Time now to look at how **main** causes the three LaunchPad LEDs to flash in sequence.

Here is **main** again (Figure 1).

main.c | tm4c123gh6pm_startup_ccs.c

```
1 #include <stdint.h>
2 #include <stdbool.h>
3 #include "inc/hw_types.h"
4 #include "inc/hw_memmap.h"
5 #include "driverlib/sysctl.h"
6 #include "driverlib/gpio.h"
7 int main(void)
8 {
9     uint8_t ui8LED = 2;
10    SysCtlClockSet(SYSCTL_SYSDIV_4|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);
11    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
12    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);
13    while(1)
14    {
15        // Turn on the LED
16        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, ui8LED);
17        // Delay for a bit
18        SysCtlDelay(2000000);
19        // Cycle through Red, Green and Blue LEDs
20        if (ui8LED == 8) {ui8LED = 2;} else {ui8LED = ui8LED*2;}
21    }
22 }
23
```

Figure 1 **main** listing from Lab2

I've already covered the **#include** statements and the basic idea of a C function in my previous post, so no need to go over those again. Our dissection of **main** therefore begins at line 9.

Line 9.

uint8_t ui8LED = 2;

This is an example of a "variable definition" and specifies that a variable named ui8LED will be used in the program. ui8LED will be of data type uint8_t and will have an initial value of 2. C is very strict in its use of variables in that all variables must be defined first before they can be used in a program. C is sometimes referred to as a "strongly typed" language for this reason.

There are various data types which can be used when programming in C. These include integers (**int**), characters (**char**) and floating-point (**float**), amongst many others. See here for more

information. However the data type uint8_t is not one of these built-in types. So how can the variable ui8LED by defined as being of type uint8_t?

One of the nice features of Code Composer Studio (and other Eclipse-based IDEs) is that we can click on almost any part of a code listing, hit F3, and CCS will take us to wherever that entry is defined. So in the case of the data type uint8_t, because I recognise this type as not being built-in to the C language I know it must be defined elsewhere. If I click on uint8_t and hit F3 CCS opens up a new tab in the editor window, showing me that this data type is actually defined in the stdint.h library, referenced in the **#include** statement of line 1 of **main**. See Figure 2.

