SCC.369 Coursework 3: Interrupts and displays

Moodle submission 17:00 Friday week 10; weighting 33% of module.

Aim

In this coursework you will enable interrupts for the first time and write an interrupt handler in C, so finally you can start to do away with busy wait loops. You will return to the micro:bit display, and use interrupts to create the illusion that the pixels can be lit simultaneously. You'll create a basic API for controlling the pixels. Then you'll implement a similar API for a small OLED display which you can connect to the micro:bit's external I2C bus.

Instructions for all subtasks

The coursework is split into three subtasks. Implement everything from first principles using pure C**.** Only #include "MicroBit.h" and don't use any library functions unless they are explicitly stated as allowed. You can (and should) use typedefs, structs and literals from the nRF SDK.

We will test your code using the example calls in MainSubtaskRunnerCW3 **and some other example calls** so make sure you've tested everything well! New for this coursework, some of the functions you will write **return program execution** back to the MainSubtaskRunner. Exciting.

As with Coursework 1, **you ARE allowed to use the CODAL serial object** if that helps with debugging. Also, you can **use the CODAL I2C object** for talking to the I2C accelerometer and external display. You should use the NVIC_xxx() macros and you can use the NRFX_DELAY_ macros if you want to. You can use memcpy(), memset() and memclr().

We will use automated tooling when marking your code, so follow these guidelines to get full marks:

1. Write and submit your CW in a file called **CW3.cpp**. No additional files please!
2. Start with **the template CW3.cpp file on Moodle** because it has the functions correctly listed, you just need to write the code for each one!
3. Within CW3.cpp, write your code for each subtask **within the indicated section of the file**.
4. **Do not change the specified function prototypes** (i.e. function name, parameters and return type) for each subtask, use the ones given in the CW3.cpp template.
5. **Do not include a main() function or main.cpp file** in your submission. **Use the main() in MainSubtaskRunnerCW3.cpp** for testing because that's what we will use when marking.

For each subtask, 20-30% of the marks will depend on code quality, things like:

- Visually well-formatted and readable code
- Good, elegant code structure, style and efficiency, e.g.:
  - Appropriate use of loops, literals etc.
  - Initialise MCU peripherals only once where possible.
  - Only change the bits of a register that you need to.
  - Keep the code lean where possible so that it **executes as efficiently as possible, minimises power consumption and memory**, and is easy to read and understand.
- Ample and thoughtful comments including:
  - Before **all** function definitions explaining function purpose, parameters etc.
  - What variables are used for
  - The choice of bit patterns and/or literals being written to registers
  - The purpose of writing to registers, executing loops etc.
- No commented-out code with no explanation!
- No 'magic numbers' in the code – if using literal constants, define what they are!

Have fun 😊 and use the labs to ask about anything you don't understand!

Subtask 1, 35%: Create a basic API for the micro:bit LED display

This subtask doesn't require any external hardware – just the micro:bit. It requires you to write a display driver and API for the 5x5 LED display. The API consists of functions that might be called by an application developer, as listed below. In addition to creating these functions in your driver you will need to define at least one more function: an interrupt handler (or interrupt service routine, ISR) that is responsible for driving the display. You will also need to define a "frame buffer" array which is accessible to all the driver functions in which the pixel values to be displayed are held. Any function(s) and variables that you create which are not part of the API should be declared static so they are only accessible within your CW3.cpp file. And remember to comment everything appropriately – nice and succinct is good, as long as there are enough comments to follow exactly what the code is doing. Also, remember that these functions all return! No while(1) any more ☺.

Since you can't drive all the LEDs on the micro:bit display simultaneously, to create the appearance of a static image you have to scan through rows so quickly that there is no visible flickering. Unless, that is, you're a fly. To do this, create a timer interrupt that results in the entire display updating at 100 frames a second, i.e. at 100Hz, ±5%. During each frame, your interrupt handler will have to drive each row, one at a time. Please scan the rows from top to bottom. Keep your code as efficient as possible, especially in terms of RAM consumed.

API functions:

Function prototype: void initMicroBitDisplay(void);

Initialise the driver, including:

- clearing the display frame buffer;
- configuring any necessary GPIOs;
- setting up a periodic timer that results in an interrupt;
- configuring that timer's interrupt handler to show the correct row of the LED display.

The LED display should be 'live' after the initMicroBitPixels() function has completed and returned.

Function prototype: void clearMicroBitDisplay(void);

Clear the entire frame buffer, thus clearing the display so no pixels remain lit.

Function prototype: void setMicroBitPixel(uint8_t x, uint8_t y);

Sets the given pixel on the micro:bit display. The top-left pixel is at (0, 0) and the top-right pixel is at (4, 0).

Function prototype: void clearMicroBitPixel(uint8_t x, uint8_t y);

This function should clear the given pixel on the micro:bit display.

Internal functions/variables:

Global variable: uint8_t microBitDisplayFrameBuffer[5][5];

The frame buffer of current pixel values. You can change the type and/or size, but keep this name please!

