Software Development

There are a number of approaches available to write the code for the processor and, indeed, the FRDM system that we are using. However, we will be using ARM mbed as the basis for our approach.

ARM Mbed

ARM provides access to a range of tools in a variety of forms:

- Web Browser
- IDE (Integrated development Environment)
- CLI (Command Line Interpreter)

We're going to look at:

the IDE, MBed Studio (which is preferred because it everything seems to integrate together);

the Web Browser based version (which can be accessed from anywhere)

MBed Studio

This is an integrated set of tools with a good UI which is available across a range of platforms (I've tried it on Win and OSX and it works well). To access these tools on your own computer, go to https://os.mbed.com/ (pretty much a one-stop shop). Scroll down to Compiler and IDE section and click on *Learn More* under Mbed Studio. This will open up a page from where you can download for Win, Mac or Linux – so whatever you are using, you should be able to get these tools to work. Follow the download instruction for your platform (just the normal pkg for OSX or a setup exe for Win). It will also be available on computers in the Lab.

Whichever tools you use, you do need to create a login (which will be the same for Mbed Studio (MS) or the web brower based approach). Once you've done it then your login will work whichever approach you take.

Once you open MS, and have logged in, you will see a blank IDE as in Figure 1:

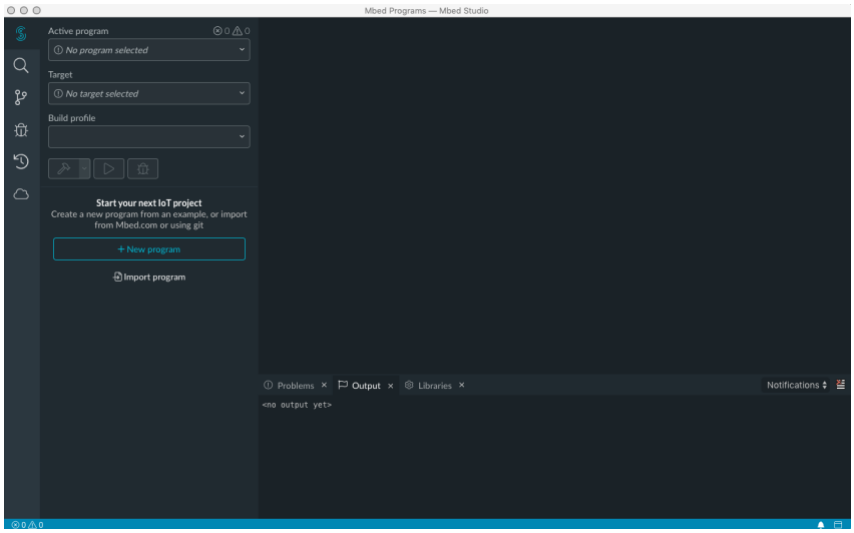


Figure 1: Basic Mbed Studio View

The buttons on the left give access to different things but the swirly S at the top will return you to this basic view. The left hand pane is where your program hierarchy will appear, the main area at the top is the window in which file will appear (for editing, for example) and the area at the bottom is where output windows are (e.g. messages during compilation).

Starting a New Program

To start a new program, just click on the dropdown under Target, find and select FRDM-KL46Z and click the blue *+ New program* button. In the window that opens, you can select an existing example program (I always select mbed-os-example-blinky) and give it a sensible name (I'm using my-program). Make sure that *Make this the active program* is checked make sure that *Store Mbed in program folder* is checked (by default this will be stored in your User location on the computer under Mbed programs/). Then click *add program*.

Now, your IDE will be populated as shown in Figure 2:

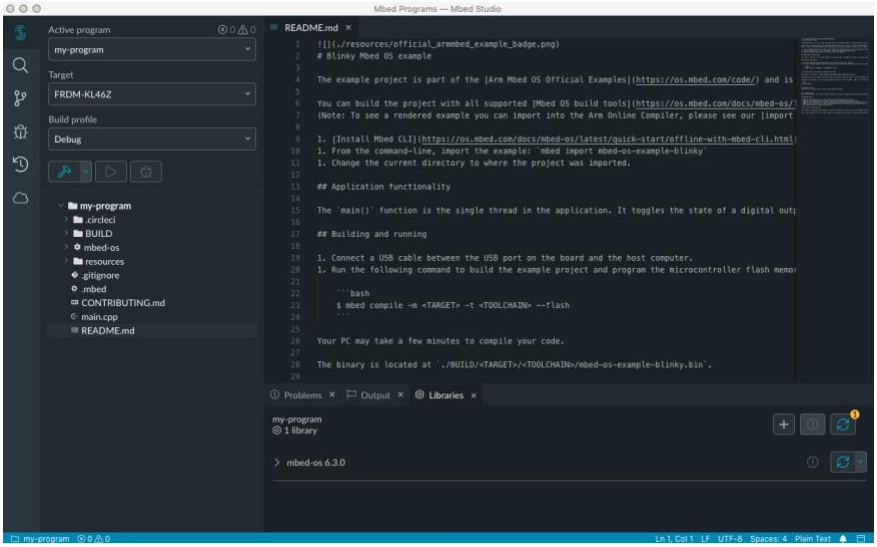


Figure 2: Mbed Studio with Program Loaded

My-program appears as an expandable hierarchy of files. If you try expanding some of these you will find a bewildering hierarchy of files – so don't bother. You will notice that README.md has appeared in the editor area. It's worth reading to see what is going on but don't worry if it doesn't all make sense. Once you've read it click on the X on the tab above to shut it.

The most important file in the hierarchy (from your perspective) is main.cpp (C++) and this is where the main() function for this program is stored. When the compiled program runs on the FRDM-KL46Z board – this is where the program starts. To edit main.cpp, just click on it in the file hierarchy. This will open it up in an editor.