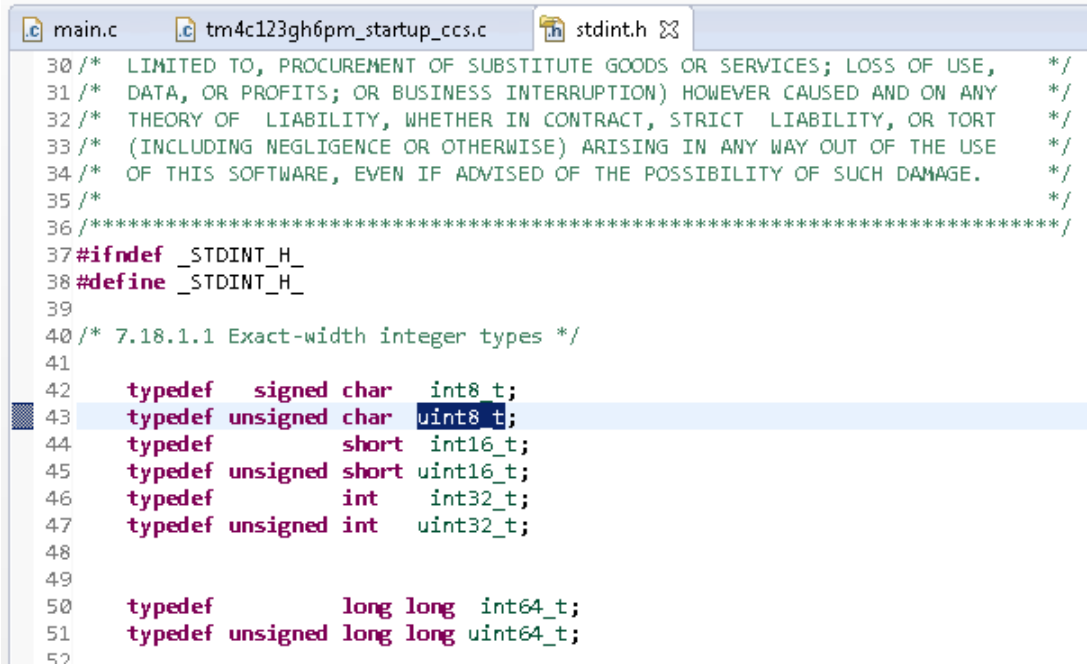

Figure 2 the uint8_t data type defined in stdint.h

Pretty useful! And also shows us that uint8_t is an "alias" for the **unsigned char** data type, which *is* one of the standard C data types, and is created by using the **typedef** mechanism. An **unsigned char** is one byte in size and can take any value between 0 and 255. The variable ui8LED can therefore also take any value between 0 and 255, and in this case it has the initial value of 2.

Line 10.

SysCtlClockSet(SYSCTL_SYSDIV_4|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);

SysCtlClockSet is the first example we have so far seen of a function written specifically for programming the microcontroller (rather than a standard built-in C function). The easiest way of determining how to use SysCtlClockSet is to refer to the TI document "TivaWare™ Peripheral Driver Library (spmu298d)" available at this link. Section 26.2.2.5 of spmu298 tells us that the SysCtlClockSet function is of the form (or has the "Prototype"):

void SysCtlClockSet(uint32_t ui32Config)

in other words, SysCtlClockSet takes a single parameter ui32Config, which is of the type uint32_t (an unsigned 32-bit integer) and returns no values (**void**). spmu298d also indicates that the function sets the clocking of the microcontroller and illustrates various different clocking options available according to the value of ui32Config.

In our specific example, the value of ui32Config is determined by "bitwise Oring" (indicated by the vertical bar symbol |) the values of SYSCTL_SYSDIV_4, SYSCTL_USE_PLL, SYSCTL_XTAL_16MHZ, SYSCTL_OSC_MAIN. These are what's known as "symbolic constants", and by using the "F3 trick" we can determine that they are defined in sysctl.h, using the **#define** pre-processor directives, as:

SYSCTL_SYSDIV_4	0x01C00000
SYSCTL_USE_PLL	0x00000000
SYSCTL_XTAL_16MHZ	0x00000540
SYSCTL_OSC_MAIN	0x00000000

These numbers are given in hex notation (0x…), but the micro works in terms of binary i.e. ones and zeros. So converting the numbers to binary we have:

SYSCTL_SYSDIV_4	0000 0001 1100 0000 0000 0000 0000 0000
SYSCTL_USE_PLL	0000 0000 0000 0000 0000 0000 0000 0000
SYSCTL_XTAL_16MHZ	0000 0000 0000 0000 0000 0101 0100 0000
SYSCTL_OSC_MAIN	0000 0000 0000 0000 0000 0000 0000 0000

where each hexadecimal digit is represented by four binary bits, giving 32 bits in total.

To determine the value of ui32Config we need to bit-wise OR these four numbers. This simply means that for each of the 32 bits, if one or more of the four variables has a 1 at that bit location then the corresponding location in ui32Config will also be a 1. So:

ui32Config 0000 0001 1100 0000 0000 0101 0100 0000

which is 0x01C00540 in hex. The SysCtlClockSet function will take this value of ui32Config and write it to the appropriate register in the microcontroller, thus configuring the device clock to our requirements. We could dig further into how exactly the SysCtlClockSet does this, but it really isn't necessary as we are given sufficient information in spmu298d to use this function. I won't discuss here precisely what this configuration means in terms of the clock configuration – much more detail on this can be found in the TM4C123GH6PM Microcontroller data sheet,

The use of symbolic constants, as illustrated in this example, make life much, much easier for the programmer. Human beings generally find it easier to recognise and understand a name such as SYSCTL_SYSDIV_4 rather than its corresponding numerical value, whether expressed in hex or binary.

