Memory tester and SPI Game Console

Francisco Oliveira
up201604934

João Conceição
up201506202

***Abstract*—The project consists of 2 tasks. In the first task, it was proposed to develop a program to carry out self test operations, to test the Flash and RAM memory banks of the ATMEGA328P micro controller. In the other task, our goal was to develop a system around the same micro controller using an SPI or I2C compatible device. For our device, we chose a LCD Nokia 5110, that uses the SPI protocol and decided to use it to create a simple Game Console.**

***Index Terms*—testing, memories, SPI, LCD Nokia, ATMEGA328P, Game Console**

I. Memory tester

All of the developed code is freely available in a Github repository [1].

All of the code was developed within the Platform IO IDE in Visual Studio Code.

A. Introduction

Testing of memories is an important part of ensuring the reliability of a system, since they allow for the detection of faults in systems with memory, and memories can become faulty over time. Testers are generally run in the boot up procedure of the system, in order to ensure that the system will operate normally. In and ordinary system such as a PC a faulty memory address may lead only to a crash, but in a safety critical system such as a plane's autopilot this could lead to a disaster.

For that self testers were developed using the C programming language with AVR's own libraries for the ATMEGA328P micro-controller within the Platform IO IDE in Visual Studio Code, which is a popular MCU for hobbyists and embedded applications although recently it is being overtaken by more modern and cost effective alternatives such as ARM and RISC-V based MCUs. This micro-controller is part of the AVR's ATMEGA family so the developed testers should work for any device in that family with minor changes since they share the same ISA and memory banks configuration (they differ in size but the way they are integrated into the MCU is the same).

The developed testers were made with ease of implementation in mind, so they were developed as an external library that can be easily included into an already existing program. In order to further simplify this, the library was written with AVR GCC specific **__attribute(constructor)** functions for the testers, so that the testers run before the main function of the user's program is called. This implementation makes the developed software non portable in the sense that if a non AVR GCC compiler is used this implementation won't work, but since most of the popular IDEs for AVR (Arduino IDE and Microchip's own IDE) use this compiler, this will work on almost all of the software developed for ATMEGA family MCUs.

The ATMEGA238P micro-controller has three types of memory banks: RAM, Flash and EEPROM. In this project the development of self testers was made for the RAM and Flash, since for the Flash and EEPROM the same testing methodology can be used.

1) RAM: The RAM of the ATMEGA328P is of the SRAM type, and is part of the same addressing space of the General Purpose registers and IO registers of the micro-controller which are also of the SRAM type. The RAM is 2048 by 8 bits. Since SRAM can be considered to have an almost infinite write endurance, the tester to be implemented can write and read from this memory as much as necessary.

2) Flash: The Flash of the ATMEGA328P is 16384 by 16 bits and has a write endurance of 10k cycles, so in order to test this memory the tester needs to avoid to write to memory in order to preserve it's lifespan.

B. Flash tester

This tester is ran first because if it fails the program is considered to be corrupted, so the RAM tester may not behave as expected. This tester also uses only registers for this reason, since when it's executed the integrity of the RAM is unknown.

1) Algorithm: The implemented algorithm uses a cyclic redundancy check (CRC) that is provided by ATMEL (now Microchip) in an Application Note [2], that goes into depth of how this methodology is implemented. If the reader is interested please visit the website cited. Note that the implementation made in this project was slightly different from the one proposed, but the base idea is the same.

2) Implementation: In order to access the flash memory bank the library $< avr/pgmspace.h >$ that is provided by the manufacturer was used. The whole tester is encased in a function (**__TEST_FLASH**()) that as mentioned previously used AVR GCC specific function attributes that allow it to run before main being called, and that the tester used only registers. For forcing the use of registers the C keyword **register** was used in all variables inside this function. The necessary checksum is declared as an external variable so the user can declare it in his own program.

When checking the generated checksum if the tester finds that it doesn't match to the expected checksum, an error function is called **void TestError()** that must be declared by the user.