The file you see is simple but still bears some investigation add it is shown as you see it in MS in Figure 3:

```
/* mbed Microcontroller Library
 * Copyright (c) 2019 ARM Limited
 * SPDX-License-Identifier: Apache-2.0
 */

#include "mbed.h"


// Blinking rate in milliseconds
#define BLINKING_RATE     500ms


int main()
{
    // Initialise the digital pin LED1 as an output
    DigitalOut led(LED1);

    while (true) {
        led = !led;
        ThisThread::sleep_for(BLINKING_RATE);
    }
}
```

Figure 3: Simple Program

You will notice that different elements are in different colours and this helps you to recognise different elements. Notice that `main()` has no parameters: this is a program that will run on an embedded system (there's no command line from which to pass in parameters). Also notice that there is a `while(true)` loop in `main()` – `main()` runs forever. If you exited from `main()` there is nowhere to go back to so every program you write should behave in this way.

`#include "mbed.h"` ensures that you have access to all that mbed has to offer and there is a huge range of support for writing useful programs and accessing facilities that the processor offers.

The first line in the main() function is the instantiation of an object of type `DigitalOut`.

```
DigitalOut led(LED1);
```

`DigitalOut` is a class – that is a collection of functions and data. This declaration is creating an object of this type (called `led` in this case). Each instance of `led` essentially has access to all of the functions in the class but all of

the data it has access to is related to this individual instance. In this case, when we create the object, `led`, we also need to say which pin on the processor this `DigitalOut` object is associated with. In this case, the pin is `LED1` (in actuality this is mapped to a pin which, on the FRDM-KL46Z, connects to a green LED). When you create this instance, it defines the corresponding pin as a digital output and allows you to write values to the pin (setting its voltage to either Vdd or 0V) directly.

led is an object of type `DigitalOut` and we would normally expect to access the functions associated with an object via statements like `led.function()`, where `function()` is one of the functions that forms a part of the class. However, `led` can be used as a variable within any program statement (this is implicitly equivalent to `led.write(value)`). If you read from `led` it returns the value that you last wrote to the pin (this is equivalent to `val = led.read()`). If you write to `led` then you are setting the value of the pin to logic 0 (0V) or logic 1 (Vdd) – it's that simple.

So what does the rest of the program in the while loop do. Firstly, the statement:

```
led = !led;
```

reads the current value being output to the LED1 pin, inverts it, and writes it back – so if the green LED was on, it is now off and vice versa.

The next statement does require a bit more understanding. Essentially it is something that causes the program to wait – but in a particular way.

```
ThisThread::sleep_for(BLINKING_RATE);
```

and, at the top of the file, `BLINKING_RATE` is defined as 500ms.

The simple explanation is that this statement causes `main()` to go to sleep (to stop) for 500ms and then start again.

So, the program changes the state of the LED, stops and waits for 500ms and then does it all again. When we run the program we will see that the green LED blinks at a rate of 1Hz.

The longer answer is that Mbed OS (essentially a simple RTOS – Real Time Operating System) can be multi-threaded. That is we can think of each thread as a program that runs in parallel with all of the other threads. `main()` is a single thread that is run from the get-go (you can add other threads from here, of course, to create a parallel program). Given that `main()` is a thread and a thread is an object of type `Thread`, one of the functions associated with the `Thread` class is `sleep_for()` and the nomenclature for accessing the functions associated with the class the code is a part of is `ThisThread::function_name()`. So to make the thread you are in go to sleep the statement above is require. If you created another thread, e.g. `extrathread` via:

```
Thread extrathread;
```

from within `main()` then you could control this thread from `main()` via:

```
extrathread.sleep_for(BLINKING_RATE);
```

and this would cause the thread you had created and (presumably) set running to go to sleep for 500ms.

In any real-time system, making threads go to sleep when they are not being used is **really** important because this allows other threads to run and prevents threads from being blocked.

Now, there is a function called `wait(time)` that does just that. You call the function, it returns after the specified time and the program continues. Why not do it this way? The answer is because the thread doesn't stop and this prevents other threads from running – it's really bad programming in a real-time program so don't do it.

Classes Available

How do you know what kind of classes you can use to make your programs work. Easy – look at the documentation. You will find the documentation at:

https://os.mbed.com/docs/mbed-os/v6.15/introduction/index.html

There's a lot of it so we can just jump directly to the area we're interested in – the available classes:

Firstly, click on *API References and Tutorials* in the left hand pane (API is Application Programming Interface and it described just that – how you interface to the Mbed OS applications).

Now click on *Full API list* . As you scroll down the page, you will note that it has collected different categories of classes together: RTOS, Event Handling, Drivers (we're interested in these), Platform (we're also interested in some of these under Time), Data Storage, Connectivity. There is a pretty comprehensive list of things we would need to create solutions!

Find `DigitalOut` under Drivers-> Input/Output Drivers and click on it. The page that opens, details the class' functions so you can understand what is going on and usually there is a use case example at the bottom (very similar to our program, for this class). This documentation will be vital to allow you to add new elements to your program to make it do useful things.

Remember, classes can be quite difficult and make your head spin a bit but using these classes to make your program do something useful and to interact with the hardware within the processor is usually pretty straightforward.

Running your Program

So, now we've got the program we want (we'll use the one we've got but generally we produce a program by editing `main()` and adding other functions). How do we run it on the FRDM-KL46Z board.

Firstly, plug the USB cable into the FRDM-KL46Z board. There are two USB mini connectors on the board – plug it into the one labelled OpenSDA (printed on the back of the PCB) – marked SDA Debug in the diagram in Figure 4.

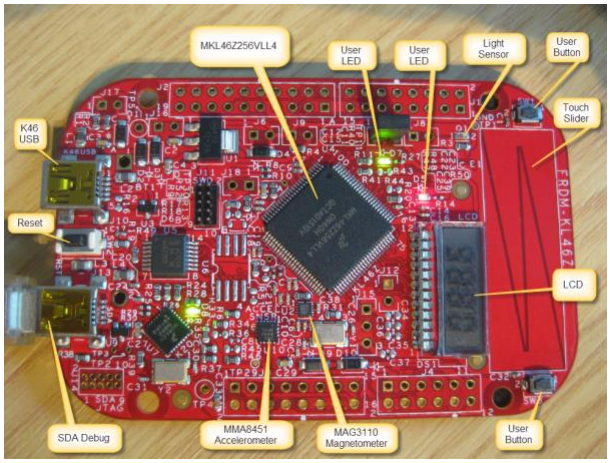


Figure 4: View of FRDM-KL46Z PCB

Now plug the other end of the USB cable into your computer. I'm using a macbook pro that only has Thunderbolt connections but I got a widget with my Samsung phone (which has a USB-C connector – partly compatible with Thunderbolt) that converts a USB-C male connector into a USB 2 female connector and this seems to work fine – if you don't have a USB 2 connection then you may need something similar.

