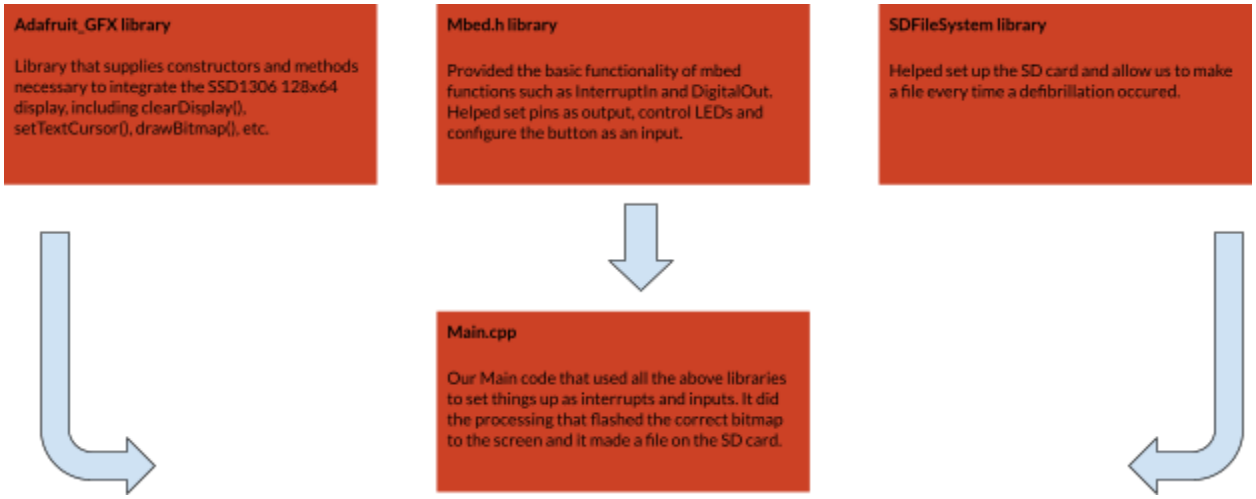
Rishi Singhal rs872
Raghav Kumar rk524
Laboratory Assignment 5 - Real Time Scheduling
Demo: https://youtu.be/_F9d9MBLKH4

I. Introduction

This project attempts to simulate the function of an **Implantable cardioverter-defibrillator** device to detect irregular pacing (heart arrhythmia) and correct it accordingly. It took the switch button as an input, did some software processing on the input data and displayed the pacing results on the 931 128x64 OLED display. We also used an SD card to store the data of the defibrillations so as to make them accessible to the doctor. The display used the I2C communication protocol while the SD card used the SPI protocol.

2. System diagram (software)



3. Hardware description (if applicable)

BOM

Monochrome 1.3" 128x64 OLED graphic display. Product ID 938 from Adafruit.
Breadboard
Wires
SD card

