MARMARA UNIVERSITY
FACULTY OF ENGINEERING

EE2004

Microprocessor Systems

NucStation

A STM32 Game Console Based on Nucleo - L073RZ

	Dept	Student Id	Name Surname
1	**EE**	**150720004**	**Egemen Kazanasmaz**

Table Of Contents:

1. Introduction:

In this project I will try to create a video game console that is similar to retro consoles from the 80s. It will be able to load games from cartridges.

I will write software libraries to develop games for the given board.

2. Design & Part Choices:

I had to cut a lot of corners to make this project happen. STM32L0 MCUs are far more capable than the 80s mentioned consoles, however lack of a proper GPU and using “C++” instead of assembly limits performance of this device.

Usually, in these types of consoles, read data copied into ram, or instructions directly run from EEPROM. But since I’m using C++ to write games quickly, I don’t have this level of access to system memory.

I also wanted to use a programmable sound generator (PSG) for music and add a second controller for the second player. However, because of the lack of time I had to remove them from the project.

I will use Nucleo-L073RZ as the brain of this project.

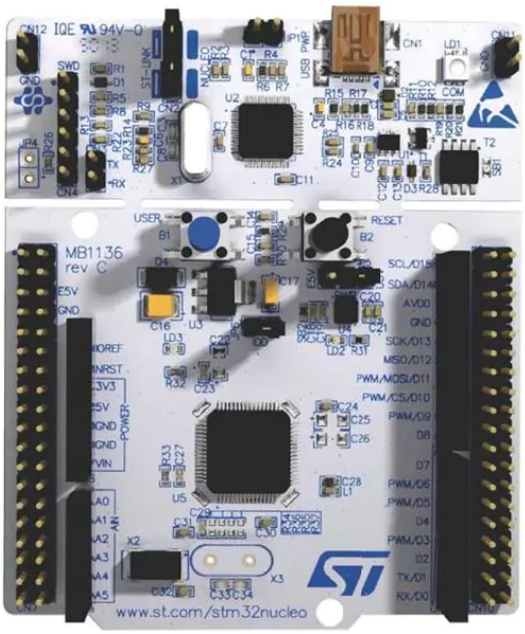

Figure 1. Main board: NUCLEO-L073RZ

I’ve chosen the PCD8544 LCD as the screen for this project, because since it’s widely used by maker communities around the world, it’s easy to find resources for development. This screen is also known as Nokia 5110 screen.

Figure 2. PCD8544 LCD SCREEN

I will use Atmel AT24C256 EEPROM as storage for my games, in the cartridges. It uses I2C protocol to write/read data.

Figure 3. Atmel AT24C256

3. Steps:

Design of the graphics:

First of all I wanted to see something on screen. PCD8544 modules have a variety of revisions from different manufacturers. Board of mine supports 5V communication. But I didn't want to risk it, so I designed a daughter board to decrease the logic level to ~3V.

Figure 4. PCD8544 with daughter board.

Then, I found a library for driving PCD8544. But it lacks bitmap drawing capabilities. So I forked it and tried to add this functionality. (https://github.com/ege-adam/Nokia-LCD5110-HAL)

Here are the first few iterations of the bitmap drawing function:

Figure 5. Instead of writing data of the bottom parts to different "banks", I accidentally send them to the same "bank".

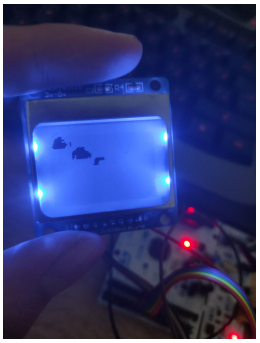

Figure 6. Banks are correct now, but forgot to reset the pixel offset.

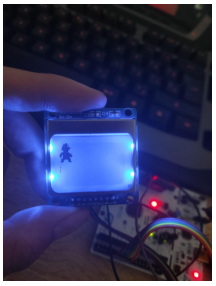

Figure 7. It looks fine now, but has 1 pixel overflow.

Figure 8. Perfect.

Design of the controller:

I’ve designed a very basic custom board for the controller.