Once you plug the board in then it should show up as a disk called DAP LINK (if it doesn't there's more work to be done and you would need to look at https://armmbed.github.io/DAPLink/

Now, your MS window will have changed slightly as in Figure 5:

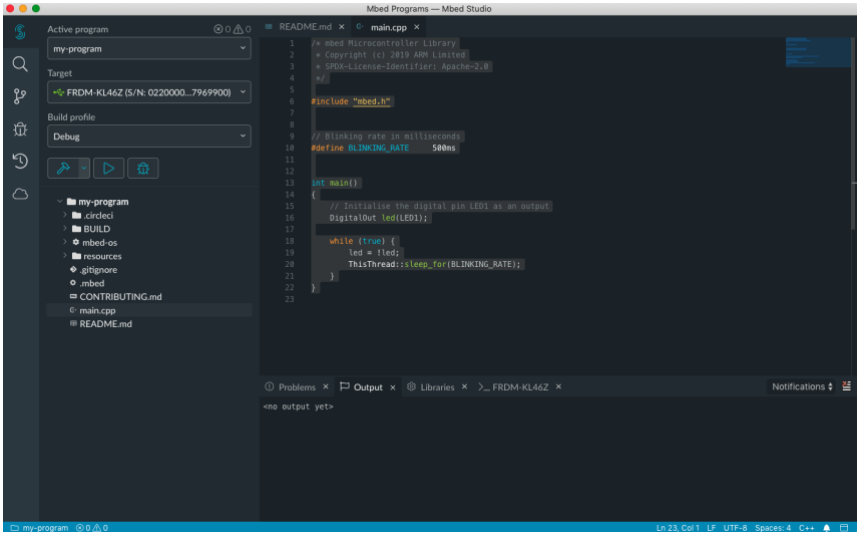


Figure 5: MBed Studio with a FRDM-KL46Z plugged in

You will notice that the information under *Target* identifies that it had selected the FRDM board that you've plugged in (serial number, etc.). You will also notice that the two buttons above the hierarchy (the one with a triangle on it – which is to run a program, and the one with the ladybird on it – which is to debug a program) have now turned blue – they've been activated. SM understands that the board is attached is as able to use it.

The additional button with the hammer on it is the build button. Without making any changes to the program, click on the *Build* button. The output display should show the output and a whole bunch of compile messages should scroll up. It will take a couple of minutes to build the whole program the first time you do it but, thereafter, it will be a lot quicker. Finally it will link the program together and it will display a table showing how much space the program occupies. If the program compiles and links properly then you should get a message telling you this. If there are any errors then click on Problems – they will be listed here.

Now your program is ready to run (the binary file is actually stored in Mbed programs/my-program/BUILD/KL46Z/my-program.bin but you probably don't even need to know this). To load the program on to the FRDM-KL46Z board, click on the *Triangle* button. The program will rebuild (pretty quick after the first time) and then it will download it on to the FRDM board. You can tell this is happening because, during the download, a green LED (towards the bottom LH side of the image of the FRDM board, above) will flash quickly. One it has finished, the program will begin to run. You will know it is running because the green LED (labelled user LED) in the image above will flash at 1Hz.

Let's change the program, slightly. Go to the editor and change:

```
#define  BLINKING_RATE    500ms
```

to:

```
#define  BLINKING_RATE    250ms
```

Now, rather than pressing the *Build* button, just press the *Triangle* button. You don't even have to save main.cpp – it does that for you. When the program starts running again, the LED will blink at 2Hz – success!

Even if you disconnect the FRDM board from the computer and then reconnect it, the program will start running immediately because it has been stored in the processor in non-volatile Flash memory.

Debugging

Clearly, this is a very simple program and we can see that it is working. The problem is that when a program is not working we need to see what is happening inside it. There is simple way to do this and a more complex way.

Generally, embedded programs don't have a user interface and so getting information out of them is difficult. However, whilst the FRDM board is connected to your computer, this is your user interface.

Printf

So, to get information, you can put `printf()` statements in your code. Let's do this simply:

Within the main function

Add a integer definition at the top of the function:

```
int i = 0;
```

now, in the while loop after the sllep_for() call, put in:

```
printf("loop %d\n", i++);
```

This will print 'loop <value of i>' and move the cursor to a new line. Then i will be incremented.

If you compile and load this on to the board, the only additional thing you will see is the LED that flashed during loading will be flashing as well.

However, go to the *View* menu in SM and click on *Serial Monitor*. This will open up a new output window where it will display any text it receives from the FDRM board and you should see a new message e.g. loop 34 appearing every time the green LED changes state.

You can actually do more complicated things with printing and debug. Look in the documentation under Debugging and Testing -> Debugging -> Methods -> Debugging using printf() statements. Here you will see methods for enabling printf messages which are then automatically excluded when you move from debug to production software and only send messages under certain conditions.

Clearly you can use `printf()` statements to work out where the program has got to and what variable values are but remember a call to printf does take time and can affect a program's behaviour.

Debugger

The processor actually has a built in debug block and this allows more complex debug. Let's leave the code as it is with the `printf()` statement in it. Now, rather than pressing the *Triangle* button, press the ladybird *Debug* button.

This will reload the program but it then changes the display as shown in Figure 6. Firstly, your `main()` function appear with the first statement (the `DigitalOut` define) highlighted and a small yellow marker in the left hand margin. The left hand column has changed to show you the threads that are running, the stack showing the sequence of calls, the variables that are visible at this point in the program and some other stuff.