Line 11.

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
```

Prototype:

```
void SysCtlPeripheralEnable(uint32_t ui32Peripheral)
```

spmu298d, Section 26.2.2.32.

This is a function which enables (or switches on) the peripheral specified by the parameter ui32Peripheral. In this case, the peripheral to be enabled is General-Purpose Input/Output (GPIO) Block F – represented by symbolic constant SYSCTL_PERIPH_GPIOF. The GPIO Block F signals appear on pins 28, 29, 30, 31 and 5 of the microcontroller, and if we look at the circuit diagram for the LaunchPad demoboard board we can see that the three LEDs (LED_R, LED_B and LED_G) are indeed connected to pins 29, 30 and 31 of the microcontroller. It therefore makes sense that we would want to enable this GPIO block. See Figure 3:

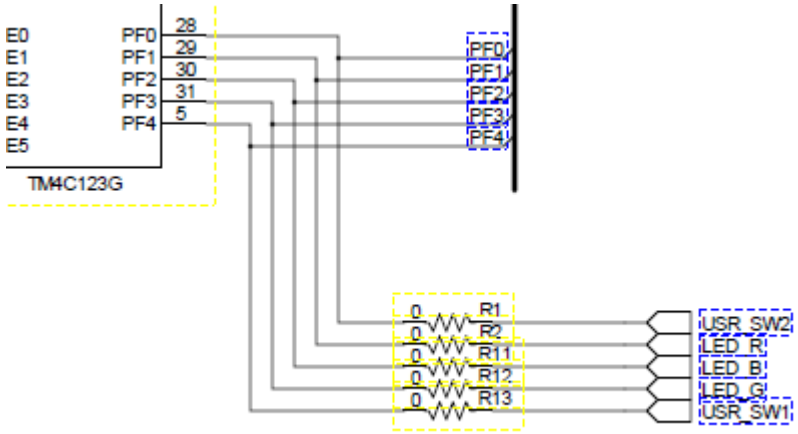

Figure 3 LED connections to the microcontroller

Taking pins 29, 30 or 31 high will turn on their corresponding LEDs via buffering transistors Q1, Q2 or Q3 (not shown).

Line 12.

GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);

Prototype:

void GPIOPinTypeGPIOOutput(uint32_t ui32Port, uint8_t ui8Pins)

spmu298d, Section 14.2.3.28.

The purpose of the GPIOPinTypeGPIOOutput function is to configure a GPIO block's pins as outputs. The function takes two parameters (and returns nothing). ui32Port determines the port to be configured, and ui8Pins the pins in that port which we wish to make outputs.

As we saw for the SysCtlClockSet function of Line 10, the ui8Pins parameter passed to the GPIOPinTypeGPIOOutput function can be derived by bitwise-ORing two or more symbolic constants – in this case GPIO_PIN_1, GPIO_PIN_2 andGPIO_PIN_3. Also as before, the detailed work is done for us in that we don't need to know the numerical values corresponding to these constants – this has already been predefined for our specific microcontroller.

In conclusion, then the purpose of Line 12 is to configure pin2 1, 2 and 3 of GPIO Block F into outputs.

Line 13.

while(1)

The “while” statement is one of the C language’s “control structures” and could be more generally expressed as:

while (expression)

{

statements

}

The purpose of the **while** structure is to repeat a block of one or more statements, contained within the curly brackets in this example, whilst the value of “expression” is true. The first time the **while** statement is executed, the value of “expression” is evaluated. If “expression” is true then the *statements* in the curly brackets are executed. Program execution then returns to the top and evaluates “expression” once again. If “expression” is still true then the *statements* are executed once again. If “expression” is false, even the first time the **while** structure is executed, then program execution skips the code in curly brackets and moves on to the next part of the program. In this way, the **while** structure could be seen as carrying out conditional looping, where one or more statements are executed repeatedly whilst a controlling expression is true.

From this explanation it should be clear (I hope!) that it makes more sense to consider the block of code from Line 13 to Line 21 inclusive, rather than just Line 13’s **while** statement alone.

Before leaving the subject of the **while** statement, there is one odd thing you may have noticed: in our example, the “expression” is simply the number 1. As this is a real, valid number, it’s value (from a logical or Boolean point of view) can never be anything other than true. Therefore the controlling expression will always be evaluated as true, no matter what else happens, and the *statements* in the curly brackets will be repeated forever (or until someone turns the power off). This is a common technique employed where we wish to make an infinite loop which just repeats a certain set of statements over and over without stopping and is also known as an “infinite loop”.

Line 15.