It has 6 keys. It has 8 connections: One of them is NC, it can be used for purposes like controller detection, extra button. And one of the connections is for ground connection. All buttons work on pull up configuration. In a perfect world, I must have used a shift register to reduce the number of cables. Same as all corners that is cutten from this project, I had to remove the shift register because of lack of time.

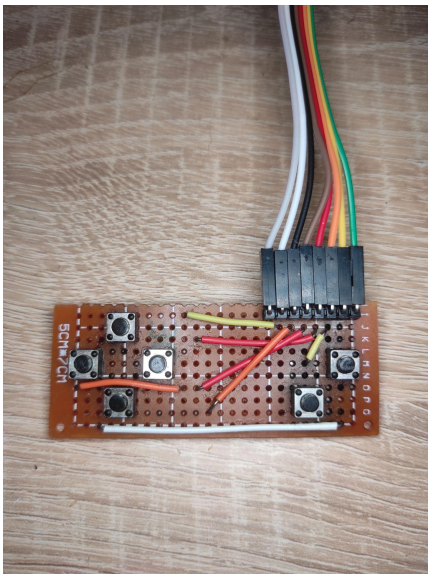

Figure 9. Picture of the controller

On the software side, I check buttons for every frame. It’s a common method to check user input within the game's default loop. Unlike real world systems, like cars for example, in games if we use interrupts or events, user input may fight with the physics updates, render updates or other updates in the game engine. If the user presses a button before physics update it may pass through walls. If the user presses a button while “render update”, half of the character might be rendered incorrectly.

So, ingame engines, we must check for user input in every frame. Preferably before physics check, after graphics update. And usually, physics engine loops work 3x of the main loop. In other words, physics checked 3 times in every frame. For performance reasons, I will use 1:1 scale.

Design of the EEPROM:

To program EEPROM I've used an Arduino Uno. Through Tera Term, with UART protocol to Arduino I was able to flash firmware files to EEPROM. While doing so, I added "EGE" to each of the firmware files. This way, when L073RZ reads from EEPROM, if it sees "EGE" in the last three bytes, it will stop reading / flashing.

To do this I had to design a daughter board, since AT24C256 works in pull up configuration.

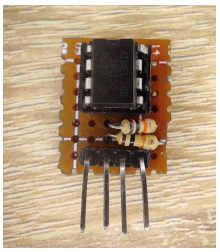

Figure 10. First iteration, no daughter board.

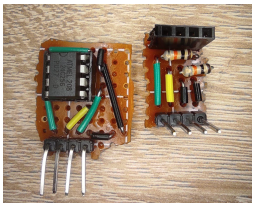

Figure 11. Second iterations, resistors moved to a daughter board and used jumper wires to cleaner connection.

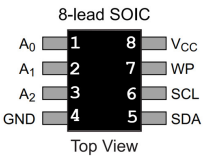

Figure 12. Pinout of AT24C256

A0 to A2 pins used for deciding I2C address. To use the default address, 0x50, I shorted them to the ground. Also I shorted WP, the write protection pin, to ground too. SCL and SDA pins connected to SDA-SCL of controller board (Arduino or L073RZ). I had to connect them to the Vcc pin through 10k resistors to pull-up.

I found a great tutorial on the internet (Link to tutorial) to use with the HAL library. Since my version is mostly unmodified from this tutorial, I won't upload it to my Github. It can read EEPROM 1 byte in each loop.

Design of the firmware & bootloader:

It was the most rough part of this project. I had to divide flash memory in two parts. One for a bootloader and one part for firmware. Bootloader is responsible for loading games into firmware. And firmware will be our games.

Let's dive deeply into the topic.

