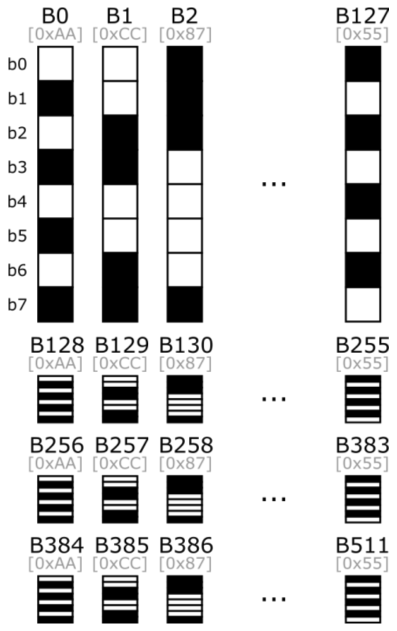
Exercise 7

For the next couple of exercises, we will be working with the LCD display on the mbed expansion board. We are dealing with a 128x32 monochrome dot matrix display, which means each pixel can be accessed individually and set to either on (black) or off (greenish-gray). The LCD is controlled through a Serial Peripheral Interface (SPI) bus which can provide serial communication to multiple devices simultaneously. Similar to the UART connection you use to send data to the PC, the LCD SPI connection is configured to transmit data one byte at a time. This leaves us with a bit of an issue: since we are dealing with a monochrome display it would be a waste of bandwidth to send a full byte of information for each single pixel. Because of this, the LCD has been split into 512 groups of 8 pixels which are updated simultaneously. This makes it a bit more tedious to change the value of a single pixel, but it allows for more rapid updating of the display. The figure below illustrates how the grouping is structured:



Each group consists of an 8-pixel vertical slice of the display, with the entire LCD being made up of 4 rows each containing 128 of these slices (a total of 512 slices). When writing to the display, you start in the top left corner moving right one slice with each byte of information. The least significant bit of each byte refers to the top-most pixel of each slice. This might seem a bit complicated, but it should quickly start making sense once you begin writing to the display!

Exercise 7.1 - LCD Communication

Configuring the SPI port and LCD is a bit of an involved process, so we have provided a couple of functions that take care of the most tedious work. These are included in the `30010_io.c/h` files you added to the project earlier, so you should already be good to go. The first function is `init_spi_lcd()` which sets up the SPI port and configures the LCD. This has to be run before doing anything else with the display. The second function, `lcd_reset()`, reboots and reconfigures the LCD and probably won't be of much use to you. `lcd_transmit_byte()` is used to send data and commands to the display and should only be used if you know what you are doing! Finally, `lcd_push_buffer()` transmits a byte array (size of 512) to the LCD and shows the data. This is the function you will use every time you want to update the display.

The LCD has an internal memory bank where it stores which pixels should be turned on or off which means that you only need to tell which pixels to activate once and it will keep them on. Unfortunately, we cannot update just a small section of this - we must overwrite it in its entirety. Because of this it makes sense to create a graphics buffer which is basically just a copy of the internal LCD memory, but with the important difference that we can update any part of it just as we like. When updating the LCD, we then first write to the buffer and then overwrite the LCD memory with the contents of the buffer (hence the name of the function `lcd_push_buffer()`).

- Initialize the LCD and create a graphics buffer (512 byte array).

Before we display anything on the LCD we have to put something in the buffer. A quick way to do that is using `memset()`:

```
memset(buffer,0xAA,512); // Sets each element of the buffer to 0xAA
```

- Put something in your buffer and send it to the LCD using `lcd_push_buffer()`.

Depending on what you placed in your graphics buffer, hopefully something should show up on the LCD, but it probably won't be anything too spectacular. The next step is the ability to write text!

Exercise 7.2 - Writing Text on the LCD

Before we can write any text we first need to define what the letters look like, i.e., which pixels should be turned on when a particular character is written. This is done with a character map. If you look on DTU Inside you will find just such a map ready for you.

- Go to DTU Inside, download the character map (`charset.h`), and import it into your project.

The character map is nothing more than a two-dimensional byte array, where the first dimension indexes each ASCII letter from 0x20 (space) to 0x72 (_). Note that the indexes in the character map are 0x20 lower than the corresponding ASCII value because we didn't include the first 32 characters. The second dimension contains data for each LCD slice, index 0 being the left-most, and each character being 5 slices wide. This means that there is room for 25(.6) letters on each line.