Function prototype: static void microBitDisplayIsr(void);

This function acts as the interrupt service routine (ISR) for a timer IRQ that is set up in initMicroBitDisplay().

Subtask 2, 35%: Create a basic API for the external OLED display provided

(Remember that, unlike with the last coursework, this time you are encouraged to use the CODAL I2C object so it's easier to send and receive I2C packets.)

Write another basic display driver, this time for the external I2C OLED display provided. This uses an SSD1306 driver IC, see datasheet here: https://cdn-shop.adafruit.com/datasheets/SSD1306.pdf. Connect the four display pins up to your micro:bit using your breadboard and jumper wires. There are I2C pull-up resistors on the micro:bit I2C_EXT_SCL and I2C_EXT_SDA lines so please use these for SCL and SDA respectively. It makes sense to run the external I2C bus at 400kbps rather than the default speed because we'd like you to minimise the time spent communicating on the I2C bus. It would also be good to minimise the amount of RAM your driver uses.

The API you are to create is similar to that used in Subtask 1; the functions you need to implement are listed below. With this display, the built-in controller IC does the low-level display handling, but you still need to maintain a display frame buffer and send updates to the display. For this task **do not use any interrupts**; simply update the display over I2C whenever a call that results in new content is made.

Again, remember that the functions all return program execution back to the MainSubtaskRunner.

API functions:

Function prototype: **void initOledDisplay(void);**

Initialise the display by performing the necessary configuration over I2C and clearing the display.

Function prototype: **void clearOledDisplay(void);**

Clear the entire display.

Function prototype: **void setOledPixel(uint8_t x, uint8_t y);**

Sets the given pixel on the OLED display. The top-left pixel is at (0, 0) and the top-right pixel is at (127, 0).

Function prototype: **void clearOledPixel(uint8_t x, uint8_t y);**

Clears the given pixel on the OLED display. The top-left pixel is at (0, 0) and the top-right pixel is at (127, 0).

Function prototype: **void drawOledLine(uint8_t x_start, uint8_t y_start,
uint8_t x_end, uint8_t y_end);**

This function should draw a straight line between (x_start, y_start) and (x_end, y_end).

Internal functions/variables:

Global variable: **oledDisplayFrameBuffer;**

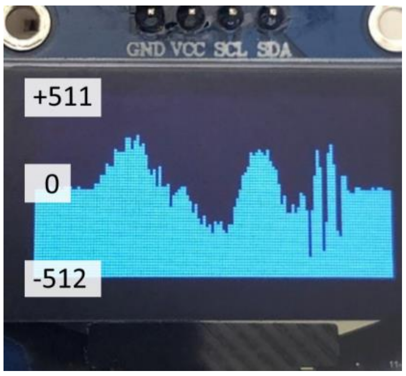
The frame buffer of current OLED pixel values. You must set the type and size of this variable, but keep this name please!

Subtask 3, 30%: Create a graphing application on the OLED display

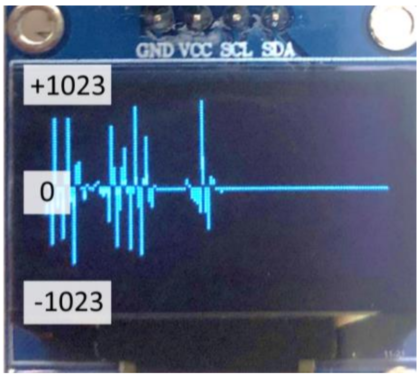
In this final subtask you will combine everything you've learned to create your own embedded systems "graphing" application!

Your graphing application will plot X axis accelerometer readings on OLED display, so that the magnitude from -512 to 511 appears on the vertical (y) axis. The display should update in real time on the horizontal (x) axis. To make this work, draw a vertical bar up from the bottom of the display to a height that represents the accelerometer data point, where +511 draws a bar all the way to the top and 0 draws a bar to the vertical mid-point of the display. Start with the first reading on the very left of the display (the first column, x=0); new readings should be added to the right of previous readings until the display is full after 128 readings are displayed. At that point the display should "scroll" horizontally to the left by one pixel to make a space for a new reading which is then plotted in the last column. As the display scrolls, old readings will disappear off the left side of the screen.

The default mode, as described above, will display the raw X accelerometer readings. But the application will have a second mode! When the B button is pressed, your application should stop plotting accelerometer data, clear the display, and switch to a second mode. Now it will instead plot jerk. You should calculate jerk simply as the mathematical difference between the latest accelerometer reading and the most recent previous reading. So jerk ranges from -1023 to +1023. Jerk values should be displayed a little differently: rather than always drawing a line upwards from the bottom of the display, you should draw a line up from the middle of the display (y=31) for positive jerk and down from the middle for negative jerk. Pressing A at any point clears the display again and reverts back to plotting accelerometer data.



Acceleration mode (default)
Accelerometer X-axis values read from the accelerometer are shown as vertical lines drawn up from the bottom of the display. A continuous X-axis reading of around 0 lights up the bottom half of the display, leaving the top half dark (shown on the left).