Since the checksum is declared in the code itself, it will be stored in the Flash memory, so it will affect the calculation of the checksum. So in order to avoid that the address in which the checksum is stored will be ignored by the CRC algorithm. In order to calculate the checksum the user can run the program within Platform IO, and set the last if of the tester (tests if the checksum is correct) as a debugging break-point. Then the user is able to extract the checksum and declare it in their program.

3) Fault injection: The developed tester also allows for the injection of faults in the specified address, although this is only useful when debugging the tester itself. Since the fault injection code is declare within an **ifdef**, this fault injection will not be compiled during deployment of the tester. For more information see the project's Github [1].

4) Improvements: An important improvement would be the development of a script/program that would automatically generate the program checksum and declare within the library, as this would even simplify even further the integration of this library. This was attempted but due to time restraints the idea was abandoned.Also the implementation of a better CRC algorithm is also an important improvement, specially since the way the CRC was implemented doesn't give 0 as this type of CRC is intended to give.

C. RAM tester

This tester uses only registers for obvious reasons, and it doesn't test the memory that is occupied by the stack, because if the program is capable of returning to the main function the RAM is considered not faulty. Note that this assumption is dangerous as that if per example when returning from the stack the program goes to a random address that allows the program to run without soft locking the processor it will look as if the memory is not faulty. In a further development the stack would need to be tested as well.

1) Algorithm: The implemented algorithm was the MATS++ algorithm, which is part of the industry standard MATS testing algorithms, that are generally used for testing RAM memory banks. This algorithm isn't as complete as some others such as the MATS C-, but it strikes a good balance between testing coverage and number of operations required.

This algorithm operates as follows:

$$\{\Updownarrow (w(0)); \Uparrow (r(0), w(1)); \Downarrow (r(1), w(0), r(0))\}$$

Explanation:

$\Updownarrow$: move up or down an address (up in our case)
$\Uparrow$: move up and address
$\Downarrow$: move down an address
$w(X)$: write an X to the current address
$r(X)$: read from the current address and if the value is not X a fault is detected

All operations within the parenthesis after the arrows need to be made before moving to the next one, so in the first step all addresses must be written to 0 before moving on to the second step

2) Fault injection: The developed tester also allows for the injection of faults in the specified address and in a specific step of the MATS++ algorithm, although this is only useful when debugging the tester itself. Since the fault injection code is declare within an **ifdef**, this fault injection will not be compiled during deployment of the tester. For more information see the project's Github [1].

3) Implementation: The whole tester is encased in a function (**__TEST_RAM()**) that as previously mentioned used AVR GCC specific attributes so that it executes before main, but it is executed after the Flash tester.

The implementation of this algorithm is very simple. Each step of the algorithm is modeled as a for loop, in which all of the addresses of the RAM bank are covered, and the operations of that step are performed on the addresses. This is repeated for all of the steps in the MATS++ algorithm. When an fault is found in the r(X) part of the steps, an error function is called **void TestError()**, which must be declared by the user.

4) Improvements: The most important improvement to this tester would be that it should test the stack as well, because only then it could be said that the entire memory bank was tested. In order to implement this it would be necessary to within the tester function to copy the contents of the stack to a part of RAM that was already verified, test the stack, and then copy the contents back.

II. SPI GAME CONSOLE

A. Introduction

Serial Peripheral Interface (SPI) is a serial communication protocol, which is used frequently by microcontrollers to control peripheral devices or even other microcontrollers. It allows synchronous communication between devices because it uses a clock to synchronize data transmission and recession and is used for a distance of up to 10 meters. The SPI devices communicate with each other in ”full duplex” mode, i.e. bidirectional and parallel communication which is always commanded by the Master and use a ”master-slave” architecture with a single master and with one or more slaves.

A standard SPI bus consists of 4 signals, Master Out Slave In (MOSI), Master In Slave Out (MISO), the clock (SCK), and Slave Select (SS). In the case of this project the microcontroller will not read from the devide, so we only need to take into account the MOSI, SCK and SS signals.