```
// Turn on the LED
```

The C language allows the programmer to insert comments into his or her code in order improve readability. Comments are ignored by the compiler and do not cause any machine-language object code to be generated. The above example is of a single line comment. Comments can also span multiple lines when enclosed between /* and */, as in this example:

```
/* This is an example

   of a comment split across two lines */
```

I won't generally refer to specific comments again as they tend to be self-explanatory. For example, the comment:

```
// Turn on the LED
```

gives us a clue as to the function of the code in Line 16…

Line 16.

```
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, ui8LED);
```

Prototype:

```
void  GPIOPinWrite(uint32_t ui32Port, uint8_t ui8Pins, uint8_t ui8Val)
```

spmu298d, Section 14.2.3.46.

The purpose of this function is to write the value specified by ui8Val to the pins specified by ui8Pins of the port specified by ui32Port. We have seen these three parameters already in Lines 9, 11 and 12, and so we can interpret Line 16 in the more human-friendly "Write the value 2 to pins 1, 2 and 3 of GPIO Block F". But how can a single value of 2 be written to three separate pins?

At this point, it *does* become useful to consider what is going on at the bit level. Each pin of the GPIO port is represented by a single bit in the 8-bit variable ui8Pins. So pin 0 is represented by bit 0 in ui8Pins, pin 1 by bit 1 in ui8Pins, and so on. See Figure 4.

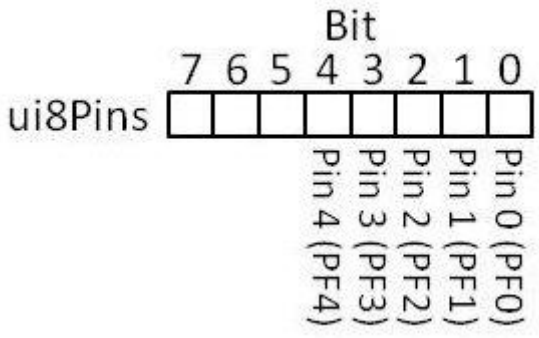

Figure 4 GPIO port pins 0 - 4 represented by ui8Pins

The value 2 (= ui8LED), expressed in 8-bit binary, is 00000010. So if we write the value 2 to the register represented by ui8Pins, then we have (Figure 5):

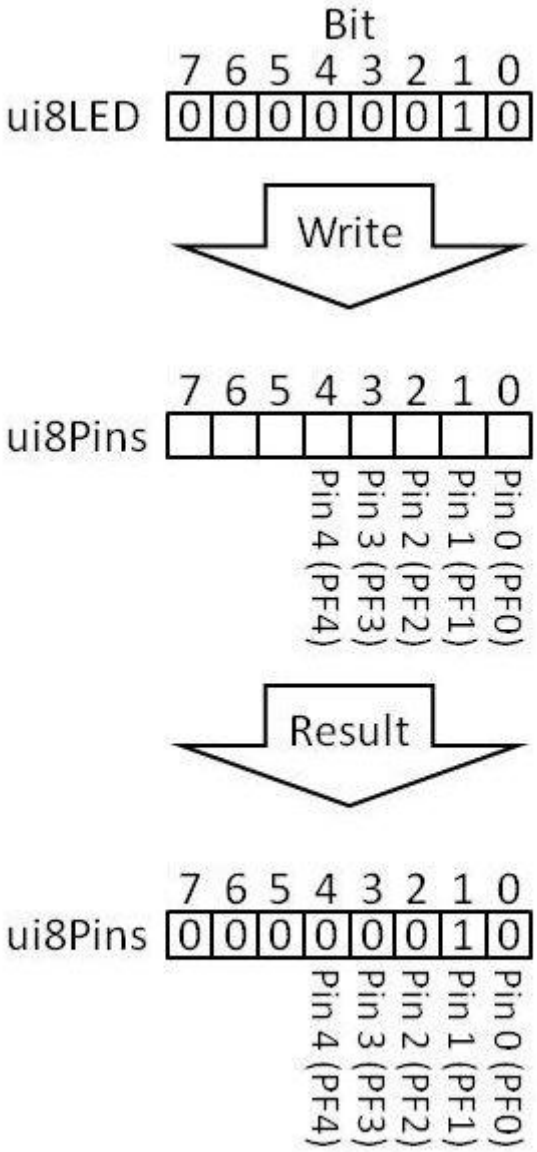

Figure 5 Writing the value 2 to the register represented by ui8Pins