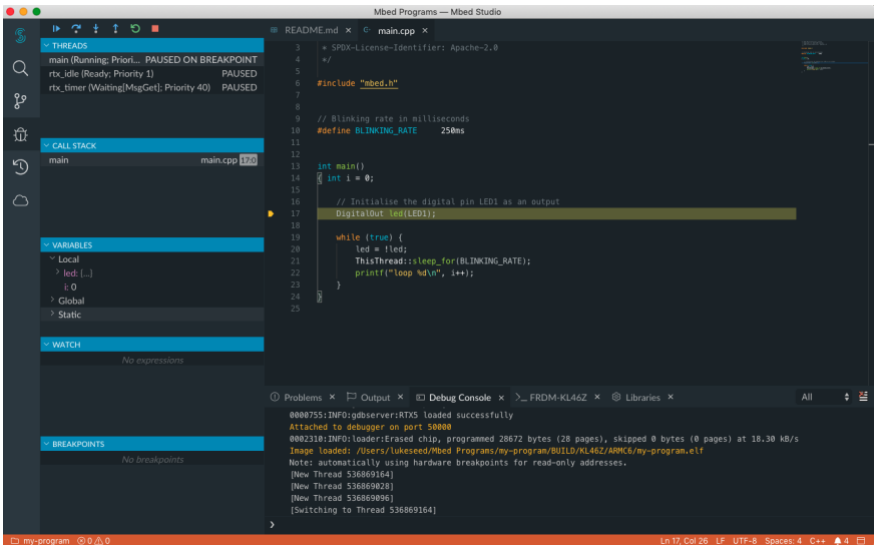


Figure 6: Debugger View in Mbed Studio

You will also notice that your program is not running – the LED is not flashing. You program has run up to the point highlighted in the editor (the beginning of `main()` and the debugger has stopped your program.

If you click on Local under VARIABLES then you will see two things that are local (defined and visible only within `main()`): `led` which is class and if you expand this you will see a whole bunch of information; and `i` which has its initial value of 0. So you can use this to work out the value of variables but only when the processor is stopped.

At the top of the LH pane there are 6 buttons and from left to right these are:

Continue to Run – this just sets the program running again;

Step over (causes the program to run a function that is where the program has stopped and then stop when the function completes);

Step into (causes the program to step into a function that is where the program has stopped and then stop at the beginning of the function);

Step out (causes the program to run though a function that the program is in and then stop when the function completes);

Restart (go back to the beginning of the program);

Stop (closes the debugger).

If you click on *Continue to Run*, your program will just start running, the LED will flash, and the text will display in the serial monitor. Additionally, the *Continue to Run* button changes to *Pause* (which will stop the program). Click on *Pause*. The program stops and wherever it stops, the program file where the program actually stops (some random place in the program) will appear in the editor and the next statement to be executed will be highlighted. However, every time you start and pause the program, you will end up in some random place over which you have little control. How do we avoid this?

Click on the *Restart* button to go back to the beginning. Now move the cursor down the left hand margin of the editor window so the cursor is against the `printf()` statement. You should see a dim red circle appear. Now click and the red circle should stay constant. This is a breakpoint. If you set the program running, if it reaches the point where a breakpoint has been set it will stop the program before the statement on the line has been executed. Now, click on *Continue to Run*. The program will run to the point where the breakpoint is and stop again. Note, the value of `i` (under VARIABLES/Local) is 0 (still) because the statement where `printf()` is called and `i` is incremented have not yet been run. Additionally, the green LED is off. Now click *Run to Continue* again. The program continues from where it stopped and then stops again when the breakpoint is *next* encountered. Now, the value of `i` is 1, the message 'loop 0' has been written to the serial monitor and the green LED is on. Every time you click *Continue to Run* the program goes round the loop once. It is clear that a debugger is a powerful tool for running and probing the behaviour of a program. You can add multiple breakpoints and there's a whole bunch of other things that the debugger will do.

Click on *Stop* to get out of the debugger.

More Complex Programs and Libraries

This program is pretty simple: it's got one C++ file holding your code. For more complicated programs you may need a number of C++ source files. You can add a

new source file into your design easily. Hover over the name of your program in the file hierarchy, right-click and Select *New File*. In the dialogue that opens, give your file a sensible name (eg. myfile.cpp) and click on *OK*. This adds the file to your program and opens the empty file so you can edit it.

The other thing you might want to do is add an external library that contains useful code. Whilst MBed OS does a lot, it does not cover everything. One thing that it does not cover is the LCD display. Again, right click over your program and now click on *Add Library*. This brings up a dialogue where you have to add the Library's URL and name. Now this is a part that works less well than the browser based tools but let's try to do this.

Go to a web browser and enter:

https://os.mbed.com/search/repository

This repository is where people have uploaded loads of Mbed libraries and programs.

Type 'SLCD KL46Z; into the search field and search.

This brings up a whole host of stuff. Find the one that says Erik – / SLCD. You cannot miss it, it has a Minion icon. Click on *SLCD*.

This opens up the page for the SLCD library. Just copy the URL in the browser:

https://os.mbed.com/users/Sissors/code/SLCD/

Back in SM, paste this into the Git or os.mbed.com URL field. The library name should populate SLCD – if it doesn't then just fill this in. Now click on *Next*. In the next dialogue, just select Default in Branch or tag and click on *Finish*.

Now you will see a new element in your program hierarchy called SLCD with a cog next to it. Expand it and have a look at its contents. The main files are SLCD.cpp (where the SLCD class is defined) and SLCD.h which is the include file for the class. If you look at SLCD.cpp, you will see a bunch of functions defined in it (which make up the functions of the class):

`SLCD()` – the constructor;

`_putc()` – links to the stream class and allows you to use `printf()` with this class

`Write_char()` – actually writes a character to the LCD

`Home()` – moves the cursor back to the LH side of the LCD display

`Clear()` – empties the display and homes the cursor

`Contrast()` – sets the contrast of the display

`DP()` – prints a decimal point at the defined position;

`Colon()` – switches the colon on or off.

`blink()` – makes the display blink

The LCD has only 4 characters and as you write to it a cursor moves from left to right. When you write beyond the end of the 4 digit display, it just goes back to the beginning again. It's very simple.

Now we've got the LCD library in the program, we can use it.

Access the class by putting `#include "SLCD.h"` at the top of main.cpp (just after the `#include "mbed.h"` statement.

Now, define an object of type SLCD by entering:

```
SLCD my_display;
```

Put this under the `DigitalOut` definition.