- Create a function that takes a string and a location (slice, line) and copies the data from the character map into the graphics buffer. You can call it `lcd_write_string()`.
- Create another function called `lcd_update()` but leave it blank for now.
- In your timer interrupt function from last exercise add code that sets a flag high every time the interrupt is triggered.
- Play around with adding time-based updates to the strings your show on the LCD. Note: you should do the updates in `lcd_update()` based on whether or not the flag is set. Also, remember to set the flag low after doing your update!

In the next exercise, we will use `lcd_update()` to get the text scrolling on the LCD!

Exercise 7.3 - LCD Text Scroll

We will get the text to scroll across the display one slice at a time. This is done by incrementing (or decrementing) an index variable every xth call to `lcd_update()`. You will have to decide on a value for x that you find suitable. When getting the text to scroll you have a few options: the first one is to simply move all the values in the graphics buffer over by one index, while a second one is to clear the array and redraw everything at an offset position. You'll have to figure out which one of the two (or a completely different method) works the best. Note that clearing the array can be achieved with memset().

- Make a function that scrolls text from the right to the left. When the text reaches the left-most edge, it should wrap back to the right edge on the same line.
- Make sure your function is able to handle strings wider than the LCD.
- Play around with different scroll directions and wrapping methods.

If you would like a bit of an extra challenge, here is an extra optional task:

- Make the text scroll vertically on the display one pixel at a time.

Exercise 8

In exercise 5 we played around with digital (on/off) input and output which allowed for simple user interactions. Now it is time to expand upon this and implement some analog (continuous) control. The way we are going to achieve this is by using the two rotational potentiometers that are mounted on the bottom of the mbed expansion board. These potentiometers are configured in two voltage divider circuits that are connected to GPIO pins PA0 and PA1 such that the voltage that the pins are exposed to will wary as the potentiometers are turned. If we setup pins PA0 and PA1 as ordinary digital input pins they will register as LOW until the pots have been turned by a certain amount after which they will register as HIGH.

We want something more! For that we will need the Analog to Digital Converter (ADC) of the STM32. You will find the “Analog-to-digital converters (ADC)” chapter (pp. 287) of [RM] useful for this exercise.

Exercise 8.1 - Analog-to-Digital Conversion

The specific microcontroller we are using has one 12-bit ADC with 16 channels available for us to utilize. 3 of the channels are connected to internal signals whereas the remaining 13 are connected to GPIO pins. Each channel is able to measure voltages between 0 V and 3.3 V in steps of [$3.3 / 2^{12}$ V] and will thus output a number between 0 and 4095 depending on the voltage.

If you take a look at the ADC chapter of [RM] you will quickly realize that the ADC has a number of fairly advanced features which might make it difficult to use, however, we will configure it for standard operation which simplifies things a lot! In the base configuration, the ADC is setup for: single-conversion (meaning a single measurement); 12-bit resolution; software triggering, as opposed to having an external trigger signal; 16-bit unsigned integer output with the 12 meaningful bits aligned to the right; and finally converting 1 channel at a time. We’ll start by configuring the GPIO pins.

- Configure pins PA0 and PA1 for ordinary input (NOT analog input/output). Hint: have a look at the comments in the code example of exercise 5.1. Also, remember to enable the clock line to port A!

The microcontroller will automatically figure out it needs to connect the pins the ADC when we perform measurements as long as the pins are configured for input. Now it's time to configure ADC1 (it's called that because some chip variants have more than one ADC).

First we configure the clock source:

```
    RCC->CFGR2   &= ~RCC_CFGR2_ADCPRE12;       // Clear ADC12 prescaler bits
    RCC->CFGR2   |=  RCC_CFGR2_ADCPRE12_DIV6; // Set ADC12 prescaler to 6
    RCC->AHBENR  |=  RCC_AHBPeriph_ADC12;      // Enable clock for ADC12
```

The first two lines configure the frequency of the clock that drives the ADC. The peripheral must be driven at a frequency below 30 MHz and is fed by the 62 MHz system clock running through a programmable prescaler. A prescaler division of 6 is chosen because we do not need to do extremely fast measurements so around 10 MHz should be more than sufficient (64/6 = 10.67).

