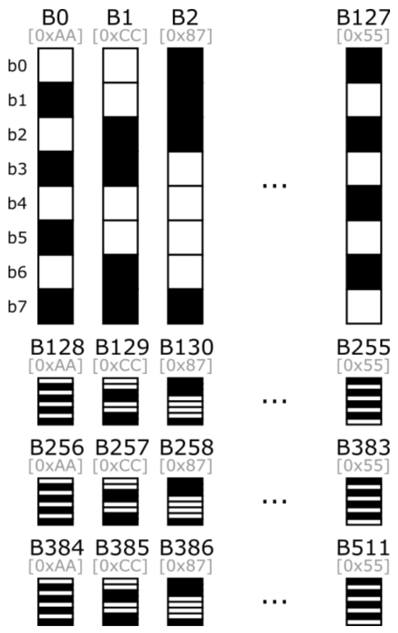
Exercise 2

In exercise 0 and 1, we got familiar with the STM32 development environment and we played around with digital GPIO, which allowed for simple user interactions. Now it is time to use the LCD on the MBED expansion board to display data.

Exercise 2.1 – Writing data to the LCD

In this exercise, we will be interfacing 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. The LCD has been split into 512 groups of 8 pixels which are updated simultaneously. This allows the display to be updated more efficiently. The figure on the right illustrates the grouping. 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.



Configuring the SPI port and LCD is a bit of an involved process so an LCD library (`lcd.h`) has been provided. 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 framebuffer. This means that it has a complete memory buffer that contains the state of each individual pixel on the LCD; whatever is written to the buffer will be seen on the LCD. Unfortunately, we cannot update just a small section of this - we must overwrite it in its entirety. Because of this, it is easier to create a local framebuffer that contains a copy of the LCD framebuffer, but with the important difference that any slice of 8-pixels can be updated at a time. When updating the LCD, we therefore first modify the local buffer and then push the contents to the LCD framebuffer (`lcd_push_buffer()`). A quick way to initialize the buffer is using `memset()`:

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

- Initialize the LCD and create a local framebuffer "`fbuffer`" (512 byte array).
- Put something in your buffer (eg. 0xAA) 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 2.2 - Writing strings on the LCD

When writing text to the LCD, it is necessary to translate each character into individual pixel slices. This is done with a character map (charset.h), included with the lcd library (lcd.h). 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 the first 32 characters are not included. 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 (excluding empty slices between each character), with four lines in total.

A number of text rendering functions are included with the LCD library. Using `lcd_write_string()`, a number of strings can be rendered to the framebuffer at a certain x and y offset.
NOTE: The x offset is in *pixels* while the y offset is in *lines*.

```
void lcd_write_string(uint8_t * str, uint8_t * lcdBuff, uint8_t x, uint8_t y);
..
lcd_write_string("1line", fbuffer, 10, 0);
lcd_write_string("2line", fbuffer, 0, 1);
lcd_write_string("3line", fbuffer, 10, 2);
lcd_write_string("4line", fbuffer, 0, 3);
lcd_push_buffer(fbuffer);
```

NOTE: In exercise 1.5, you used `sprint()` to print formatted text. By replacing the string ".." with a character array populated by `sprint()`, it is possible to use this function to print formatted text to the LCD.

- Print some text to the LCD and experiment with using the x and y offset.
- Print some formatted text to the LCD using `sprint()`.

Exercise 2.3 - FLASH Memory

The STM32F302R8 has a built-in 64 kB FLASH memory that is used to store your program. However, it is also possible to use it for storing data that will be saved even if the device is powered off. The FLASH memory is divided into 32 "pages" each containing 2048 bytes of information. When you want to write to a given address in the FLASH memory, you first have to erase the entire page containing that address. This means that you cannot write to the same location twice, without erasing the flash beforehand.

The FLASH memory starts at address 0x08000000 and ends at 0x08010000. If we only use the last page which starts at address 0x0800F800 to store our data, we will have 62 kB available for the program itself. That should be sufficient, but you still have to be careful to avoid overwriting the program stored in flash.

Reading from FLASH

Reading from the FLASH is fairly straight forward: we simply read from a specific address in memory and see what is stored there. Here is an example, reading from page 31.

```
uint32_t address = 0x0800F800;
uint16_t tempVal;
for ( int i = 0 ; i < 10 ; i++ ){
    tempVal = *(uint16_t *)(address + i * 2);  // Read Command
    printf("%d ", tempVal);
}
```

This piece of code reads the first 10 elements of data stored in the last page of the FLASH memory. Note that in this example, everything is stored as `uint16_t`, so if you have an 8-bit number you have to convert it and if you have a 32 bit number you have to split it up. The code works by incrementing the address variable for each item, typecasting it to a pointer to an uint16_t, and then looking at the value stored at the address. Note that the address is incremented by 2 because a 16-bit variable takes up two bytes.