Now we need to write to the display. Put the following statements in the loop, below the `printf()` statement:

```
my_display.Home(); // moves the cursor back to the
beginning of the display
```

```
my_display.printf("%d", i); // prints i
```

and that's it.

Now try load and run the program.

Unfortunately, there is a problem (which you will encounter in a number of places): since the library was written, Mbed has changed. In some cases the changes are so great, that a library just won't work. In this case, the fix is simple.

Open up SLCD.h (which is where the error occurred). At the top under the include mbed.h statement, put in `#include "Stream.h"`

This fix just makes the Stream.h file visible (it's needed in SLCD). In earlier times, this file was included in mbed.h but not now for some reason.

Now when you load and run it, in addition to doing all the things it has been doing, it should also be printing the value of `i` to the LCD. Obviously, this will only work properly for 10000 seconds because once `i` > 9999 the display will not be big enough to show the number.

Useful Classes

There are some indispensable classes that will make programming a real-time system straightforward and chief amongst these are classes that deal with interrupts.

Interrupts are internal and/or external events that can be used to trigger a function within your program. The key thing about an interrupt is that it is a hardware mechanism and your program does not need to spend time checking for it – when the event occurs the hardware recognizes the fact and stops your program doing whatever it was doing and makes it run the appropriate function. When it's finished, the program just carries on doing what it was doing before the interrupt occurred. I cannot emphasise how important interrupts are.

An external event might be an input changing state and there is a class, `InterruptIn` for dealing with this.

An internal event might be a timer expiring an, again, there is a class for this, `Ticker`.

Let's look at `Ticker` in more detail and adapt our program to do what it has been doing but using a `Ticker`.

To do this we are going to edit the file as shown in Figure 7:

```
/* mbed Microcontroller Library
/* mbed Microcontroller Library
 * Copyright (c) 2019 ARM Limited
 * SPDX-License-Identifier: Apache-2.0
 */

#include "Ticker.h"
#include "mbed.h"
#include "SLCD.h"


// Blinking rate in milliseconds
#define BLINKING_RATE     250ms
#define SLEEP_PERIOD      5000ms

// Initialise the digital pin LED1 as an output
DigitalOut led(LED1);

Ticker my_timer;

void UpdateLED(void)
{
    led = !led;
}

int main()
{ int i = 0;
    // define the LCD class
    SLCD my_display;

    // set up the ticker to call UpdateLED every 250ms
    // once we've done this it just happens in the background
    my_timer.attach(UpdateLED, BLINKING_RATE);

    // we'll still do the stuff in this look but only once every 5s
    while (true) {
        ThisThread::sleep_for(SLEEP_PERIOD);
```

```
        printf("loop %d\n", i++);
        my_display.Home();
        my_display.printf("%d", i);
    }
}
```

Figure 7: More Complex Program

You will note that altering the led value is now in a void/void function `UpdateLED()`. You cannot pass parameters into this function or get values out because your program never calls it. To make this work, I've moved the definition of `led` outside of any function.

I've also defined a `Ticker` called `my_timer`.

Now the loop in `main()` still sleeps (but now for 5s) and every 5s it wakes up, prints loop i, and writes `i` to the LCD.

The only additional thing is the statement above the loop (which is only called once).

```
my_timer.attach(UpdateLED, BLINKING_RATE);
```

This says automatically call `UpdateLED()` every 250ms. Once this function is called, thereafter, every 250ms, whatever is happening in the main loop (even if it is not sleeping) is interrupted, `UpdateLED()` is run once and the LED changes state.

Now if you run this program you see that it is all happening.

At this point, you know enough to get writing programs!

Web Browser Tools

ARM offers this approach to allow software development via online tools and a web browser. You can access the tools via the web browser and the software runs on remote servers. This gets around issues of compatibility, installation, etc. because all that is needed to do the software development is a web browser.

To access these tools, open a web browser and go to:

https://os.mbed.com/

and click on the Complier button at the top.

You will be asked to register to access mbed (if you haven't done so already) and this is a pretty straightforward process (email, username, password) but you will need access to email to confirm the account that you set up.

Once you have set up an account, you can log in from this page at any time. Moreover, by adding the FRDM-KL46Z platform to your account you can do this from a variety of places. If you click on Hardware -> Boards at the top of the os.mbed.com page, this takes you to a page where you can see all the development boards supported by Mbed. Type FRDM into the search field, click on FRDM-KL46Z from the set of board that appears and, on the page, that opens, click on *Add to your Mbed complier*.

When you have registered, added the board, and the mbed compiler opens, it will appear as in Figure 8:

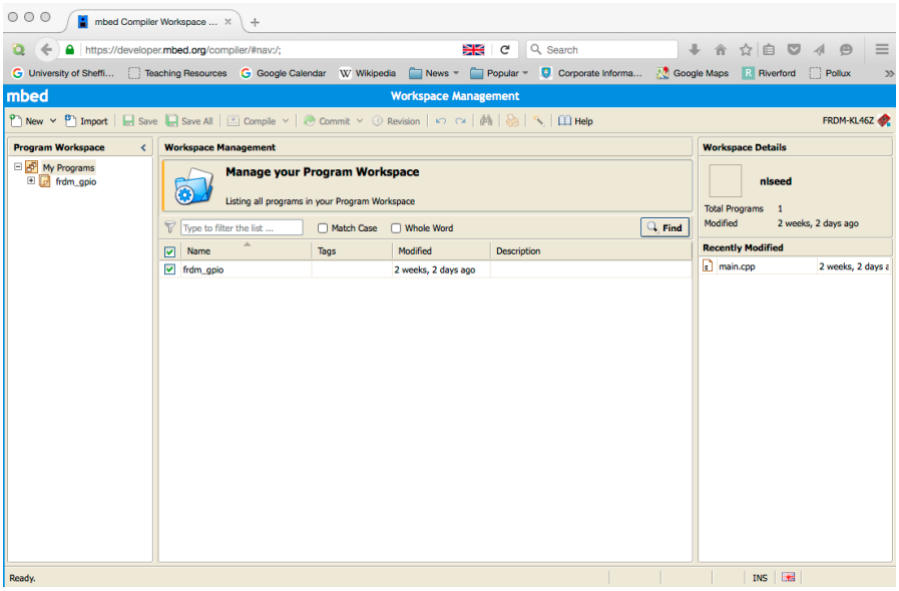


Figure 8: Mbed compiler (on OS X)

It is similar(ish) to SM. The left hand pane is your workspace and each program is a separate (expandable) item. In this case there is only one example program called frdm_gpio. If you look on https://developer.mbed.org/platforms/FRDM-KL46Z/ you will see that you can download example programs from here (e.g. *frdm_gpio*) and this is often a good starting point to get yourself using the software. If you click on an example program and then click on the *Import this Program* button then this will lead you through importing the program into your workspace. This is very easy. You can also import items into your compiler by clicking on the *Import* button on the toolbar of the mbed compiler (see Figure 8).

