Lab 6 Documentation

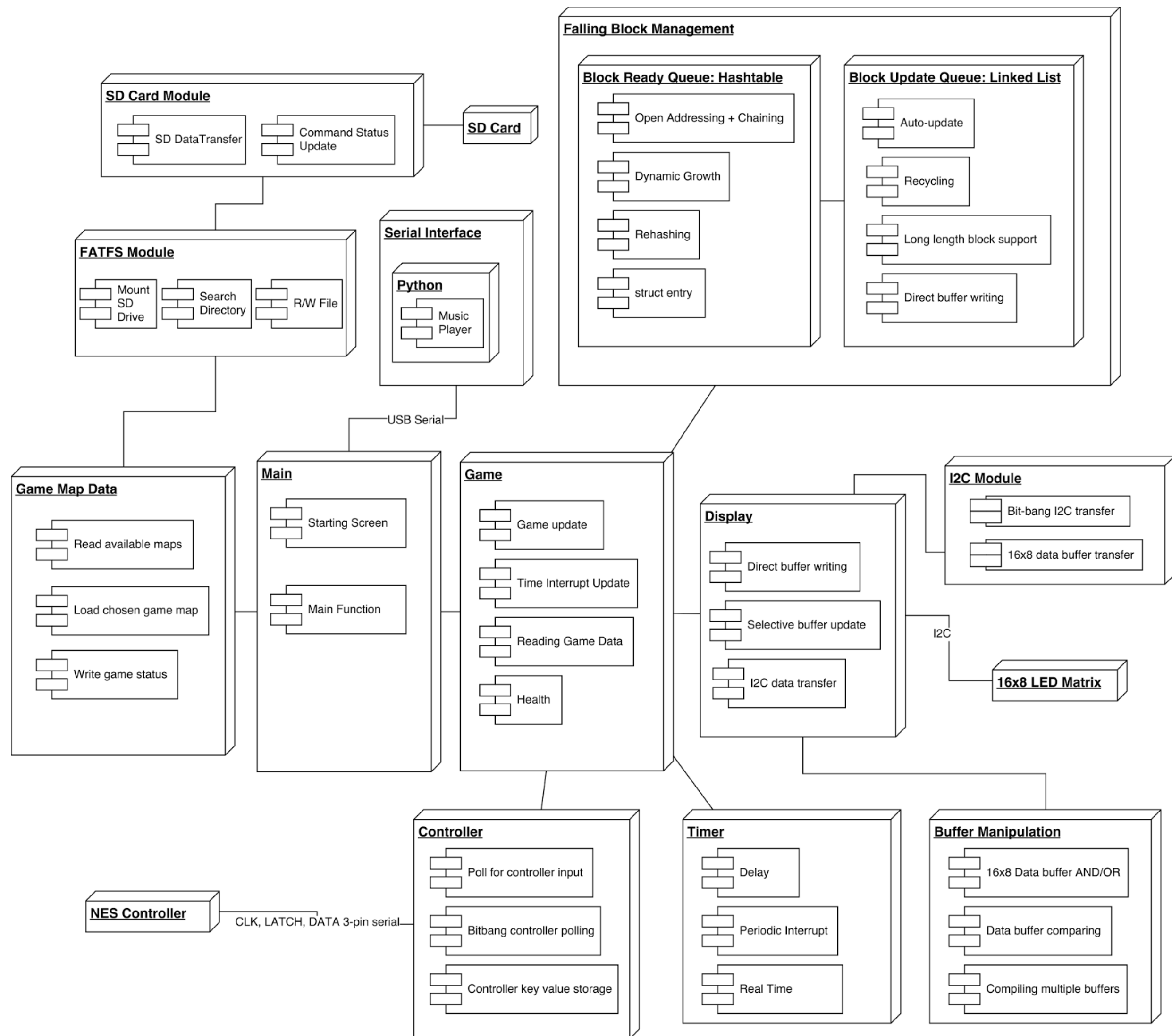
Dongze Yue(dy85), Yang Liu(yl572)

1. Introduction

We build a rhythm game for our final project. It's like a very simple Osu!. The game will read maps from an SD card and display blocks on the LED screen accordingly. When the game starts, it will send an message through USB serial port to laptop so the laptop can read the message and start playing the corresponding music. The blocks displayed on the LED matrix will fall down as game goes, and the player need to press the button on a controller at the right time to earn game scores.

The player can also add new maps to SD card so the pleasure would never stop - we provide a Bash script so every player can generate map files from an Osu! map he or she likes. The device is highly portable so the player can play the game any time, as long as he or she brings a microUSB cable to power the board. Therefore, the game can provide a short period of relaxing time for those who want to take a break.

