDs '6'
        wait(0.2);
        A=1; B=1; C=1; D=0; E=0; F=0; G=0; DP=0;  // set LEDs '7'
        wait(0.2);
        A=1; B=1; C=1; D=1; E=1; F=1; G=1; DP=0;  // set LEDs '8'
        wait(0.2);
        A=1; B=1; C=1; D=1; E=0; F=1; G=1; DP=0;  // set LEDs '9'
        wait(0.2);
    }
}
```

- Before moving on to functions, we can simplify our code using a BusOut object.

- The BusOut object allows a number of digital outputs to be configured and manipulated together.

- We can therefore define a BusOut object for our 7-segment display and send the desired data byte value in order to display a chosen number.

- Exercise 2: Verify that the code here performs the same functionality as that in the previous exercise.

```
// program code for Exercise 2
#include "mbed.h"

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

int main() {
    while (1) {        // infinite loop
        Seg1=0x3F;     // 00111111 binary LEDs to '0'
        wait(0.2);
        Seg1=0x06;     // 00000110 binary LEDs to '1'
        wait(0.2);
        Seg1=0x5B;     // 01011011 binary LEDs to '2'
        wait(0.2);
        Seg1=0x4F;     // 01001111 binary LEDs to '3'
        wait(0.2);
        Seg1=0x66;     // 01100110 binary LEDs to '4'
        wait(0.2);
        Seg1=0x6D;     // 01101101 binary LEDs to '5'
        wait(0.2);
        Seg1=0x7D;     // 01111101 binary LEDs to '6'
        wait(0.2);
        Seg1=0x07;     // 00000111 binary LEDs to '7'
        wait(0.2);
        Seg1=0x7F;     // 01111111 binary LEDs to '8'
        wait(0.2);
        Seg1=0x6F;     // 01101111 binary LEDs to '9'
        wait(0.2);
    }
}
```

- There are some issues with this current method for coding the 7-segment display:

  - If we wanted to add a second 7-segment display and count from 0-99, we would clearly have a problem – we’d need to write lots of extra code!

  - There is also very little flexibility with this coding method – if we want to change the functionality slightly (for example to display every third number or to change the timing) we have to make quite a lot of changes

  - There is a better way to program this type of functionality; using functions

```
// program code for Exercise 2
#include "mbed.h"

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

int main() {
    while (1) {        // infinite loop
        Seg1=0x3F;     // 00111111 binary LEDs to '0'
        wait(0.2);
        Seg1=0x06;     // 00000110 binary LEDs to '1'
        wait(0.2);
        Seg1=0x5B;     // 01011011 binary LEDs to '2'
        wait(0.2);
        Seg1=0x4F;     // 01001111 binary LEDs to '3'
        wait(0.2);
        Seg1=0x66;     // 01100110 binary LEDs to '4'
        wait(0.2);
        Seg1=0x6D;     // 01101101 binary LEDs to '5'
        wait(0.2);
        Seg1=0x7D;     // 01111101 binary LEDs to '6'
        wait(0.2);
        Seg1=0x07;     // 00000111 binary LEDs to '7'
        wait(0.2);
        Seg1=0x7F;     // 01111111 binary LEDs to '8'
        wait(0.2);
        Seg1=0x6F;     // 01101111 binary LEDs to '9'
        wait(0.2);
    }
}
```

C function syntax

- The correct C function syntax is as follows:

```
Return_type function_name (variable_type_1 variable_name_1, variable_type_2 variable_name_2,…)
{
... C code here
... C code here
}
```

- As with variables, all functions must be declared at the start of a program
- The declaration statements for functions are called prototypes
- The correct format for a function prototype is the same as in the function itself, as follows:

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

Designing a C function

- It is beneficial for us to design a C function that inputs a count variable and returns the 8-bit value for the corresponding 7-segment display, for example:

```
char SegConvert(char SegValue) {         // function 'SegConvert'
    char SegByte=0x00;
    switch (SegValue) {                 //DPGFEDCBA
        case 0 : SegByte = 0x3F;break;  // 00111111 binary
        case 1 : SegByte = 0x06;break;  // 00000110 binary
        case 2 : SegByte = 0x5B;break;  // 01011011 binary
        case 3 : SegByte = 0x4F;break;  // 01001111 binary
        case 4 : SegByte = 0x66;break;  // 01100110 binary
        case 5 : SegByte = 0x6D;break;  // 01101101 binary
        case 6 : SegByte = 0x7D;break;  // 01111101 binary
        case 7 : SegByte = 0x07;break;  // 00000111 binary
        case 8 : SegByte = 0x7F;break;  // 01111111 binary
        case 9 : SegByte = 0x6F;break;  // 01101111 binary
    }
    return SegByte;
}
```

- Note that this function uses a C ‘switch/case’ statement which performs like an ‘If’ operation with a number of possible conditions.

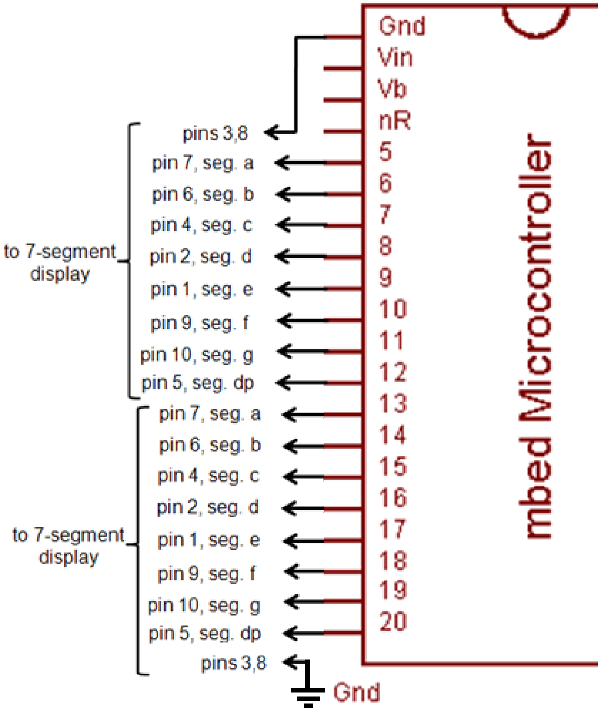
Implementing a C function

- Exercise 3: Verify that the following program, using the SegConvert function, performs the same functionality as in Exercises 1 and 2.

```
#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
int main() {                            // main program
    while (1) {                         // infinite loop
        for (char i=0;i<10;i++) {
            Seg1=SegConvert(i);
            wait(0.2);
        }
    }
}
char SegConvert(char SegValue) {    // function 'SegConvert'
    char SegByte=0x00;
    switch (SegValue) {                 //DPGFEDCBA
        case 0 : SegByte = 0x3F;break;  // 00111111 binary
        case 1 : SegByte = 0x06;break;  // 00000110 binary
        case 2 : SegByte = 0x5B;break;  // 01011011 binary
        case 3 : SegByte = 0x4F;break;  // 01001111 binary
        case 4 : SegByte = 0x66;break;  // 01100110 binary
        case 5 : SegByte = 0x6D;break;  // 01101101 binary
        case 6 : SegByte = 0x7D;break;  // 01111101 binary
        case 7 : SegByte = 0x07;break;  // 00000111 binary
        case 8 : SegByte = 0x7F;break;  // 01111111 binary
        case 9 : SegByte = 0x6F;break;  // 01101111 binary
    }
    return SegByte;
}
```

Reusing functions to reduce programming effort

- Exercise 4: To explore the power of function reuse, add a second 7-segment display to pins 13-20. You can now update the main program code to call the SegConvert function a second time, to implement a counter which counts from 0-99.



```
// main program code for  Exercise 4

#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);
            }
        }
    }
}

// SegConvert function here...
```

Summary of C functions

- Functions can be used to:
  - Process and manipulate data; we can input data values to a function, which returns manipulated data back to the main program. For example, code mathematical algorithms, look up tables and data conversions, as well as control features which may operate on a number of different and parallel data streams
  - Allow clean, efficient and manageable code to be designed
  - Allow multiple engineers to develop software features independently, and hence enable the practice of modular coding
- Functions can also be reused, which means that engineers don't have to re-write common code every time they start a new project
- Notice that in every program we also have a main() function, which is where our main program code is defined

Building complex mbed projects with functions

- Before discussing modular coding, we will design a more advanced mbed project using a number of functions
- Exercise 5: Write a program which reads a two digit number from a host terminal keyboard and outputs that number to two 7-segment displays
  - Four function prototypes are declared prior to the main program function:

```
void SegInit(void);              // function to initialise 7-seg displays

void HostInit(void);             // function to initialise the host terminal

char GetKeyInput(void);          // function to get a keyboard input from the terminal

char SegConvert(char SegValue);  // function to convert a number to a 7-segment byte
```

  - We will also use the mbed serial USB interface to communicate with the host PC and two 7-segment displays as in the previous exercises

Building mbed projects with functions

Create a new project and add the following to your main.cpp file:

```
// main program code for Exercise 5
#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 2 digit numbers
    }
}

// add function code here...
```

Building mbed projects with functions

- The following functions need to be added after your main program code:

```
// functions for Exercise 5
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 on the 7-seg display\n\r");
}

char GetKeyInput(void) {
    char c = pc.getc();      // get keyboard data (note numerical ascii range 0x30-0x39)
    pc.printf("%c",c);       // print ascii value to host PC terminal
    return (c&0x0F);         // return value as non-ascii (bitmask c with value 0x0F)
}

// copy SegConvert function here too...
```

- You will also need to copy in the code for the SegConvert function

- Your code should now compile and run

Modular programming

- Large projects in C and C++ need splitting into a number of different files. This approach improves readability and maintenance.
- For example:
  - The code for an embedded system might have one C file for the control of the attached peripherals and a different file for controlling the user input.
  - It doesn’t make sense to combine these two code features in the same source file.
  - If one of the peripherals is updated, only that piece of the code needs to be modified.
  - All the other source files can be carried over without change.
  - Working this way enables a team of engineers to work on an single project by each taking responsibility for a software feature.

Modular programming

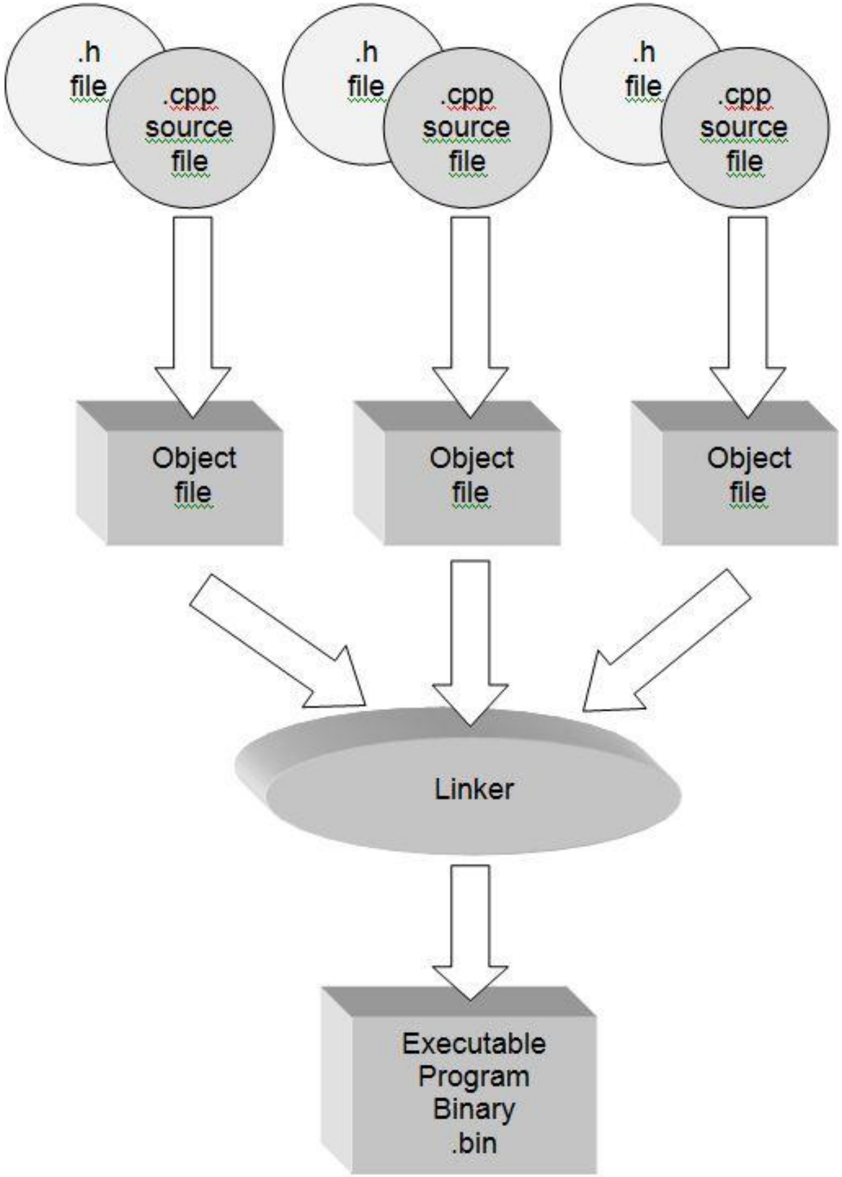
- Modular coding uses header files to join multiple files together.
- In general we use a main.cpp file to call and use functions, which are defined in feature specific .cpp files.
- Each .cpp definition file should have an associated declaration file, we call this the ‘header file’.