Because the mode we will be using is the standard reset configuration, all we need to do is to reset the ADC:

```
    ADC1->CR    =  0x00000000; // Clear CR register
    ADC1->CFGR &=  0xFDFFC007; // Clear ADC1 config register
    ADC1->SQR1 &= ~ADC_SQR1_L; // Clear regular sequence register 1
```

That is the configuration completed, but before we can actually do any measurements we have to perform a calibration. Luckily the ADC has the ability to perform this automatically, but first we must enable the internal reference voltage source and wait a little bit:

```
    ADC1->CR |= 0x10000000; // Enable internal ADC voltage regulator
    for (int i = 0 ; i < 1000 ; i++) {} // Wait for about 16 microseconds
```

And finally perform the calibration:

```
    ADC1->CR |= 0x80000000; // Start ADC1 calibration
    while (!(ADC1->CR & 0x80000000)); // Wait for calibration to finish
    for (int i = 0 ; i < 100 ; i++) {} // Wait for a little while
```

The first line tells the ADC to start the calibration, while the second line waits until the calibration has finished. The third line tells the MCU to wait for a little while. This is needed because of some idiosyncrasies in the ADC design (see [RM] for more information). Then we enable the peripheral:

```
    ADC1->CR |= 0x00000001; // Enable ADC1 (0x01 - Enable, 0x02 - Disable)
    while (!(ADC1->ISR & 0x00000001)); // Wait until ready
```

Similar to the calibration; the first line enables the ADC, and the second waits for it to be ready to start making measurements.

Now we should be good to go!

At this point you probably have a fairly good understanding of how everything on the microcontroller is done by setting certain bits in certain registers to certain values. It can honestly get a bit tedious! So, to save some time, we will be using a couple of the built-in hardware library functions to read from the ADC.

First we tell the ADC which channel to read from, the rank of the measurement (important when doing multiple readings simultaneously), and the sampling time:

```
ADC_RegularChannelConfig(ADC1, ADC_Channel_1, 1, ADC_SampleTime_1Cycles5);
```

This tells the ADC to read from channel 1 (PA0), sets the rank of the measurement to 1, and the sampling time to 1.5 clock cycles.

Then we do the measurement:

```
ADC_StartConversion(ADC1); // Start ADC read
while (ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == 0); // Wait for ADC read
```

This consists of starting the measurement and then waiting for it to finish. Finally we can read the result using:

```
uint16_t x = ADC_GetConversionValue(ADC1); // Read the ADC value
```

- Create a program that configures the ADC and then performs measurements on PA0 (channel 1) and PA1 (channel 2). Note: you can find which channels are connected to which pins by looking at table 13 in [DS].

Now that we can perform measurements, we should try displaying them somewhere.

Exercise 8.2 - Writing Numbers

Back in exercise 1.3 we gave you a function to write fixed point numbers to the PC over the UART connection. The function used `printf()` to print out a formatted number. Let's have a closer look at that with the following example:

```
printf("Value = %02ld\n", val);
```

The '`%`'-sign tells the function that it should be expecting a number, the '`0`' that follows is a flag that signifies that we want to print it with leading zeroes, the '`2`' indicates that the number should be printed with at least two digits, the '`l`' specifies the type of the number to be a `long`, and the '`d`' denotes the type of number to print to be a signed integer. In general the syntax adheres the following format:

```
%[parameter][flags][width][.precision][length]type
```

Arguments in `[]`-braces are optional. Wikipedia has a nice overview of which values all the different arguments can take: https://en.wikipedia.org/wiki/Printf_format_string.

- Write the ADC measurements to PuTTY using `printf()`.

You should see the printed values change as you turn the potentiometers.

There is a similar function called `sprintf()` which can be used to write numbers to strings. An example:

```
uint8_t a = 10;
char str[7];
sprintf(str, "a = %2d", a);
```

This will result in `str` containing `"a = 10"`. The string contains 6 characters, however, because C stores strings as null-terminated char arrays an array of size 7 is needed to accommodate the '\0'-character.

- Use `sprintf()` to write the ADC measurements to a string and print them on the LCD display.

It is also possible to use `printf()` and `sprintf()` for writing strings as was done in exercise 6.2 – They are two quite useful functions!