Vending Machine

E155 Final Project Report

December, 10 2021

Ava Sherry and Leila Wiberg

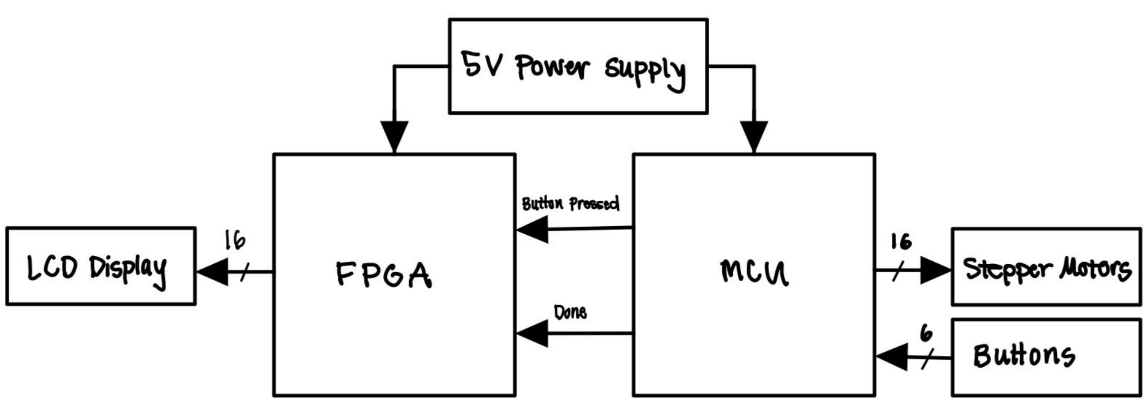
Abstract

They say millionaires have at least 6 streams of income. We decided to use this opportunity to add another stream of income by creating a vending machine. Instead of buying one, we used our knowledge of microprocessors to make a 6 item vending machine with an LCD driven by an FPGA and motors driven by the MCU. The user simply needs to follow the instructions on the LCD screen to get an item from the vending machine. The LCD displays “Select Item” in the idle state while the MCU checks for a button press. The user presses one of six buttons, positioned relative to the corresponding motor, and the motor turns. As the item is dispensing, the LCD displays “Dispensing…” and no other buttons can be pressed. In later iterations, we plan to incorporate an RFID sensor to require payment for items, however we felt that Harvey Mudd students deserved their items for free.

Introduction

Normal children want Barbie houses, toy cars, or iPads, but Ava is different. Ever since she was a child, she dreamed of having a vending machine of her own. This project aims to achieve that childhood dream of providing 24/7 snacks and other items to our friends. After an all nighter in the Digital Lab, we realized the importance of having machines that can provide sustenance at all hours of the day.

This vending machine uses a MAX1000 FPGA and a STM32F401RE MCU to drive an LCD display and 6 motors, respectively. The display waits in the idle state where it displays “Select Item”, directing the user to press one of 6 buttons corresponding to 6 spaces in the vending machine. Each space has a spiral dispenser and is mounted on a 28BYJ-48 stepper motor. The motors are driven by 4 GPIO pins on the microcontroller. The microcontroller waits for a button press, then begins to turn the corresponding motor as well as send a signal to the FPGA that a button has been pressed, prompting it to change the display to read “Dispensing…”. Once the motor stops turning, the MCU sends a signal to the FPGA to return to the idle “Select Item” state, and then begins looking for a button press again.



New Hardware

Stepper Motors

We selected stepper motors to turn the spiral dispensers. Stepper motors are best for this application because they will be moving at a low speed but a high torque, they hold their position when not in use, and they are relatively precise with their movements. We selected 6 28BYJ-48 motors--one for each spiral dispenser. These are unipolar motors, which makes them simple to drive. The 28BYJ-48 motors require only 5V of power supplied from a stripped USB charger which makes powering them safer and simpler than high power and high torque motors. Using the 5V supply from the MCU would draw too much current (240mA, whereas the MCU can only supply about 20mA), so the motors require an external power supply and a driver. We selected the ULN2003 driver (*28BYJ-48 -- 5V Stepper Motor Manual*). The ULN2003 is a transistor array with Darlington pairs to amplify the current and voltage to drive the motor. It can drive up to 500mA and it has a 2.7kOhm base resistor to directly interface with 5V devices (*ULN2003 Datasheet*).