- Header files have a .h extension and typically include declarations only, for example compiler directives, variable declarations and function prototypes.
- A number of header files also exist from within C. These can be used for more advanced data manipulation or arithmetic. For example, math.h can be included to more easily perform trigonometric functions.

- To fully understand the design approach to modular coding, it helps to understand the way programs are pre-processed, compiled and linked to create a binary execution file for the microprocessor.

- First a pre-processor looks at a particular source (.cpp) file and implements any pre-processor directives and associated header (.h) files.
  - Pre-processor directives are denoted with a ‘#’ symbol.

- The compiler then generates an object file for the particular source code.

- Object and library files are then linked together to generate an executable binary (.bin) file.



Modular programming

- We commonly use the #include directive to tell the pre-processor to include any code or statements contained within an external header file.

- When including predefined C header files, we must put the filename in <>, for example:

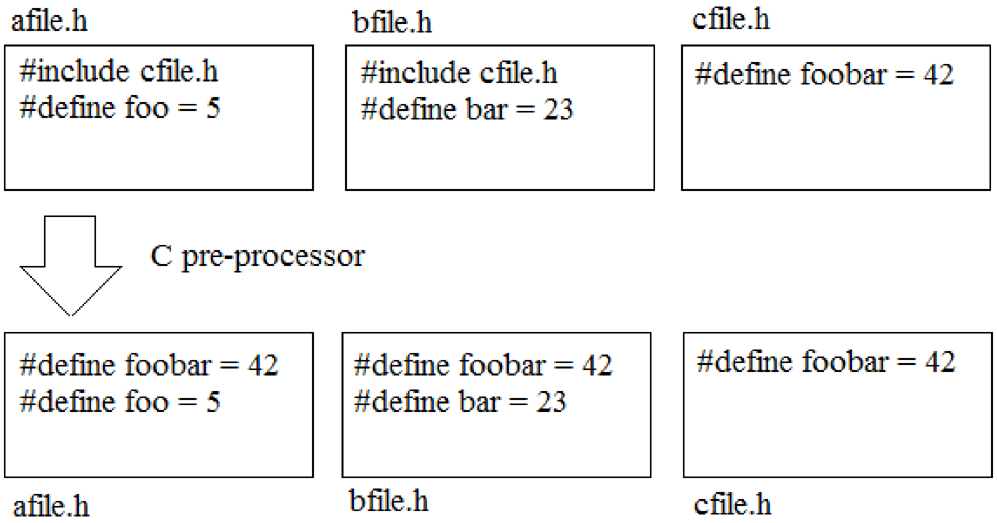
```
#include <string.h>
```

- When we include our own header files, we put the filename in quotations, for example:

```
#include "myfile.h"
```

- It is important to note that #include acts as a cut and paste feature. If you include "afile.h" and "bfile.h" where both files also include "cfile.h" You will have two copies of the contents of “cfile.h".

- The compiler will therefore see instances of multiple declarations of *foobar* and highlight an error.



afile.h

#include cfile.h
#define foo = 5

bfile.h

#include cfile.h
#define bar = 23

cfile.h

#define foobar = 42

C pre-processor

#define foobar = 42
#define foo = 5

afile.h

#define foobar = 42
#define bar = 23

bfile.h

#define foobar = 42

cfile.h

Using header files

- We can use the #ifndef pre-processor directive to ensure that header code is only ever included once by the linker.
- #ifndef means literally ‘if not defined’.
- We can define a pre-processor variable (using #define) at the start of a header file, and then only include that header code if the variable has not previously been defined.
- This way the header code is only included once for linking.
- As good practice it is therefore recommended to use the following template for header files:

```
// template for .h file
#ifndef VARIABLENAME_H         // if VARIABLENAME_H has not previously been defined
#define VARIABLENAME_H         // define it now

// header declarations here…

#endif                         // end of the if directive
```

Using header files

- Header files usually contain
  - #include statements for built in C libraries and bespoke function libraries
  - Function prototypes

- Additionally, we should include details of any mbed Objects which are required to be manipulated from outside the source file. For example:
  - a file called functions.cpp may define and use a DigitalOut object called ‘RedLed’
  - If we want any other source files to manipulate RedLed, we must also declare the Object in the header file using the extern type, as follows:

```
// functions.h file
#ifndef FUNCTIONS_H                  // if FUNCTIONS_H has not previously been defined
#define FUNCTIONS_H                  // define it now

extern DigitalOut RedLed;            // external object RedLed is defined in functions.cpp
// other header declarations here…

#endif                               // end of the if directive
```

  - Note that we don’t need to define the specific mbed pins here, as these will have already been specified in the object declaration in functions.cpp

Creating a modular program

- Exercise 6: Create the same keyboard controlled, 7-segment display project as in Exercise 5 using modular coding techniques, i.e. 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 7-segment display output
- We also need the following associated header files:
  - HostIO.h
  - SegDisplay.h
- Note that, by convention, the main.cpp file does not need a header file

Creating a modular program

- Create a new project and add the required modular files
  - To add new files, right click on your project and select ‘New File…’
- The main.cpp file will hold the same main function code as before, but with #includes to the new header files

```
// main.cpp file for Exercise 6
#include "mbed.h"
#include "HostIO.h"
#include "SegDisplay.h"

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 2 digit numbers
    }
}
```

Creating a modular program

- The SegInit and SegConvert functions are to be ‘owned’ by SegDisplay.cpp, as are the BusOut objects named ‘Seg1’ and ‘Seg2’.

- The SegDisplay.cpp file should therefore be as follows:

```
// SegDisplay.cpp file for Exercise 6

#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;
}
```

Creating a modular program

- Note that SegDisplay.cpp file has a #include to the SegDisplay.h header file.

- SegDisplay.h should be as follows:

```
// SegDisplay.h file for Exercise 6

#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
```

Creating a modular program

- The DisplaySet and GetKeyInput functions are to be ‘owned’ by HostIO.cpp, as is the Serial USB interface object named ‘pc’.
- The HostIO.cpp should therefore be as follows:

```
// HostIO.cpp code for Exercise 6

#include "HostIO.h"
#include "SegDisplay.h"        // allow access to functions and objects in SegDisplay.cpp

Serial pc(USBTX, USBRX);       // communication to host PC

void HostInit(void) {
    pc.printf("\n\rType two digit numbers to be displayed on the 7-seg display\n\r");
}

char GetKeyInput(void) {
    char c = pc.getc();         // get keyboard data (note numerical ascii range 0x30-0x39)
    pc.printf("%c",c);          // print ascii value to host PC terminal
    return (c&0x0F);            // return value as non-ascii (bitmask c with value 0x0F)
}
```

Creating a modular program

- Note that HostIO.cpp file has #includes to both the HostIO.h and the SegDisplay.h header files.

- HostIO.h should be as follows:

```
// HostIO.h file for Exercise 6

#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
```

- Your Exercise 6 program should now compile and run

Extended Task

- Exercise 7:

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

  The user can input a value between 1-9 which will move the servo motor to a specified position. An input of 1 moves the servo to 90 degrees left and 9 moves the servo to 90 degrees the right. Numbers in between 1-9 move the servo to a relative position, for example the value 5 will point the servo to the centre.

  You can reuse the GetKeyInput function from the previous exercise.

  You may also need to create a look up table function to convert the input value to a sensible PWM duty cycle value associated with the desired servo position.

Summary

- Developing advanced embedded systems
- Functions and subroutines
- Working with 7-segment displays
- Building mbed projects with functions
- Modular programming
- Using header files
- Creating a modular program