2. **System diagram**

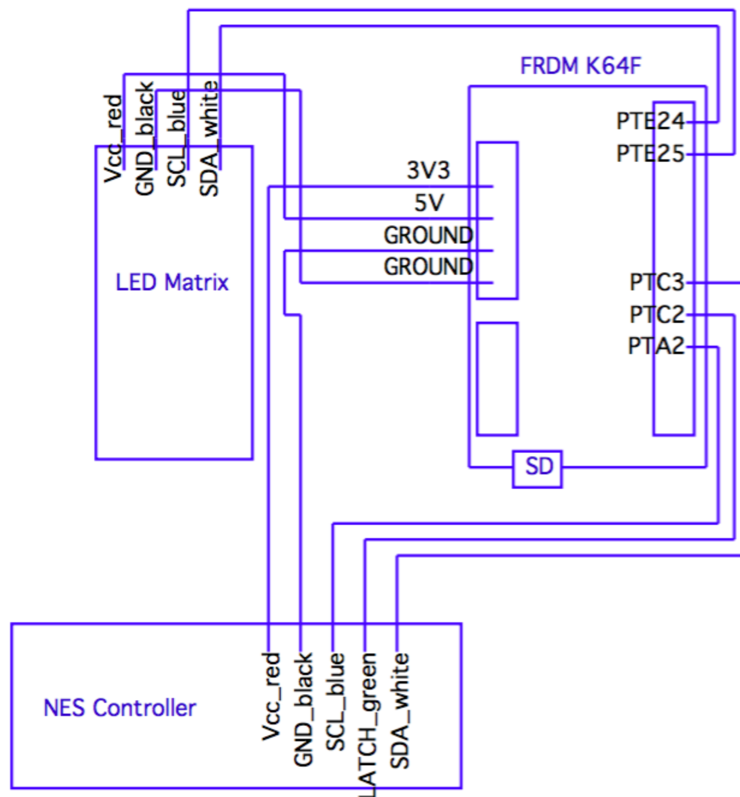


The block diagram posted above shows our major embedded system architecture. The system contains four layers together with several interactive modules. The four layers are the Communication layer, Game Realization layer, Game Data layer and Hardware layer. The communication layer is in charge of communicating with the user such as sending print statements via USB Serial to the computer and displaying the game menu for user to select levels on the LED Matrix. The game realization layer is in charge of actually running the game by actively checking the database at each cycle and to change and advance the falling block on the display. A timer interrupt triggered at each microsecond is implemented for updating the game. Also, a real time structure has been used, since our rhythm game is totally based on timing. The falling blocks in the game are all implemented as real-time tasks that needs to be constantly checked and updated. This will be elaborated later in the report. The game data layer is in charge of loading, localizing, processing and updating the map files that user downloaded

in the SD card. We implement a hashtable to store the game data, because their arrival time totally depends on the map and cannot be linearly stored. We also use a linked-list structure for the update list to update active falling blocks on the screen, since at each cycle all the active blocks needs to be traversed and linked-lists are easy to traverse through. The hardware layer contains all the fundamental hardware protocols, such as I2C and serial that are used to talk to the LED and the controller. Also it contains the basic modules that is frequently used in higher-level code such as timer interrupt delay. Each layer works interactively together to make the game working.

3. Hardware description

Pinout for FRDM K64F please refer to figure 2 in the reference.



4. **Detailed software description**

Our game works as the following: While a music is being played in the background, there will periodically be blocks (with varying length) falling down from the top of the screen, and the user will need to control the two paddles A(right) and B(left) on the bottom platform to hit the block when the falling block just touches the bottom of the screen. All the blocks comes in with the downloaded map file. And each block has three main attributes: the arrival time, the arrival track (A or B) and length. All the blocks will fall down according to the beats of the music, so the arrival time are not nice numbers (more sophisticated numbers relating to the actual music file). Also, by accurately hitting the blocks with the keys, the user will gain points in the fever bar on the sides. The fever bar will also vary with the current point value.

We will elaborate on how we implemente the game and the description of our software design by going through each layers of our system.

Communication Layer:

In order to debug, we import a library called fsl_debug_console. It has a macro defined function called PRINTF, that will print string values through USB serial output at the baud rate 115200. This way, the ARM board can talk to computer and we can use printed value to debug. Furthermore, because we want a background music accompanying our game, but there is no audio output on board so we decided to use our laptop to play the music. Thus, we write a small Python script that will listen to the USB serial port. The script will open the serial port with baud rate at 115200, and then it will stay in an infinite loop to poll messages from the serial port. As long as it sees a “Game:Start” value, it will break the loop and play the corresponding music.