B. The SPI Device

The SPI device used in this task was a LCD Nokia 5110, which is a basic graphic monochromatic LCD screen with a resolution of 48 x 84 pixels.

The LCD contains the following 8 pins:

- **VCC**: Positive power supply between 2.7V and 3.3V;
- **GND**: Ground Pin;
- **SCE**: Enables the LCD controller (Active Low);
- **RST**: Reset Pin (Active Low);

- **D/C**: Select between command mode (low) and data mode (high);
- **DN(MOSI)**: Pin in which the Data is Transmitted;
- **SCLK**: Serial clock;
- **LED**: LED backlight supply.

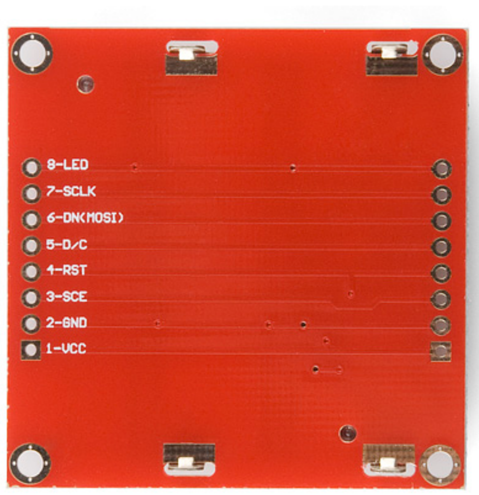

Fig. 1. LCD pins

According with the datasheet of the LCD Driver PCD8544 [3], this module works in the range of 2.7 to 3.3 V (power supply, backlight and control lines), for any 5V logic micro-controller board like Arduino, some sort of logic level shifting is required (otherwise display may get damaged).

To connect the Microcontroller to the LCD module, it was used a voltage divider for each line with the exception of the VCC and GND pins. That means there are 6 voltage dividers. Each voltage divider consists of 1k and 2k (2x1k) resistors, this drops the 5V into 3.3V. The VCC is directly connected to the 3.3V pin of the Arduino Board.

Besides the setup described above, it was used 4 push buttons to control the snake movement and 1 extra button for several game options.

Relatively to the SPI protocol, the DN pin corresponds to the MOSI signal pin, the SCLK to the clock and the SCE to the Slave Select (SS).

C. LCD Operation

In order to control the display, it is necessary to send control information via the Serial Data Pin (DN). The Driver can receive 2 different types of messages, a data message with information to display an image in the screen or a command message with a command to configure the LCD.

To send a message, firstly, the controller is enabled by setting SCE low, then the message content must be written in the DN pin. The last step consists in disabling the controller by setting SCE high. Data and command messages are distinguished by the D/C pin, which must be High when sending a Data message and Low to a command message.

The initial configuration of the LCD must be done by command messages using the set of instructions present in the page 14 of [3] and in [4] in the LCD Operation section.

The Display has a total of 4032 pixels. However it can't have each pixel controlled individually, but only in blocks of 8, with each pixel being a single bit, making the control at the byte level.

The LCD Display is divided in 6 banks of memory, each one with 84 columns with each column representing 1 byte (8 pixels per column) and every pixel 1 bit. When the respective bit is set to 1 the pixel is turned ON and when is set to 0 is turned OFF, having the Least Significant Bit on the Top and the Most Significant Bit on the Botton of the column.