Writing to FLASH

Writing to FLASH is a bit more involved, as seen in the snippet below:

```
    FLASH_Unlock();
    FLASH_ClearFlag(FLASH_FLAG_EOP | FLASH_FLAG_PGERR | FLASH_FLAG_WRPERR);
    FLASH_ErasePage( address );
    for ( int i = 0; i < 10; i++ ){
        FLASH_ProgramHalfWord(address + i * 2, data[i]);
    }
    FLASH_Lock();
```

First we unlock it by setting FLASH_Unlock. Now we are in the danger zone, where you can easily mess up your entire program! The two following lines are used to erase the page that we wish to write to:
Then we write our data. In this example we write the 10 elements contained in the data-array. Finally, we write protect the flash.

You can monitor the flash memory contents by selecting the menu “Debug | Debugging windows | Examine memory” and then selecting the base address of page 31 “0x0800F800”. Enable “Live updates” to ensure that the memory contents are updated automatically.

A library is available on DTU Inside. (flash.c/flash.h). This library provides functions for reading and writing uint16, uint32 and floats to the flash memory, based on the example. The snippet below illustrates how to use it for writing a floating point value or uint32. PG31_BASE represents the address of page 32 “0x0800F800” as defined in flash.h. Add your own page addresses to this file if necessary.

```
    tempfloat = read_float_flash(PG31_BASE,0);
    init_page_flash(PG31_BASE);
    FLASH_Unlock();
    write_float_flash(PG31_BASE,0,(float)1.0);
    FLASH_Lock();
    tempfloat = read_float_flash(PG31_BASE,0);
    ...
    tempval = read_word_flash(PG31_BASE,0);
    if(tempval!=(uint32_t)0xDEADBEEF){
        init_page_flash(PG31_BASE);
        FLASH_Unlock();
        write_word_flash(PG31_BASE,0,0xDEADBEEF);
        FLASH_Lock();
    }
    tempval = read_hword_flash(PG31_BASE,0);
```

- Use the flash library to write various variable types (uint8, uint16, uint32 and floats) to the flash memory. Use the “Examine memory” and breakpoints tool to verify the flash contents.
- Fill page 31 with incremental patterns (eg. 0x01010101, 0x02020202) and verify the contents. Remember that you only need to unlock/lock the flash once, if you are writing a continuous chunk of data. Also remember that you cannot overwrite an already stored value, without first erasing the whole page.

Exercise 2.4 – Analog-to-Digital Conversion

For the next exercise, we will be experimenting with the built-in ADC (analog-digital converter) and the potentiometers on the mbed expansion board. The two rotational potentiometers are connected between ground and +3.3V, with the center pin (wiper) connected to the GPIO pins PA0 and PA1 (i.e. as a voltage divider). The voltage measured at the two GPIO pins will therefore change as the potentiometers are turned. If we setup pins PA0 and PA1 as ordinary digital input pins, they will register as LOW until the input voltage exceeds the LOW->HIGH threshold of the STM32. To make things more interesting, we will use the Analog to Digital Converter (ADC) to measure the input voltage on these pins. You will find the "Analog-to-digital converters (ADC)" chapter (pp. 287) of [RM (STM32F302x8_Reference_Manual.pdf)] useful for this exercise. And we will use chapter 3 "Analog-to-digital converter (ADC)" in [SPL p. 46].

The specific microcontroller we are using has one 12-bit ADC with 16 channels available. 3 of the channels are connected to internal signals whereas the remaining 13 are connected to GPIO pins. Each channel can measure voltages between 0 V and 3.3 V in steps of [3.3 / (2^{12} -1) V] and will thus output a number between 0 and 4095 depending on the input voltage.

How to use this driver

1. select the ADC clock using the function *RCC_ADCCLKConfig()*
    a. The first line configures the frequency of the clock that drives the ADC. The ADC must be clocked at a frequency below 30 MHz and is fed by the 64 MHz system clock running through a programmable prescaler. The system clock can be divided by the following ratio: 1, 2, 4, 6, 8, 12, 16, 32, 64, 128, 256.
    b. A prescaler division of 8 is chosen because we do not need to do extremely fast measurements so around 10 MHz should be more than sufficient (64/8 = 8 MHz).
2. Enable the ADC interface clock using *RCC_AHBPeriphClockCmd();*
3. ADC pins configuration
    a. Enable the clock for the ADC GPIOs using the following function: *RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOx, ENABLE);*
    b. Configure these ADC pins in analog mode using *GPIO_Init();*
4. Configure the ADC conversion resolution, data alignment, external trigger and edge, sequencer length and Enable/Disable the continuous mode using the *ADC_Init()* function.
5. Activate the ADC peripheral using *ADC_Cmd()* function.
6. And wait for the ADC to be ready
    a. *while(!ADC_GetFlagStatus(ADC1,ADC_FLAG_RDY)){}*

Settings for the ADC (Hint: search for "ADC_InitTypeDef" in the SPL document [SPL] to find values for below.):

- single-conversion
- 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
- converting 1 channel at a time.