Jerk mode
Jerk values calculated from the X-axis accelerometer data. The baseline, no activity, is a horizontal line at y=31 (shown on the right). Positive jerk, when the latest accelerometer value is higher than the previous one, is plotted as a line going up from y=31. Negative jerk is plotted as a line going down from y=31.

All this functionality is enabled by a call to a function we want you to create in your CW3.cpp file:

Function prototype: void graphData(uint8_t refreshRate);

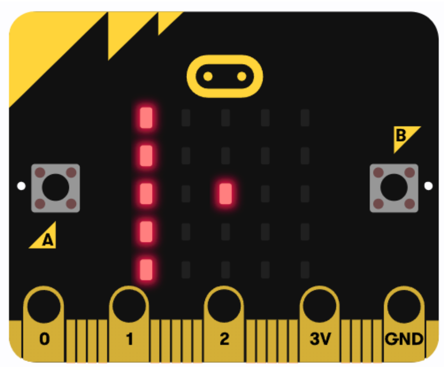
This graphData() function does **not** return, it loops forever reading/calculating/displaying data. You will need some helper functions too. You can ignore the parameter unless you want extra marks as described below.

So to recap:

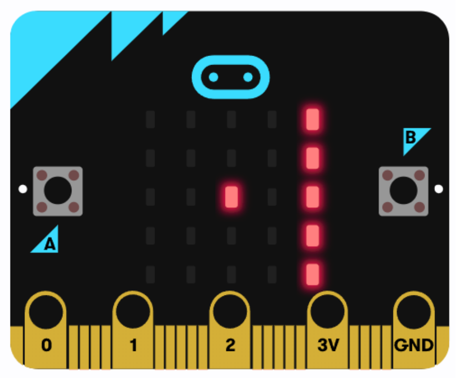
- All accelerometer readings should be from the X axis only.
- Acceleration values will be in the range -512 to +511 and these should map to vertical lines on the display with between 0 to 63 pixels lit up, drawn from the bottom of the display up.
- The jerk values you calculate should be in the range -1023 to +1023 and these will map to vertical lines from y=31 down to y=63 (for negative values of jerk) and from y=31 up to y=0 for positive values of jerk. A value of zero will result in a single pixel at y=31.
- Remember that you are allowed to talk to the accelerometer using a second CODAL I2C object and its methods, see the example code in MainSubtaskRunnerCW3.cpp.

Doing all the above gets you about half of the marks for Subtask3! If you want to push yourself, you can get even more marks if you:

- Update the graph at the refreshRate passed into graphData(), up to a limit that's determined by the efficiency of your code. The faster you can update the display, the better! We suggest you use a timer for this, but it's probably easier if you busy wait for the timer event rather than using an interrupt.
- Enable your timer interrupt from Subtask 1 so that the micro:bit LED display is active. When in mode A (accelerometer) light up the column of LED pixels on the left, nearest to button A, plus the LED in the middle of the display. When in mode B (jerk) instead light up the column of LED pixels on the right, nearest to button B, plus the LED in the middle.
- For the mode switching, use the GPIOTE peripheral to generate an interrupt when either button A or button B is pressed. Use another ISR to process these button press interrupts so that your code switches modes accordingly without having to poll the buttons.



Mode A for accelerometer with the leftmost column of micro:bit LED pixels lit, plus the centre pixel.



Mode B for jerk with the rightmost column of micro:bit LED pixels lit, plus the centre pixel.

We can show you our reference code running in the lab sessions if you want to see it live 😊.

The efficiency of your code will be considered in marking. The priority is to keep data transfers over I^2C to a minimum. Feel free to enhance the aesthetics of your graphing application (such as with gridlines and text) but do not change the underlying functionality or significantly impact your efficiency as you may lose marks. Show us in the lab to make sure.

Mark Scheme

For each subtask, 70-80% of the marks will be awarded for meeting the functional requirements given. 20-30% of the marks will depend on code quality as described on the first page above. If you do not use the filename, function prototypes and hardware configuration specified (all repeated in **red** below) you will lose marks. Please submit a single .zip file (and no exotic compressed file formats, just regular .zip). Your work will be assessed by a combination of automatic processing and manual inspection. Your final grade will be based on a weighted mean of your subtask marks.

Subtask	**Hardware config**	**Weight**	**To be submitted** (submit code in **CW3.cpp**)
1: Basic API for the micro:bit LED display	Just a micro:bit	35%	**initMicroBitDisplay(); clearMicroBitDisplay(); setMicroBitPixel(); clearMicroBitPixel(); microBitDisplayIsr();**
2: Basic API for the external OLED display	micro:bit wired up to I2C OLED provided	35%	**initOledDisplay(); clearOledDisplay(); setOledPixel(); clearOledPixel(); drawOledLine();**
3: Create a graphing application on the OLED display	micro:bit wired up to I2C OLED provided	30%	**graphData(); any helper functions…**