NVM	NVM addresses	Size (bytes)	Name	Description	
Flash program memory	0x0800 0000 - 0x0800 007F	128 bytes	Page 0	sector 0	Bank 1
	0x0800 0080 - 0x0800 00FF	128 bytes	Page 1		
	-	-	-		
	0x0800 0F80 - 0x0800 0FFF	128 bytes	Page 31		
	. . .	. . .	. . .	. . .	
	0x0800 7000 - 0x0800 707F	128 bytes	Page 224	sector 7	
	0x0800 7080 - 0x0800 70FF	128 bytes	Page 225		
	-	-	-		
	0x0800 7F80 - 0x0800 7FFF	128 bytes	Page 255		
	. . .	. . .	. . .	. . .	
	-	-	-		
	0x0801 7F80- 0x0801 7FFF	128 bytes	Page 767	sector 23	
	0x0801 8000 - 0x0801 807F	128 bytes	Page 768	sector 24	Bank 2
	. . .	. . .	. . .	. . .	
	0x0802 F000 - 0x0802 F07F	128 bytes	Page 1504	sector 47	
	0x0802 F080 - 0x0802 F0FF	128 bytes	Page 1505		
	-	-	-		
	0x0802 FF80 - 0x0802 FFFF	128 bytes	Page 1535		
Data EEPROM	0x0808 0000 - 0x0808 0BFF	6 Kbytes	-	Data EEPROM Bank 1	
	0x0808 0C00 - 0x0808 17FF		-	Data EEPROM Bank 2	
Information block	0x1FF0 0000 - 0x1FF0 1FFF	8 Kbytes	-	System memory	
	0x1FF8 0020 - 0x1FF8 007F	96 bytes	-	Factory Options	
	0x1FF8 0000 - 0x1FF8 001F	32 bytes	-	User Option bytes	

Figure 13. Flash memory organization of STM32L073RZ.

From the reference manual we can see that the main flash is divided into banks, then divided into sectors, and lastly to the pages.

I will use the first 32kb of 192kb memory as the bootloader. And I will use the last 160kb as firmware. In this configuration the bootloader starts from 0x08000000 and ends in 0x08007FFF at page 255. So firmware starts at 0x08004000.

Unlike most of the STM32 boards, on the default and most recent HAL library, it was impossible to erase data through sectors or addresses. I had to use pages. Also while writing, I had to use 32 bit words (4 byte), however my EEPROM library could only read 1 byte at each loop. Because of the RAM limitations it was impossible to load data into a variable then flash it. So I did a trick to merge every 4 readings into a 32 bit word.

```
for(j = 0; j < 64; j++)
{
    lastFour[wordCounter] = buffer[j];
    wordCounter++;
    if(wordCounter == 4)
    {
        //FLASH
        lastFourWord[0] = lastFour[0] | (lastFour[1] << 8) | (lastFour[2] << 16) | (lastFour[3] << 24);
        wordCounter = 0;

        if(sector == 0) FlashTheGame(lastFourWord, sector, true);
        else FlashTheGame(lastFourWord, sector, false);

        sector++;
        itoa((int) i, percentLoad, 10);
        LCD_print(percentLoad, 0, 0);
    }
}
```

Figure 14. Code for converting 4 bytes into a word.

After flashing, we can finally boot into the game.

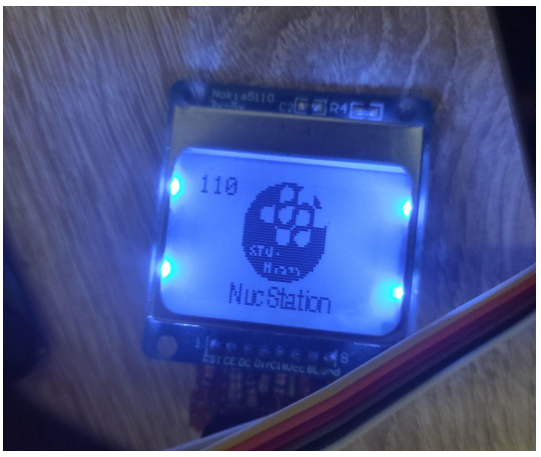

Figure 15. Bootloader loading game

Design of the games:

As of writing this report, there are two games only. I planned 5 games at the start, but because of the problems in the development of libraries, I couldn't manage to finish others in time.

NucStack:

You have to stack bars in time, or else you will lose.

NucSnake:

A classical snake game where you play as Mr./Ms. Snake and try to collect as many baits as you can.

4. Resources:

STM32 & EEPROM

Arduino EEPROM (Used for flash tool development)

Bootloader for STM32F76XX (Used as reference to write my own bootloader for STM32L073XX)

STM32L0XXXX Reference Manual

image2cpp (Used to convert bitmap images to byte array)

Nokia-LCD5110-HAL (My modified version to draw bitmaps)