We have to display all the game on a 16x8 unicolor LED screen, so the image being displayed on the screen can be described by a 16x8 binary matrix buffer. Therefore we came out with an effective way of displaying the game: insteading of writing and computing everything at the same time, each component in the game has its own 16x8 binary buffer. So during the update process of each component, it only writes to its own buffer. At the end of the update cycle, there is a master display update, which will take all the existing buffers and stack them together into the main display buffer by doing logical OR.
The falling of the blocks are also made easy: since we create a separate buffer for the blocks being displayed, and all the falling blocks has the same speed, we can basically bit shift all the columns of the buffer at certain interval to emulate the effect that all the blocks has “dropped down” for one row.
When user presses A or B on the controller, a section on the bottom of the screen will be lit. This is achieved by writing a hard-coded key-press buffer to the main display buffer if a key is being pressed down.
The fever bar on the sides display the current fever point the user has obtained. It was simply done by bit manipulation of the first and last column.

Game Realization Layer:

The game contains two major update routine. The first update routine is real-time sequential logic and is at the PIT1 interrupt handler. The timer is set to generate an interrupt every 1ms. At each cycle, the real-time variable (current_time) is incremented. Also, the game checks the database whether there is a scheduled block falling down at the current time. If so, the attributes of the falling block will be extracted from the database and appended to the update-queue, a linked-list that keeps track of the active falling blocks on the display. If the current time is a multiple of the pre-defined block falling speed, then the game will call the method to bit-shift the block buffer to lower all the falling blocks. Also, the game will also traverse the update list and decrease each block's length by 1. If any entry on the list has reached zero length, it means all parts of the block has been displayed and will therefore be removed from the update list. This way of managing the falling blocks greatly improves memory performance since once the block is fully displayed onto the screen, all its related information are freed from the memory, and the rest of its falling process will be handled by the bit shift method. Besides the basic display of the falling blocks, the update function also checks for the end of game (if there are no further blocks in queue), reload the hashtable if threshold time is met (since the size of hashtable is limited, this will be elaborated further in the data layer) and decreases fever point with respect to time.

The second update routine is written in the game_update() function that is called in a while(1) loop, so therefore it is a combinational logic and updates information that requires to be checked asynchronously. The first variable the function checks is the game state, as it controls the flow of the game. By switching different modes of the game, the game can start, pause, reload and end the current map. When the game start is start, the update function then actively polls the controller to see whether user presses a key or not. (We didn't choose interrupt because interrupt is not available for NES controllers, it will be elaborated further in later section.) If so, corresponding buffer will be loaded to the display buffer. If the user hit a block with the corresponding key, a portion of the fever points will be added depending on the accuracy of the hit. All the other display updates introduced in the previous section are also implemented in the game_update() function.

Game Data Layer:

Looking at the data that we import from SD card again, each block comes with three attributes - arrival time, length and track. It surely makes sense that we use the blocks' arrival time as a key to store them in the data structure so we can run queries frequently and read them later. The arrival time of blocks differs a lot numerically, as it is counted in microseconds, and all the numbers does not contain certain pattern. Therefore, we adapt a hashtable to store all the block data. At every timer interrupt cycle, the game can check the hashtable with the current time as the key. If there is any hits, that entry is then pulled out and appended to the update queue.
The way of actually implementing the hashtable is very tricky, because the FRDM-K64F board has a large limit in runtime memory and dynamic reallocation of pointers. When we firstly attempted to hash all the game data at the same time into a giant hash table, we found out that

the game suddenly froze and stopped working. Therefore, we choose a fixed size for the hashtable (the table doesn't dynamic grow), setup a loading threshold and write a rehash function to it. Basically, every time the game tries to load all the map data, only a fixed amount of the most recent falling blocks is inserted into the hashtable and set the last loaded block's arrival time as the next loading time. When game advances to the next loading time, the hashtable gets rehashed, by creating another table, inserting unfinished tasks into it and destroying the original hashtable. After rehashing, the function that loads tasks gets called again and loads further falling blocks into the table. This system ensures minimal memory usage and gives the device more freedom.
As described above, at every cycle, the task that is arriving at current time will be added into update list. The update list is simply implemented as a linked-list, as the game need to traverse through the list often. Each node of the list contains a block of any length. Every time the display buffer advances, the length in each node gets decremented by one, and all the nodes with length zero gets removed from the list and freed forever. This again ensures minimal memory occupancy.

Hardware Layer:

LED Display: The communication to the LED Matrix Display is via I2C. In the project we manually bit-banged the I2C protocol and successfully established communication with the display. Besides the regular I2C protocol, the displaying data is being sent 8-bit at a time for 16 times. Therefore, we used a uint16_t buffer[8] instead of uint8_t buffer[16], for easier data transfer.