Before we can do any measurements, we must perform a calibration. The ADC can perform this automatically, but first we must enable the internal reference voltage source and wait for 10uS (worst case warm-up time). See section 15.3.6 in [RM] for details.

```
// set internal reference voltage source and wait
ADC_VoltageRegulatorCmd(ADC1,ENABLE);
//Wait for at least 10uS before continuing...
for(uint32_t i = 0; i<10000;i++);
```

And finally perform the calibration. See section 15.3.8 in [RM] for details:

```
ADC_Cmd(ADC1,DISABLE);
while(ADC_GetDisableCmdStatus(ADC1)){} // wait for disable of ADC
ADC_SelectCalibrationMode(ADC1,ADC_CalibrationMode_Single);
ADC_StartCalibration(ADC1);
while(ADC_GetCalibrationStatus(ADC1)){}
for(uint32_t i = 0; i<100;i++);
```

First the ADC is disabled and a while loop waits for the ADC to turn off, then the calibration mode is set where after the calibration is done, the while loop waits for the calibration to be done. The last line tells the MCU to wait for a little while. This is required because the ADEN bit cannot be set during ADCAL=1 and 4 ADC clock cycle after the ADCAL bit is cleared – i.e. after the calibration has finished (see [RM] for more information).

Finally, the ADC is enabled:

```
ADC_Cmd(ADC1,ENABLE);
While((!ADC_GetFlagStatus(ADC1,ADC_FLAG_RDY)){}
```

At this point you probably have a fairly good understanding of how everything on the microcontroller is configured. Next 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. See section 15.3.12 in [RM] for details. 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:

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

- Write a function that configures the ADC according to the above example. You can call this function `ADC_setup_PA()`.
- Write a function that performs measurements on PA0 (channel 1) and PA1 (channel 2). You can call this function `ADC_measure_PA(uint8_t ch)`. Note: you can find which channels are connected to which pins by looking at table 13 in [DS].

- Display both 12-bit measurements on the LCD display and update it at a fixed interval. You can use the timer to ensure that the measurement is updated at 10Hz.

Exercise 2.5 – Absolute voltage from ADC measurement

The ADC uses a reference voltage of V_{REF+}. On the 64-pin STM32F308 used on the Nucleo board, this reference cannot be connected to a separate external voltage reference but is connected directly to the analog power supply pin (V_{DDA}) so $V_{REF+} = V_{DDA}$ in this case. Furthermore, V_{DDA} is connected to V_{DD} through a ferrite bead, unless the solder bridge SB57 is removed. Thus, the ADC generates a digital value corresponding to the ratio between V_{DDA} and the voltage applied to the ADC channel (V_{CHx}). The value of V_{DDA} is not known, but for now we will assume that it is 3.3V

At this point, we have a pair of 12-bit values representing the analog input voltages. Each measurement ranges from 0 to 4095. For situations, where the exact voltage of V_{DDA} is known, the following formula can be used to calculate the absolute voltage measured by the ADC:

$$V_{CHx} = \frac{V_{DDA}}{FULL\_SCALE} \cdot ADCx\_DATA$$

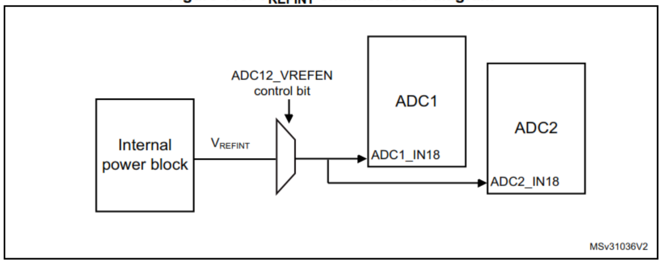
where FULL_SCALE represents the full 12-bit resolution (eg. 2^{12}-1 for 12 bit) and ADCx_DATA is the digitized value generated by the ADC, after sampling the selected channel.

- Assuming an ideal V_{DDA} = 3.3V, calculate the absolute voltage for the two ADC channels and display them on PuTTY/LCD at a fixed interval.
    - $\frac{3.3}{2^{12}-1} * ADCx_{DATA} =$
    - Remember to cast to float

The STM32F3 series includes an internal voltage reference (V_{REFINT}). If the exact value of V_{DDA} is not known, it can be derived from the internal reference:

$$V_{DDA} = 3.3V \cdot \frac{VREFINT\_CAL}{VREFINT\_DATA}$$

Figure 110. V_{REFINT} channel block diagram



where VREFINT_CAL is the VREFINT calibration value acquired from the factory (V_{DDA} at 3.3 V) and VREF_DATA is the digitized value that is generated by the ADC when measuring V_{REFINT}, using the currently unknown V_{DDA}.