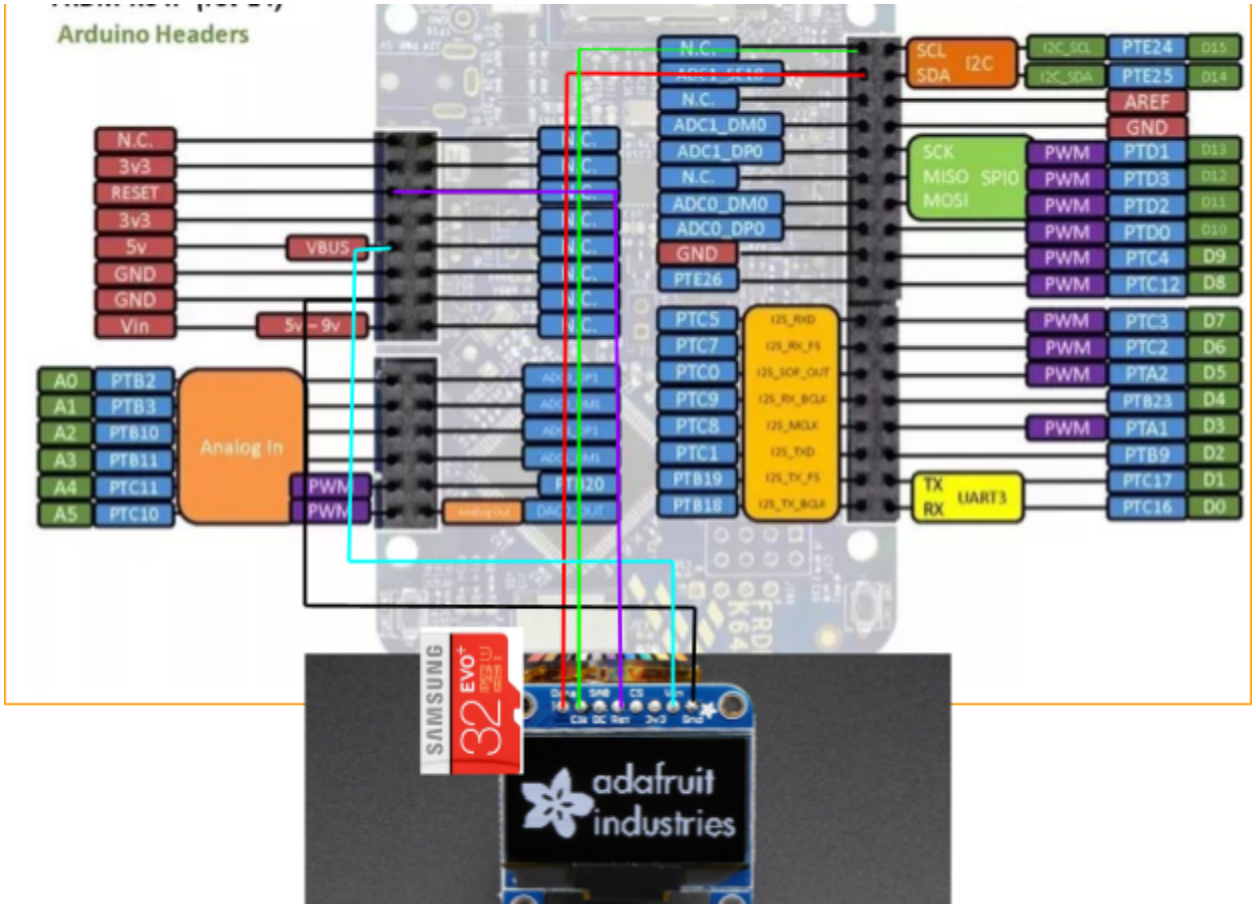


Figure 1. Updated Schematic of the Project (SD card offset for visual aid)

4. Detailed software description

Our project required four important header files - mbed.h, Adafruit_SSD1306.h, Adafruit_GFX.h and SDFileSystem.h. The Adafruit_GFX library houses basic graphics things to allow the drawing of text and simple shapes and included Adafruit_SSD1306.h.

We started with instantiating several objects and global variables. We first made the SD card object that communicated using the SPI protocol. We identified the right pins on the board for the various lines of SPI (MOSI, MISO etc) and understood how the constructor for the object worked in order to correctly make the SD card object. The File pointer is used to create and store a file into the SD card whenever a defibrillation takes place. The buffer array is for input to the file. In order to make the OLED display work we instantiated an I2c object with the I2C ports PTE24 and PTE25 as arguments. We then instantiated an Adafruit_SSD1306_I2c object with the I2c object we created before, the port D4, and the OLED's address, height, and width as parameters. Moving on, we made a DigitalOut object and passed in the red LED as parameters to the constructor. This allowed us to configure and use the red LED on the board at our will. Using

the InterruptIn object, we instantiated the SW2 button as an interrupt. This would go on to simulate the ‘heartbeat’ in our project. The Serial pc object allowed us communicate serially with the PuTTY that helped us while debugging. We had two global variables heart_rate and counter that we used in our program. Finally, we also made a timer object that helped keep track of time within the program so we could figure out when to carry out the pacing and the defibrillations.

Starting from the main() function, we first tied all the LEDs to high because they were active low and we wanted to start with all of them turned off. We then cleared the display by calling the clearDisplay function on screen, our I2C object. Using some other helpers, we printed “Start Measurement” on the middle of the screen indicating the user to start pressing the button. By using the ‘toggle_red()’ helper function, we toggled the red led signalling the start of measurement. In order to calculate the normal heartbeat, we started the timer using t.start(). The way the InterruptIn works is that it takes a pointer to the function that we want to act as an ISR. So in our case, whenever there was a rise in the button (i.e. the button was pressed and the voltage went high), it called the flip function. The flip function in turn bumped up the counter by one each time it was called. This way we kept track of the number of times the button was pressed. Once we pressed the button five times, it entered the ‘if’ conditional, toggled the red LED signalling end of measurement. Using the t.read() function, we calculated the times elapsed in these five button pushes and divided it by five to get the average or the ‘normal heartbeat’. Using similar functions but different text position, we displayed a message to the user telling him or her about their normal heart rate. The wait function just waited for the number of seconds that we passed in as parameters. After resetting the time, we went into the detection function.