By writing the value 2, represented in binary and stored in ui8LED, into the register corresponding to ui8Pins, we have turned on the Pin 1 output of GPIO Block F. Phew! Awkward to describe in words, but actually pretty simple in concept and hopefully made a bit clearer by the diagrams.

OK, so we've turned on the red LED attached to pin PF1 of the micro. What about the other two LEDs and cycling between them? Onwards…

Line 18.

SysCtlDelay(2000000);

Prototype:

void SysCtlDelay(uint32_t ui32Count)

spmu298d, Section 26.2.2.10

This function is used to provide a delay whose length depends on the value of ui32Count. A ui32Count value of 1 would result in a delay of three microcontroller instruction cycles. spmu298d makes the point, though, that accurate delay times cannot be generated as the instruction cycle time will vary from configuration to configuration. The comment in Line 17 also makes this point: "Delay for a bit". At a very rough guess, from observing the LEDs on the demoboard, I would estimate that the delay is approximately 250ms or one quarter of a second, given that the LEDs appear to change roughly four times per second.

Line 20.

if (ui8LED == 8) {ui8LED = 2;} **else** {ui8LED = ui8LED*2;}

if … **else** is an example of a C language "selection statements" and can be more generally written as:

if (expression) *statement1* [**else** *statement2*]

"expression" is evaluated, and if it is true then *statement1* is executed, otherwise *statement2* is executed.

In our example, variable ui8LED is evaluated to see if it is equal to 8. If it is then the value of ui8LED is reset to 2. If ui8LED is not equal to 8 then its existing value is multiplied by 2. Before I explain what this code is doing, I need to say a few words about the two different uses of the = sign in Line 20.