LCD Character Display

For this project we used an LCD Character Display to give instructions to the user. In its idle state the LCD displays “Select Item”. Once a button is pressed, the MCU sends a signal to the FPGA which changes the text to “Dispensing”. The LCD will continue to display “Dispensing” until the FPGA receives the “done” signal from the MCU. This “done” signal tells the FPGA to change the text back to “Select Item”.

The LCD screen has 16 pin outs, as shown in Figure 1.

Pin No.	Symbol	Level	Description
1	VSS	0V	Ground.
2	VDD	+5.0V	Power supply for logic operating.
3	V0	--	Adjusting supply voltage for LCD driving.
4	RS	H/L	A signal for selecting registers: 1: Data Register (for read and write) 0: Instruction Register (for write), Busy flag-Address Counter (for read).
5	R/W	H/L	R/W = “H”: Read mode. R/W = “L”: Write mode.
6	E	H/L	An enable signal for writing or reading data.
7	DB0	H/L	This is an 8-bit bi-directional data bus.
8	DB1	H/L	
9	DB2	H/L	
10	DB3	H/L	
11	DB4	H/L	
12	DB5	H/L	
13	DB6	H/L	
14	DB7	H/L	
15	LED+	+5.0V	Power supply for backlight.
16	LED-	0V	The backlight ground.

Figure 1. LCD Character Display Pinouts

To write a character to the display you must first send a series of initialization commands then set the register select to 1 (data register) and send an 8-bit encoding of the ASCII character you wish to display. The letter encodings can be found in Appendix A.1. The initialization commands are shown in Table 1. Once you set the new register select, read/write, and data bit registers, you must write the enable high then low to send the data to the sensor.

reg_select	read_write	data_bits	command
Initialization			
0	0	0011xxxx	function set (only reads upper 4-bits)
0	0	0011xxxx	function set (only reads upper 4-bits)
0	0	0011xxxx	function set (only reads upper 4-bits)
0	0	001 DL N F xx	function set data length: 8-bit (DL = 1), 4-bit (DL = 0) display line: 2-line (N = 1), 1-line (N = 0) display font type: 5x10 (F = 1), 5x8 (F = 0)
0	0	00001000	turn display off
0	0	00000001	clear display
0	0	000001 I/D S	entry mode assign cursor moving direction and shift Increments (I/D = 1), Decrements (I/D = 0) Right Align (S = 1, I/D = 0), Left Aligh (S = 1, I/D = 1)
0	0	00001 D C B	display on set display (D), cursor (C), and blinking of cursor (B)
Write Character			
1	0	xxxxxxxx 01000001	ascii encoding of character write “A”

Table 1. LCD Character Display Commands

RFID Sensor

The purpose of the RFID sensor was to act as a means to “purchase” the items in our vending machine. The simplified block diagram of the RC522 RFID Sensor can be seen in Figure 2. Contactless UART protocol used to communicate between the sensor and the RFID tags. We used SPI protocol to communicate between the MCU and the RFID sensor. To use the RFID sensor you must first initialize the SPI connection and the RC522 sensor and turn the antenna on. Then you can begin sending commands to the CommandReg. Table 2 shows the different commands for the RC522