The detection function had almost the same functionality as the main in that it called the monitor function whenever there was a rise in the button. It should be noted that now the button was configured to called an ISR different from the one it was calling before, something we haven’t done in past labs. Monitor had the same structure as flip but it maintained a threshold of 0.5 seconds (for normal pacing) and 1.5 seconds for defibrillations. After every five beats, (kept track of using counter), monitor would run these calculations and compare the average value of the latest set of five beats to the ‘normal heart beat’ called heart_rate. If the average was too low or too high, it called the defibril function.

Using the same constructs for displaying text the defibril function first displayed some text. Then using the drawBitmap function of the I2C object, we drew out the bitmap on the screen. The various parameters passed into this function take care of whether the screen needs to be filled or not, and if yes, then to what colour. ‘Defib’ is the name of the bitmap that is the image of the ‘shock’ as seen in the video. After waiting for two seconds, we bump the counter for the number of defibs and make a new file in the SD card, deleting the old one.

However, if in monitor, the heart rate is irregular but not so much that it requires defibrillation, it goes into the irregular function that loops five times to display a sequence of yet another bitmap. These bitmaps are names heart for the jagged lines or pulses and noPulse for the

flat line as seen in the video. It is important to note however, that the delay between the pulses was set to be the normal heart rate. This is important because we want the pulsing to happen at exactly the pace of the normal heart rate. After both defib and irregular, we print All Good to the screen.

Writing to SD Card

The project uses SPI to write to the SD card, passing in the appropriate MISO, MOSI, and CS lines of the board designated for SD cards, as seen in Figure 2. We interfaced with with the SDFileSystem library, writing the appropriate information to the board (MISO, MOSI, CS corresponding to the SD card) and this library then passes it into the SPI library and sets the frequency to begin data transmission, 100kHz for initializing and 1MHz for data. The SDFileSystem library uses FATFileSystem's library to make a new directory for the SD card and make/edit text files. It passes in the name chosen when initializing the sd card, set in the parameters along with the SPI ports, so that FATFileSystem can create a tree based on this parent directory, we called it sd. A running counter (initially 0 at the beginning of the program) was implemented for the number of defibrillations that occurred, and it increments whenever the code enters the defibril() function. Also in the defibril() function is where the project uses the object initialized for this SD card. It first tries to read for the file "defib.txt" within "/sd" to see if a file already exists, using fopen (a function in the standard cstdio library) to open the file and setting the access mode to read. If there is such a file, it will close the file and delete it so that it can continue to create a new "defib.txt." The code now creates "defib.txt" using fopen and write access mode, and then uses fprintf (operates similarly to printf but used for printing inside files) to write into the text file. Using fprintf, it writes the number of defibrillations that occurred, using the running counter, to the file. The SD card now contains a "defib.txt" file that can inform the doctor of how many defibrillations have occurred so far.

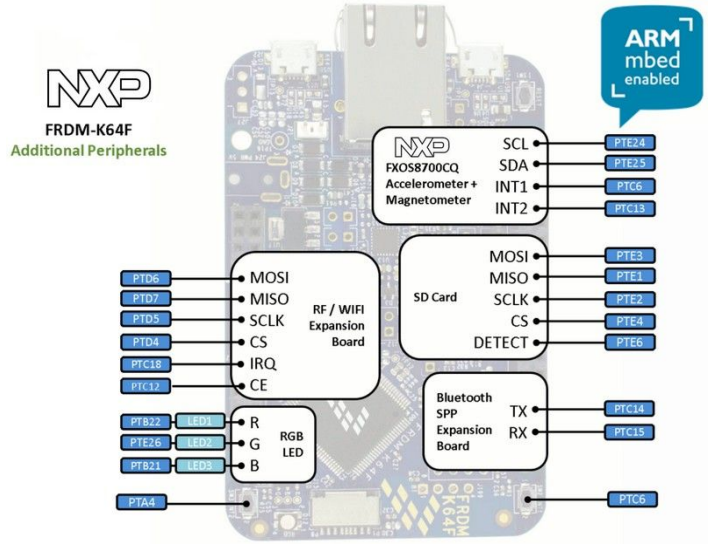


Figure 2. Diagram from NXP showing the designated ports for MISO, MOSI, SCLK, and CS for the SD card.

5. Testing

Testing was done through small unit tests to ensure the smaller segments of our code that didn't require external libraries, such as the logic to traverse through different segments of the program, were tested first before adding on these complexities. Then, anytime a new feature was added, it was first tested in isolation before being implemented in the main code. Some examples of test cases to ensure robustness were, testing the button rise on an LED toggle to ensure it jumps to a corresponding interrupt handler, testing the real time timer with delays to ensure the values seem accurate, testing if the timer still functions when started before an interrupt, and writing to an SD card that is already written to. The OLED screen implementation itself, was mainly tested visually as functionality was generally determined by the display on the screen. Although starting with simple pixels, then rectangles, the implementation of the OLED was soon extended to testing if the bitmaps were displayed correctly.