In C, a single " = " is known as an "assignment operator", where the statement ui8LED = 2 would assign the value of 2 to ui8LED. This is the ordinary, familiar usage of the = sign. In contrast, == is known as an "equality operator" and in the example above, the statement ui8LED == 8 *tests* whether or not the variable ui8LED has the value of 8. If we had written ui8LED = 8 instead of

ui8LED == 8 then we would have changed the value of ui8LED to 8 rather than testing whether its value was already 8. This is a subtle but important distinction, and incorrect usage of = and == can result in code which compiles without error but does not behave as expected.

OK, back to our **if** … **else** statement. This piece of code is most easily understood with the aid of a flowchart, as shown in Figure 6 below.

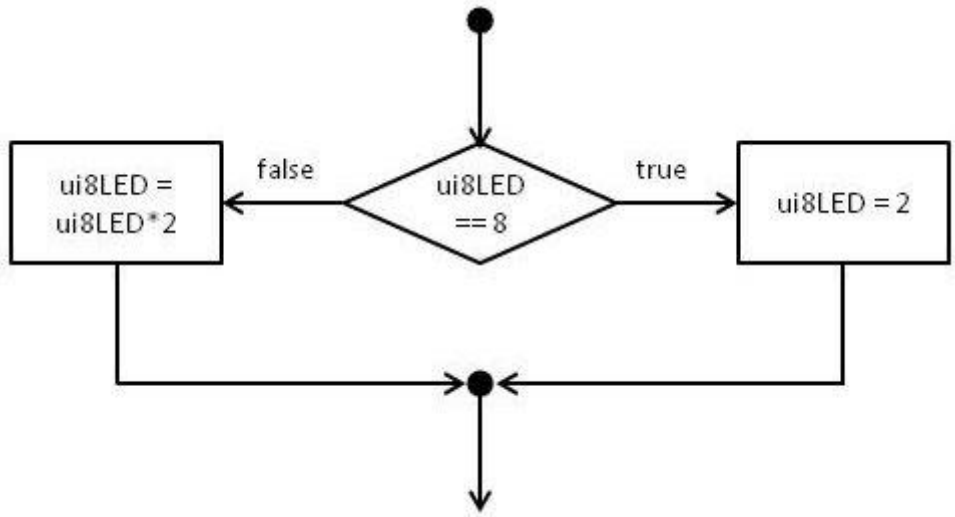

Figure 6 The **if** ... **else** statement as a flowchart

Remembering that the **if** … **else** statement is situated within an infinite **while** loop, we enter the **if** … **else** structure at the top of the flowchart and ui8LED is evaluated in the diamond decision symbol. If the value of ui8LED is found to equal 8 ("true") then its value is reset to 2 (right-hand branch) and we exit the **if** … **else** structure at the bottom of the chart. If the value of ui8LED does not equal 8 ("false") then the value of ui8LED is multiplied by 2 (left-hand branch) and we also exit the **if** … **else** structure at the bottom of the chart. This process is repeated in every iteration of the **while** loop, and the value of ui8LED continuously cycles round the pattern 2, 4, 8, 2, 4, 8, and so on.

So how does cyclically altering the value of ui8LED in this manner cause the LEDs to flash in sequence? In Figure **5** I demonstrated that the bit pattern (ui8LED) written to the variable ui8Pins determines which of the output pins of GPIO Port F will be high, and its corresponding LED turned on. The decimal number 2 written in binary