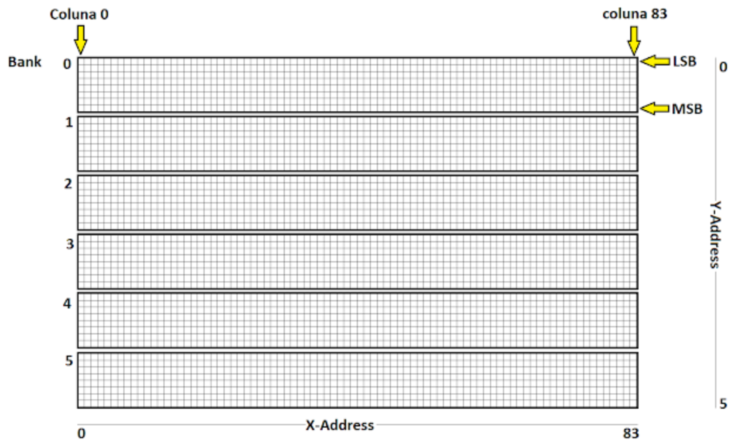

Fig. 2. LCD Memory Structure

An important feature regarding the LCD configuration is the Addressing Mode, that states the way that the content of the sequential data messages is displayed into the screen.

The Horizontal Addressing displays the next byte coming from a Data Message in the next column of the same bank or in the case of being the last one of the current bank it displays it in the first column of the next bank.

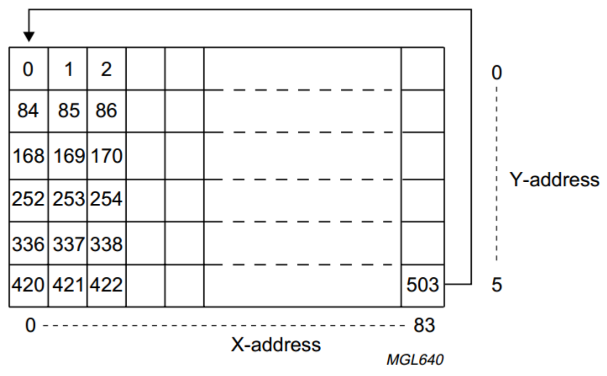

Fig. 3. Horizontal Addressing

The Vertical Addressing displays the next byte coming from a Data Message in the same column of the next bank or in the case of being the last bank it displays it in the next column of the first bank.

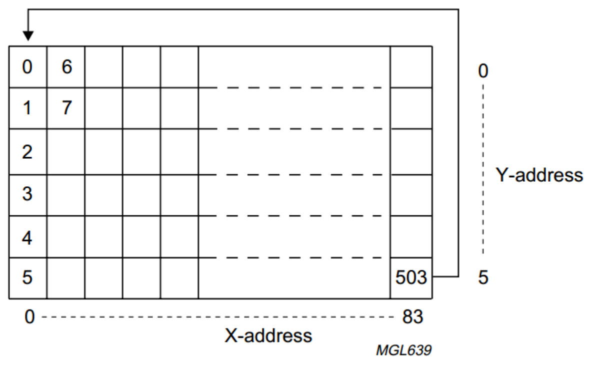

Fig. 4. Vertical Addressing

For both modes, if the current column is the last from the last bank, the next data message will be displayed in the first column of the first bank.

D. Approach

For the project itself, it was used Vertical Addressing to display the Data Messages at a rate of 1 Mbit/s.

Since the Microcontroller is unable to read the stored values in the LCD memory, the values should be stored into some program variables that allow the track of the system's state. For that matter and knowing that the screen is updated byte by byte, every value regarding the image in the screen should be stored by order of addressing and updated in an array with 504 positions. The 504 positions come from the fact that 48x84 pixels make a total of 4032 pixels and when aggregating the bits into groups of 8, we get 504 groups(bytes) and that having 4032 variables would exceed the Microcontroller RAM limit of 2k bytes.

In order to update the screen image, two methods can be used. The first consists in running through all positions of the array and update all values one by one, from the first memory address of the LCD to the last. The second corresponds to a jump to a specific memory address on the LCD using the instructions **Set X address of RAM** and **Set Y address of RAM** sent as command messages to the module and make the desired update at that specific address. This project makes use of both methods in different states.

E. Game Rules

The game corresponds to a remake of the classic mobile Snake Game, that was commonly present in the monochromatic mobile phones from the early 2000's. In this game, the player starts with a small snake that aims to reach the food in order to increase the body size while having to avoid colliding with the walls and its own body, in order to get the highest score possible.