Once you have imported a program, click on + against its name and expand it to view the program's contents. You should see main.cpp (the main program file) and mbed which is expandable. Expanding mbed you should see Classes and expanding Classes will allow you to see all of the classes that are available for you to use – by default for the FRDM-KL46Z (this may not work – I find it's a bit erratic but the classes are still available to use and are described in the documentation as detailed in the section for SM). However, you should be aware that this set of classes does not cover the full functionality of the processor. In Figure 9, this has been done and the `DigitalIn` class has been selected. This opens up a file in the main part of the window, allowing you to see the class members and examples of how to use the class. Generally it is very easy to use resources covered by the mbed classes.

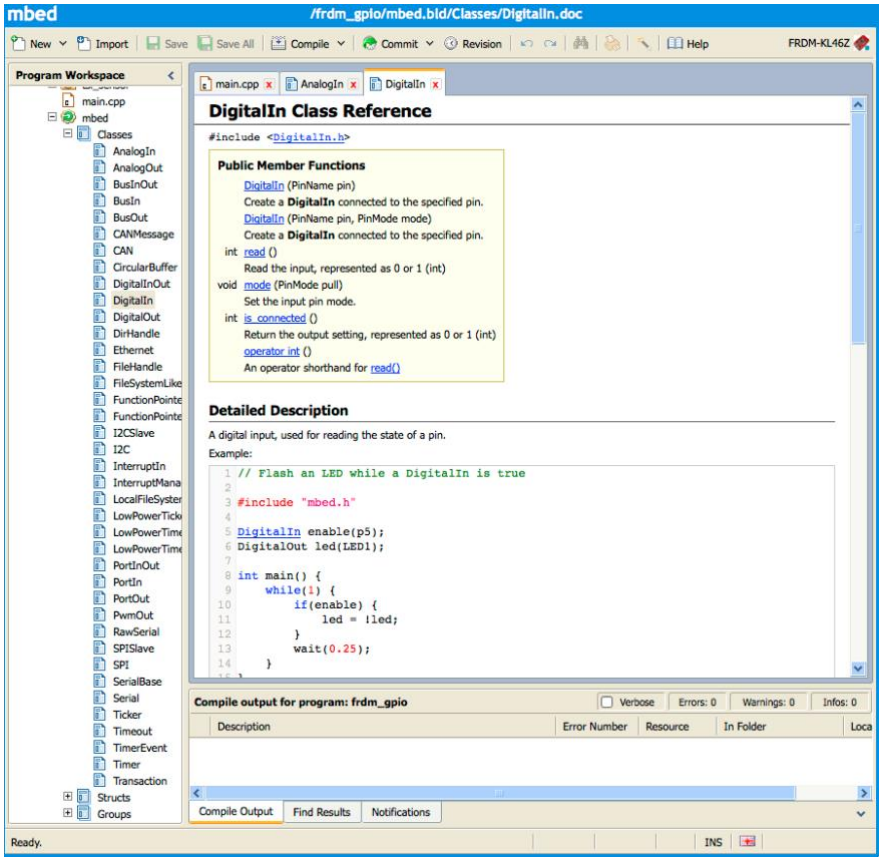


Figure 9: Mbed classes

In the example code, defining a digital input called `enable` that is connected to pin `p5` is as simple as `DigitalIn enable(p5)`. Thereafter, `enable` can be used as a variable (which can be read but not written to).

It is clear that there are lots of classes, each supporting a different aspect of activity. This approach is very simple and will allow you to build applications quickly.

The base classes, however, do not cover everything. For example, there is no support for the LCD display. To find other classes or programs, click on the Import button on the toolbar at the top of the window. This action opens a wizard and, from this wizard, click on the Library tag ad enter a search term e.g LCD. This will search the *mbed* repository for any matching names. In this case a number of matching libraries will be found but the one that we are interested in is SLCD (which is tagged as being for the KL46). Select the row to mark it and then click on the import button. On the import dialogue, select your active program in Target Path and then import. This library should now appear as an expandable element within your program.

Expanding *SLCD*, *Classes* and selecting *Classes* will allow you to see how to access the LCD as in Figure 10:

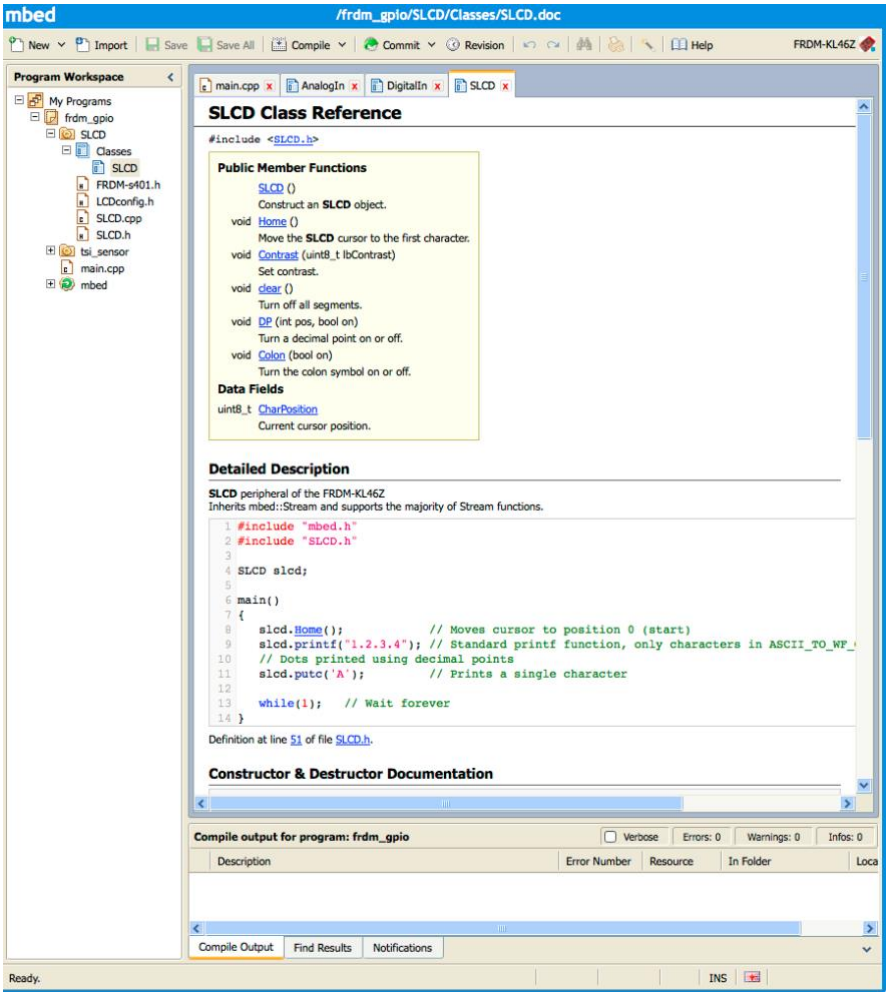


Figure 10: SLCD Class

This is more straightforward than in SM.

Building a Program

Once you have written a program, click on the Compile button on the toolbar and the application will be built. If there are errors, these will appear in the console pane at the bottom of the window. If the build is successful then the program-name.bin file will be downloaded to your local machine and this .bin file could be downloaded to the FRM-KL46Z development board and executed (when an mbed development board is plugged into a PC it will appear as an external disk and dragging-and-dropping a .bin file to the DAP LINK disk and pressing the reset button on the PCB will execute your program). However, the basic mechanism for debugging programs is quite basic (but sufficient).

Debugging

All that is possible without MS, is to set up a communication link from your program to communicate with a terminal application on the host computer. You can send `printf()` statements in your program to send debug information to the computer. Whilst this is workable, it does not really give you a lot of access to what is happening on the KL46F.

To do this you can use prinf() statements as described in the section on SM.

So, for example:

```
printf("Hello World\n");
```

Do bear in mind, that, in this example, 12 characters are transferred – a total of 120 bits and at 9600 baud this will take 12.5ms during which you program will be occupied. So the trick is not to transfer too much data.

You can build these statements into your code, allowing you to identify when the program has reached particular points, or to report on values e.g.

```
printf("MyVar=%4d\n", myVar);
```

which will print:

MyVar= 3

where 3 would be the current value of MyVar, right justified in a 4 character wide field.

Terminal Emulation

At the computer end it depends if you are using OSX or Win:

Win

For Win, you will have to find a terminal emulator. PuTTY is one such (which can be downloaded from Blackboard) but there are loads – see:

https://www.puttygen.com/windows-terminal-emulators

Be warned some terminal emulators that support encryption are illegal in certain countries!

You will need to find (via Control Panel) which COM: port is associated with the serial channel that is overlaid on the USB link.

You will then need to create a session on the Terminal Emulator that is plain serial, associated with the COM: port you identified in the last step and ensure that the channel is 9600 baud, 8 data bits, 1 stop bit, no parity.

OSX

For OSX, the terminal is already built in (run screen from a terminal). However, you may need to install a driver to access the USB link used by mbed.

Open a terminal and type in:

```
ls /Dev/tty.usb*
```

This will bring up a set of matching devices. Your serial channel will be there e.g. /Dev/tty.usbmodem01234

Now, type in using the name that appears, eg.

```
screen /Dev/tty.usbmodem012345
```

This opens up a serial monitor and, again any text from your program will appear here.

A Bit More Detail About a Peripheral Block

Whilst you probably do not need to view things in too much detail (the classes hide most of the underlying detail), it might be instructive to consider how some

of the peripheral blocks work. We will look at the LCD as an example. LCDs are quite complicated to drive (intrinsically the waveforms needed to drive an LCD display are not digital) but, intrinsically, once you understand what is going on, it is quite straightforward.

The display on the PCB is driven using 12 signals – see the diagram in Figure 11.

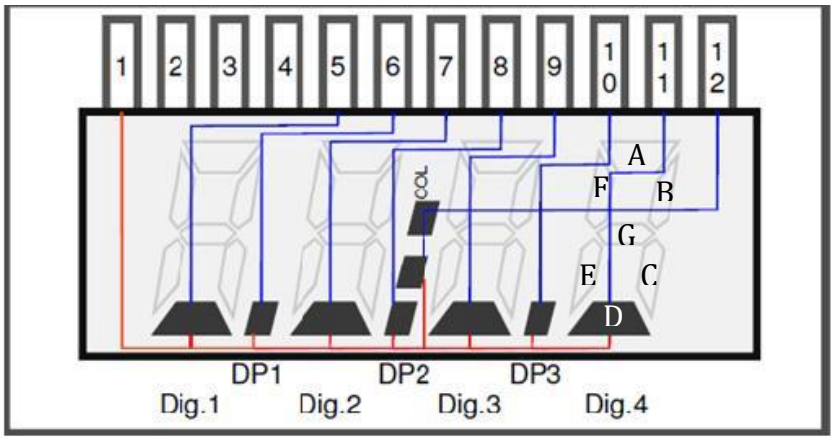


Figure 11: LCD

Each character is composed from seven, separate segments with intervening decimal points and a colon between the middle two characters. There are four *backplane* or *col* signals (pins 1 to 4) and 8 *foreplane* signals (pins 5 to 12). In the diagram in Figure 11, the foreplane signals are coloured blue and the col1 signal is labelled red. Essentially, the system should cycle repeatedly energizing signals *col1* through *col4* in turn. At the same time, certain *foreplane* signals will be energized is synchronism with *backplane* signals. When a *col* and a *foreplane* signal are energized, the corresponding segment will go black. So in this case, when *col1* is energized and pin 11 is energized, segment D of Digit 4 will switch on.