is 00000010, meaning that Pin 1 will be high as shown in Figure 5. Multiplying the value of ui8LED by 2 gives us a value of 4 in decimal which is 00000100 in binary, and so writing this value to ui8Pins makes Pin 2 high and Pin 1 is now low. Multiplying ui8LED by 2 again gives a value of 8 which is 00001000 in binary with

Pin 3 high and Pins 2 and 1 low. If ui8LED is reset to 2 then we are back at the start of the sequence. Considered in binary, the sequence looks like this:

00000010
00000100
00001000
00000010
00000100
00001000

and so on., and the end result is that we see the three LEDs connected to Pins 1, 2 and 3 flashing in the same sequence. It's a neat trick and, if you think about it, would work in any number base, not just base 2 binary. Multiplying any number by its base number always shifts the number one place to the left. In decimal (base 10), for instance, if we start with the number 5 and multiply by 10 (the base number) we have 50. Multiplying by 10 again results in 500, and so on.

Summary.

The code of Figure 1 Lines 9 – 22 is summarised in the block diagram of Figure 7.

So we're finished now, right? The **main** code would compile and flash the LEDs in exactly the way intended, wouldn't it? Well, no, not exactly. If you look at the CCS screenshot of Figure **1**, next to the main.c tab you'll see a second tab labelled "t4mc123gh6pm_startup_ccs.c". A comment at thebeginning of this code describes it as "Startup code for use with TI's Code Composer Studio". The startup code sets up certain parameters within the micro which are essential to its basic operation, and the code is compiled along with main.c in order to produce the final working .out file. The workings of t4mc123gh6pm_startup_ccs.c will be the subject of my next post.

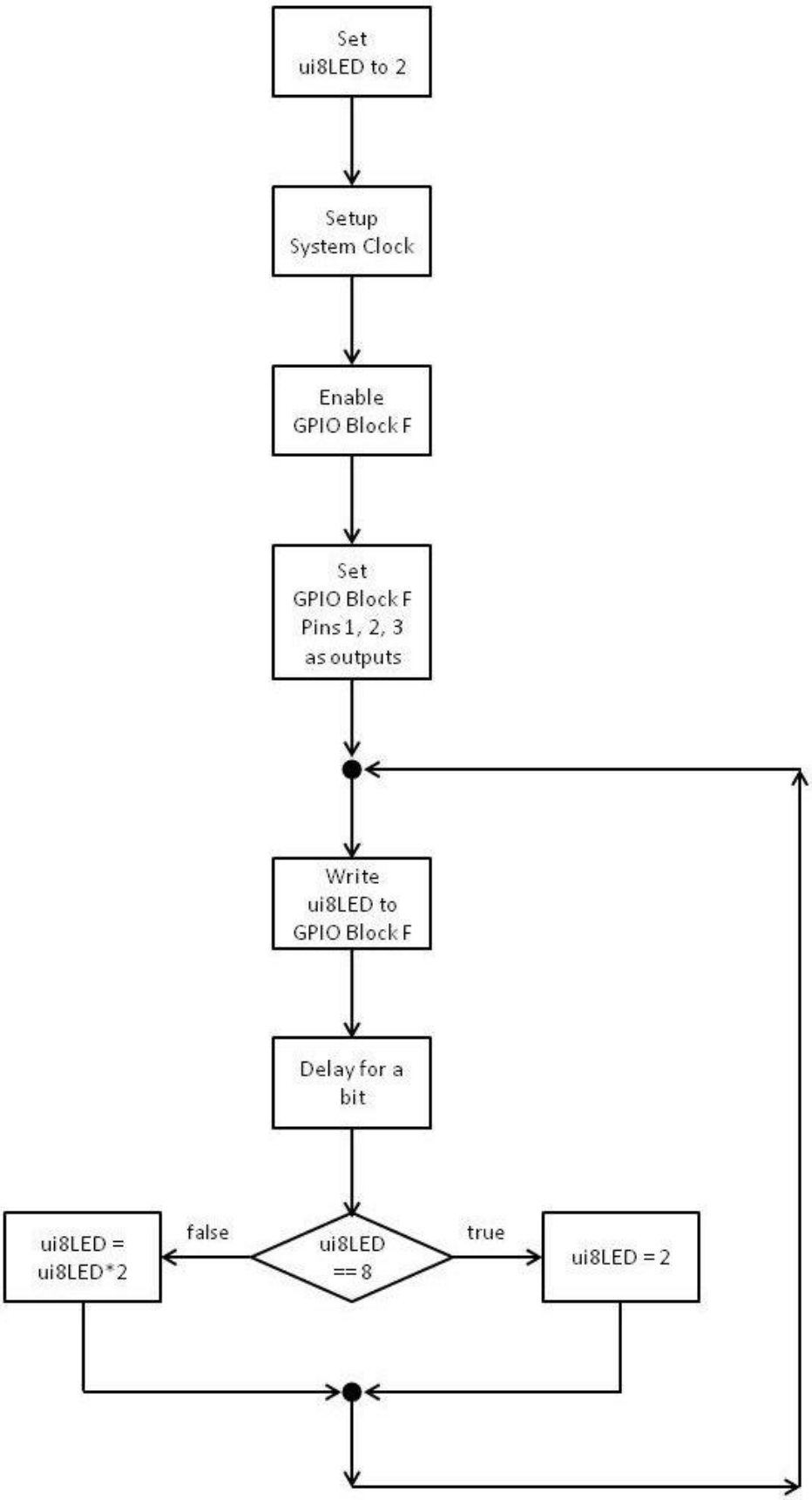

Figure 7 The code of Figure 1 Lines 9 – 22 summarised in block diagram form