F. Game Development

The Game was developed with a set of different menus, besides the game itself. A start menu that allows the game to begin. A pause menu that saves the current information about the game and allows to continue the game later or to quit it and a Game Over Menu that shows the score, after the snake dies and a Win Menu that shows that the player reached the goal.

In order to allow the logic transition between menus and the continuity of the game, it was developed using a State Machine as shown below.

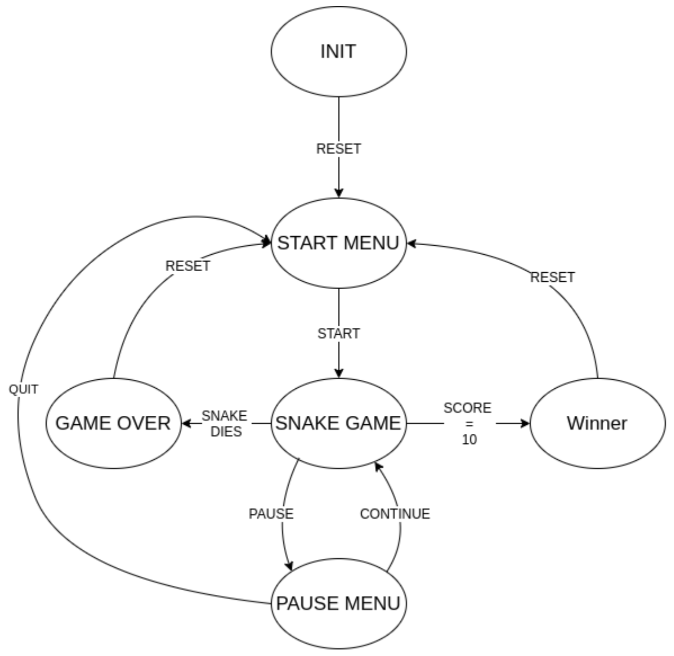

Fig. 5. Game State Machine

In the **Snake Game** state, the user can control the snake movement using the 4 direction buttons as mentioned before and jump to the **Pause Menu** state by clicking the extra button. While during the **Snake Game** state, the image is updated via the first method mentioned before, during the **Pause Menu** state the image is a result of writing some characters using the **Set address of RAM** instructions in those specific addresses.

However the second method does not update the values sent to the screen in the array of values, keeping the last frame of the snake game stored and ready to continue as the user leaves the **Pause Menu** by choosing the **Continue** option. Other states such as **Game Over**, **Winner** and **Start Menu** that only require characters are fully updated using the second method.

The use of characters is possible due to a file named ”lcd_chars.h” that contains a matrix with the bytes configuring a set of characters in the order at which they appear in the ASCII table with each character being 5 bits wide and 7 bits tall. After this file is uploaded into the Microcontroller, the matrix is stored into the Flash Memory so there’s no need to store the characters displayed into the RAM Memory.

G. Issues and Solutions

Due to some bugs related to the undesired shifting of the image on the LCD (bugs that weren’t debugged), it was opted to send each bit of data one by one on the Serial Pin and change the clock state for each one, instead of just writing the whole byte of data in the SPI Data Register. That allowed a higher control in the data that was being sent to the LCD Driver and stopped the undesired shifting of the image.

H. Future Improvements

Adding new options in the game menus would be an interesting update such as back light brightness adjustment through the control of the LED pin using a PWM with a configurable duty cycle. Another interesting option would be the choice of the snake’s body width in pixels.

In a more ambitious perspective, the improvement would require the development of other games and the creation of a library of games.

REFERENCES

[1] https://github.com/joaomcconceicao/AVR-RAM_and_FLASH_tester
[2] http://ww1.microchip.com/downloads/en/AppNotes/doc1143.pdf
[3] https://www.sparkfun.com/datasheets/LCD/Monochrome/Nokia5110.pdf
[4] https://github.com/FXGO98/SELE_LCD_Nokia_Task