Controller: The way of communicating with the NES controller is very tricky. The Nintendo controller doesn't contain an active pin that will signal the microcontroller that there is an external input. Instead, it is a very passive controller that user needs to manually drive and latch and clock pin on the controller to actively shift out the current key being pressed bit-by-bit and compare to the pre-defined library. If multiple keys are being pressed, the return value is simply the sum of all the single key values together. Since the controller is very passive, we use polling to get the controller input instead of interrupt, and the speed is surprisingly fast.

SD Card: To make it easier to update maps, we decide to store maps in SD card. We use two layers to achieve the function of reading SD card. The data in the card has to be read in the unit of blocks, and the first layer, named SD Card Module in the block diagram, provides that function. Upon SD Card Module, we have a higher layer called FATFS Module. This layer treat the SD card as FAT file system and it can read data stream from the card. It will read map data from card line by line.

5. Testing

We cannot use debug mode in Keil to debug, because in debug mode the time interrupt will be messed up by breakpoints. Our game is highly dependent on a timer so we have to use another method that does not affect timer for debugging.
We decide to use serial output to debug. We make a function PRINTF that can print string through USB serial onto laptop. Put PRINTF function in the place where we want to see information and we can read the message on our laptop. This method significantly facilitates our debug.

6. **Results and challenges**

We have so far accomplished all of our goals written in the proposal. Despite that we thought that our game could be too ambitious and complicated to implement, we still managed to finish the whole project with all the working features. The only part that we did not cover in the proposal is the audio playback. Since the memory on board is too limited and streaming audio can use up a lot of space and bandwidth, we decide to have the laptop listening to serial commands on the other side, and play the music synchronously with the game.

The most complicated part of our design would be the game data layer and game realization layer that dodges memory hazards and loads the large map file without crashing the system. Originally we implemented another data structure block_t to store all the information for one falling block, and append such object to a linked list for updating. However, we found out that once we loaded more than 23 entries in the linked list and hashtable, the whole game basically froze. Therefore, we came out with the idea of progressively and actively loading just a chunk of the data into the dynamic memory space and reusing the space for reloading later data. It worked surprisingly well.

Extra credit work: In order to make the game come true, we did a lot of coding on the laptop side, including a bash script that takes in official Osu! Beatmaps and converts it into our own beatmap format. Also we have a python script that runs in the back and listens to the serial port. Once it heard the starting of the game, it will play the corresponding soundtrack that user chose.

7. **References, software reuse**

www.nxp.com FRDM-K64F Kinetis Software Development Kit: All the libraries, sample code for filesystem and debug interface.
www.mcuoneclipse.com MCU on Eclipse: Tutorials of setting up I2C devices and configuring Keil MDK
www.adafruit.com Documentation of LED Backpack and tutorial of setting it up.
https://github.com/adafruit/Adafruit_LED_Backpack Arduino Library for LED Matrix: Reference for creating our own library for LED Display.
http://www.cs.yale.edu/homes/aspnes/pinewiki/C(2f)HashTables.html Implementing Hashtables in C
http://osu.ppy.sh/ Providing Beatmap files and soundtrack for our game

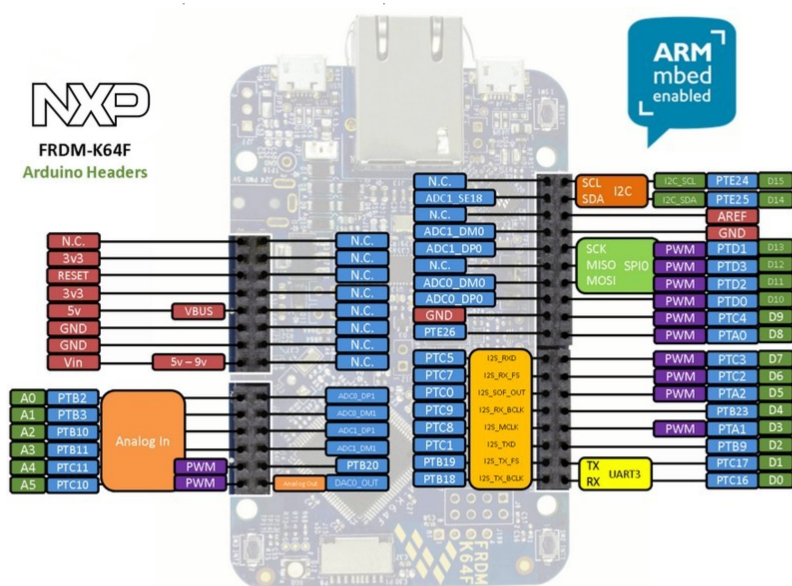


Figure 2. Pinout for FRDM K64F