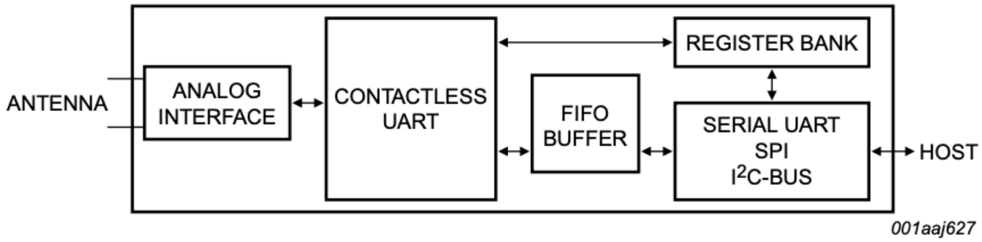


Figure 2. Simplified Block Diagram of the MIFARE RC522 RFID Sensor

Command	Command code	Action
Idle	0000	no action, cancels current command execution
Mem	0001	stores 25 bytes into the internal buffer
Generate RandomID	0010	generates a 10-byte random ID number
CalcCRC	0011	activates the CRC coprocessor or performs a self test
Transmit	0100	transmits data from the FIFO buffer
NoCmdChange	0111	no command change, can be used to modify the CommandReg register bits without affecting the command, for example, the PowerDown bit
Receive	1000	activates the receiver circuits
Transceive	1100	transmits data from FIFO buffer to antenna and automatically activates the receiver after transmission
-	1101	reserved for future use
MFAuthent	1110	performs the MIFARE standard authentication as a reader
SoftReset	1111	resets the MFRC522

Table 2. RC522 Command Overview

To communicate with the RFID Tag (PICC) we used the Tranceive command. This command transmits data stored in the FIFO buffer and receives data from the PICC which is then stored back in the FIFO buffer. To read the UID of the PICC you must follow the state diagram shown in Figure 3.

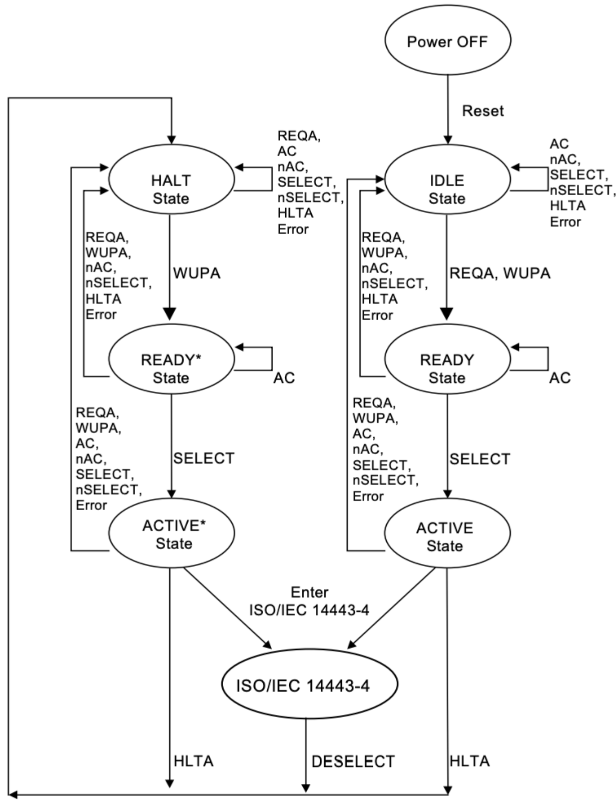


Figure 3. PICC Type A State Diagram

When a PICC is presented to the sensor (PCD) the PCD must send the wakeup command (WUPA) to put the PICC in the Ready state. The PCD must then send the select command to the PICC. The select command returns the UID of the PICC. The PCD must then send the HALT command (HLTA) to put the PICC in the Halt state. So that it can be read again. To send a command to the PICC the PCD writes the data to be sent to the FIFO buffer then writes the Transcieve command to the Command register.