To allow all of the segments to switched on, pin 11 controls segments F/G/E/D of Digit 4 whilst pin 12 controls segments C/B/A/COLON. So, if pin 11 is energized then when *col1* is energized the segment D of Digit 4 is switched on, if *col2* is energized then segment E is switched on, if *col3* is energized, segment G is switched on and when *col4* is energized, segment F is switched on. If pin 12 is energized then when *col1* is energized the colon is switched on, if *col2* is energized then segment A is switched on, if *col3* is energized, segment B is switched on and when *col4* is energized, segment C is switched on. This is repeated for the other pairs of *foreplane* pins. That, is pins 4,5 control Digit 1, pins 6,7 control Digit 2, and pins 8,9 control Digit 3 (where the decimal point is substituted for the colon). So, by setting up sequences of signals on the 12 pins, the display can be controlled.

The LCD controller on the processor will support much more complicated LCD displays, with eight *backplane* signals and 56 *foreplane* signals – a total of 64 possible pins (some of which may not actually exist on any particular processor and in the case of this processor only 40 pins are available – some of which are also used for other functions) – and so part of the process of initializing the LCD is to identify which of the available processor pins are used to drive the LCD.

Looking at the PCB schematic, the 12 LCD pins (marked as nets *LCD_01*…*LCD_12* on the schematic) are allocated as shown in Table 1:

Net	Pin Name	Pin #
LCD_01	PTD0/LCD_P40/SPI0_PCS0/TPM0_CH0	93
LCD_02	PTE4/LCD_P52/SPI1_PCS0	6
LCD_03	PTB23/LCD_P19	69
LCD_04	PTB22/LCD_P18	68
LCD_05	PTC17/LCD_P37	91
LCD_06	PTB21/LCD_P17	67
LCD_07	PTB7/LCD_P7	57
LCD_08	PTB8/LCD_P8/SPI1_PCS0/EXTRG_I	58
LCD_09	PTE5/LCD_P53	6
LCD_10	PTC18/LCD_P38	92
LCD_11	PTB10/LCD_P10/SPI1_PCS0	60
LCD_12	PTB11/LCD_P11/SPI1_SCK	61

Table 1: LCD Pin Allocation

Behind each of the pins is an 8-bit register, the contents of which can be transferred serially out via the pins to generate the sequence of signals necessary to drive the LCD.

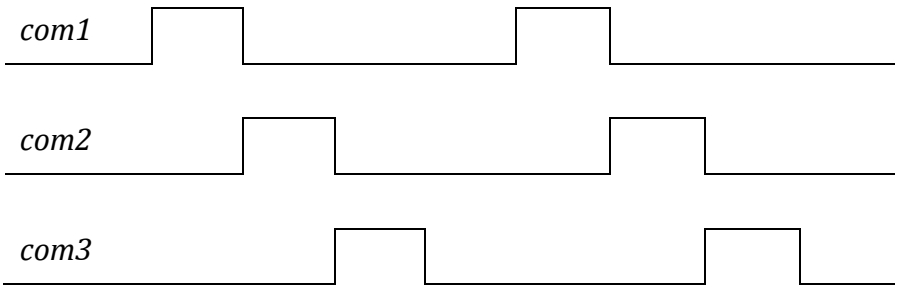
Looking at Table 1, *com1* (*LCD_01*) is connected to LCD_P40, *com2* is connected to LCD_P52, *com3* is connected to LCD_P19, and *com4* is connected to LCD_P18. The registers sitting behind these pins are WF40, WF52, WF19, and WF18. With only four backplane pins, the LCD block is programmed to serially output only the bottom 4 bits of the registers and so these registers are programmed as follows:

WF40 00000001
WF52 00000010
WF19 00000100
WF18 00001000

A similar process occur (in synchronism) with the registers attached to the *foreplane* pins. So, *LCD_11* and *LCD_12* are controlled by WF10 and WF11. So, if these registers hold:

WF10 00001100
WF11 00000110

And so these values generate *backplane*/*foreplane* waveforms as shown in Figure 12:



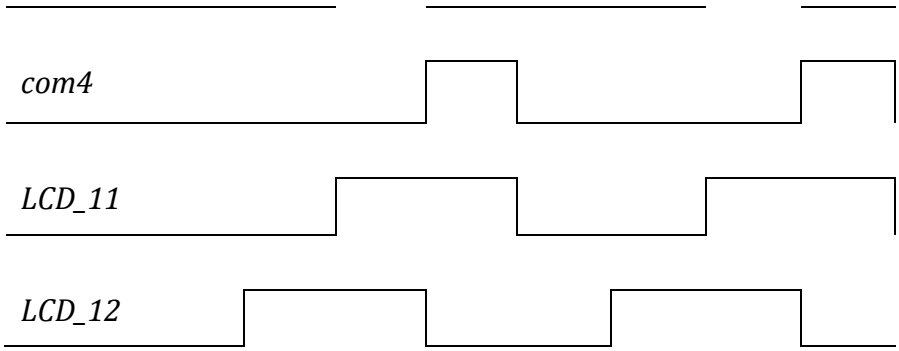


Figure 12: LCD Waveforms

As the bottom 4 bits are fed out repeatedly in sequence.

Consequently, segments G, F, C, and B will be switched on and Digit 4 will display '4'.

Initialisation

To initialize the LCD, to perform in this way is mainly a matter of setting up a few configuration registers with appropriate values and this is done by the class constructor for the LCD class.

Conclusion

As you can see, the process of driving the LCD could be quite complex if we decided to access the processor's registers directly and write the basic code to interface with the hardware. However, the mbed environment, in conjunction with libraries/classes that can be imported makes the process of writing even real time programs quite straightforward. You won't necessarily be able to get the maximum possible performance out of the processor using the mbed classes but it is a good starting point. More generally, the ARM Cortex M0 processor used on the FRDM-KL46Z PCD has a user manual that runs to circa 2000 pages and contains, literally, 100s of control registers. Whilst progamming these directly is possible (if you look in the SLCD.cpp file, for example, you can see that the LCD registers are being written to directly), using the classes allows you to leverage a huge amount of prior design effort and will leave you the time to actually write useful code.