6. Results and challenges

This project was different from what was proposed. Although functionally the same, instead of an SPI implementation intended on the TFT ST7735R LCD an I2C OLED screen was implemented instead. A large amount of time was spent debugging the SPI implementation for the TFT ST7735R. Initially no data was being sent through, and upon further investigation using an oscilloscope probe, chip select was never being toggled. This prompted a realization that chip select was not declared as a GPIO pin, critical to ST7735R's ability to toggle this value. Once this modification was changed, a toggle for cs was also done in this program too, outside the ST7735R functions to see if the pin was finally functional. Once chip select was fixed the next approach was to ensure the clock line was also functional. To aid in debugging, the ST7735R library was modified, making its writecommand function public to write test bits and test spi on smaller bits of data and also lowering the clock frequency in case that was the cause of its disappearance on the oscilloscope and in order to make it more easy to display a change on the LCD screen when something did change due to a lower baud rate. This helped us get to the root of our issues and allowed us to eventually get a clock signal to display on the oscilloscope, and a second probe showed the MOSI data line also being triggered at the falling edge of the clock, demonstrating data transfer.

Now that we knew the board was functional in its SPI communication, we returned to the standard functions found to interface with the LCD such as drawRectangle() and fillScreen(). The screen was still blank but we caught a flicker, so we reduced the clock frequency and saw static being displayed on the LCD. The LCD was now receiving data. However, it seemed that whatever data sent was either corrupted during communication or not readable by the LCD. This

was further investigated upon testing with an arduino. When running a test program, the same issue arose, prompting us to believe either the library has a flawed command in the list of hex commands given, or there is a problem with the LCD screen. Due to time restraints, the answer is still not known but interfacing with an external device was ultimately our greatest challenge.

However, to demonstrate the knowledge we gained from this experience, an SD card was added to the project and upon small incremental steps, ensuring the functionality of SPI on the board, the SD card was written to and read from. The remaining components of the project remained the same, although we were unable to use the concurrency framework from Lab3 as hoped since we moved to MBed's compiler without the assembly code present from Lab3 on Keil, but the two functions, despite the lack of a process queue, operate quick enough where the performance is similar to concurrent processes. While implementing threads was attempted, this was not practical for the project's use since button interrupts would take one thread to the resource (LCD) used in the other thread.

If we were to implement this project a second time, our primary focus would be to implement SPI on Keil's IDE instead as there is much more control given on this IDE. Given this control, while there may be a lot of reuse from the ST7735R library, we wouldn't have to completely rely on whether or not the library's code is robust. If a bug does arise, we could more easily test it and through debugger mode see where the issue arose from. Using Keil's IDE would also allow us to use process queues from Lab3 to try to execute our processes concurrently.

7. Work distribution

Design and Implementation - While Rishi primarily came up with the project idea due to his interest in Biomedical devices, both of us followed an incremental approach to set the ground running. We first had a couple of brainstorming session which in addition to writing the pre-proposal and proposal gave us a good idea of the project and the major technical challenges we were about to face. Although we had made deadlines, we seldom kept any and hence had to improvise often.

Code Review and Testing - We reviewed the code both while we were coding the program up, as well as afterwards. Through pair programming, we could have one person looking at the code as it was being typed in order to see if it made sense, while the other person would move forwards while coding - we could frequently catch bugs such as misspelled variables and syntactical errors this way. After each major section of code was completed, the two of us would take a look at it and run through it mentally to see if it made logical sense.

Documentation - Because we knew that this project would require us to do a lot of search on the internet, we maintained an extensive Google Document that we updated each time that we worked. This way we could always go back and track our progress. If anything some part of the code ever broke, we could see what changes we had made that could have led to the event. In

addition we also commented our code so that if one of us worked on it remotely, the other could easily understand the work he did.

Collaboration - We primarily collaborated using pair programming, working side-by-side and debugging issues that came up on the fly. The report was done collaboratively too, albeit remotely.

Communication and Code sharing - We talked and sent messages on the Facebook messenger application while sharing code using Github and Google Docs.

8. References, software reuse

a. MBED Online Compiler
    i. https://www.mbed.com/en/

b. Adafruit_GFX MBED library
    i. https://os.mbed.com/users/nkhorman/code/Adafruit_GFX/
    ii. Config file was altered to enable all functionality rooted in abstract classes

c. I2C test script to print addresses for each device found on I2C bus
    i. https://os.mbed.com/questions/6129/I2C-Not-Working/

d. PuTTY
    i. https://www.putty.org/

e. Adafruit Wiring Diagram for SSD1306 OLED Display
    i. https://learn.adafruit.com/monochrome-oled-breakouts/wiring-128x64-oleds