Table 4. Internal voltage reference calibration values

Calibration value name	Description	Memory address
VREFINT_CAL	Raw data acquired at a temperature of 30 °C (± 5 °C), V_{DDA}= 3.3 V (± 10 mV)	0x1FFF F7BA - 0x1FFF F7BB

The factory calibration value is stored at address 0x1FFFF7BA. The macro below can be used to read VREFINT_CAL from memory. In this way VREFINT_CAL represents the `uint16_t` stored at this address.

```
#define VREFINT_CAL *((uint16_t*) ((uint32_t) 0x1FFFF7BA))    //calibrated at 3.3V@ 30C
```

This reference is internally connected to the input channel 18 of the two ADCs (ADCx_IN18). The nominal reference voltage is ~1.25 V. The VREFEN must be enabled by a function call to enable the conversion of internal channels ADC1_IN18 or ADC2_IN18 and this must be done BEFORE enabling the ADC:

```
ADC_VrefintCmd(ADC1,ENABLE); // setup ref voltage to channel 18
```

```
for(uint32_t i = 0; i<10000;i++); // I think this is needed...

// turn on ADC


ADC_RegularChannelConfig(ADC1, ADC_Channel_18, 1, ADC_SampleTime_xCycles5);
```

The V_{REFINT} channel is selected as it was done with PA0 and PA1. When sampling this particular channel, however, it is required to use a higher sample time aka a lower frequency. According to Table 27 in the [DS], an ADC sampling time of at least 2.2uS is required when measuring the internal reference voltage.

- Determine the correct sample time, based on the ADC prescaler that you selected earlier. Measure V_{REFINT} and store it as VREFINT_DATA.
- Calculate the ADC step size, using the formula for V_{CHx} and V_{DDA} and VREFINT_CAL.
- Use the ADC step size to calculate the absolute voltage for the two ADC channels and display them on the LCD at a fixed interval.

Exercise 2.6 – Calibrate the ADC

For this part of the exercise, you must add calibration functionality to your instrument. Although self-calibration has already been performed, it is necessary to calibrate your voltmeter against a known-good instrument, in this case the bench multimeter found in the lab:

- Connect the bench multimeter to PA0 (or PA1) on the expansion headers. A voltage of 3.200V must be applied to PA0 (or PA1). Adjust the voltage using the two potentiometers.
- Compute the average 12-bit measurement over 16 consecutive measurements. The absolute voltage calculated from the averaged measurements should be close to 3.2V at this point.
- Calculate a correction factor (float), which multiplied by the average, gives 3.200V. Repeat for PA1.
- The above calibration routine must be started using GPIO interrupts. Setup one of the joystick switches to trigger an EXTI interrupt, as done in Exercise 2.3. The interrupt must return immediately and should only be used to set a global variable. This variable will be checked in the `main()` loop and then reset. Note: Global variables used by interrupt functions must be "volatile". Do you know why?
- Calibrate the ADC and print the correction factor to PuTTY and/or to the LCD.
- Connect the voltmeter to V_{DDA} on the expansion header and measure the reference voltage. Assess whether the correction factor is reasonable.
- Use the flash memory (see exercise 3.2) to store the correction factor in the flash memory. The calibration factor should be reloaded from flash, if it is within a valid range (eg. 0.5< CALFACT < 1.5). Otherwise, it should be initialized to 1.0.

```
VREFINT_V:  1.230549 VDDA_V:  3.306496
PA0:3956 PA1:3965
PA0_V:  3.209736 PA1_V:  3.217038
PA0_Vavg:  3.191031 PA1_Vavg:  3.200721
CALFACT0:  1.004845
```

Report

Write a report of the exercise. The report should be about 5 pages long including a description, a chart, a table of used ports, documentation of results (eg. screendumps) and a well commended featured source code including header file.

Participants on a team draw up a single report with a cover, which show participants names, student number and group number. It should not be specified in the text, who wrote what. The report is uploaded preferably as PDF file in the course Assignment. The team will have an annotated report back via e -mail.

The report should typically include most of these standard points:

Title

Abstract Subject , methods, results.

Introduction general description, no details.

Problem definition What is it about.

problem limitation What you cannot with this method or this hardware.

Theory How do the various components and methods.

Hardware Diagram and table of used ports.

Software Function Description (no source code here).

Test method/software What worked, what did not and why.

Conclusion Must correspond to the problem formulation.

Appendix Source code, diagrams, possibly. Print drawings,

Data Sheets (only important pages), references and links.