Schematic

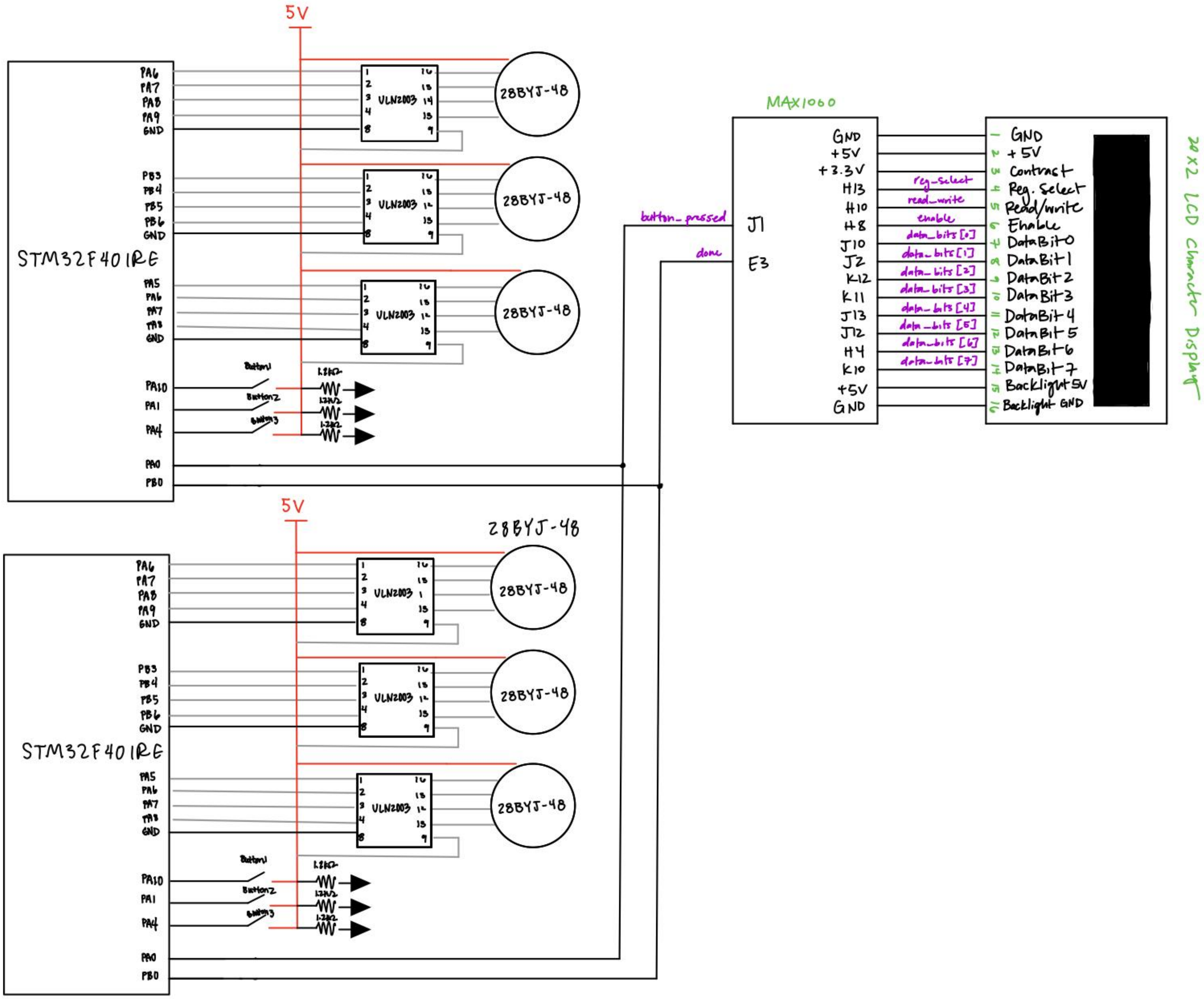


Figure 4. Full Schematic for Vending Machine

Microcontroller Design

RFID Sensor

For the RFID sensor we ran into issues communicating with the PICC. We were able to write the .c and .h files for the new sensor. The .c file contained the following functions; writeRegister(reg, value), readRegister(reg), wakeUpTag(), selectTag(uid), haltTag(), and rc522Init(). We were able to confirm that our writeRegister and readRegister functions were working correctly by scoping them using the logic analyzer. Figure 5 shows that we were able to correctly write 0x7F to the FIFODataReg and read that value back. This showed us that our SPI connection between the MCU and the RC522 was working

Figure 5. Logic Analyzer of PCD SPI Connection

We ran into issues with this sensor when we tried to communicate between the PCD and the PICC. We tried to run the wake up command which put 0x52 in the FIFO buffer. When a card is presented the PCD should receive data from the card (i.e the FIFODataReg should no longer read 0x52). We suspect there were issues in our initialization steps since we never got the FIFO buffer to change when a card was present. Figure 6 shows the scope from the logic analyzer when we ran the wakeUpTag command and presented a card to the reader. As you can see the FIFODataReg continues to read 0x52.

Figure 6. Logic Analyzer of wakeUpTag function

Stepper Motors

The MCU drives 3 28BYJ-48 stepper motors with 4 pins for each motor. A second MCU was needed to increase the number of available GPIO pins. The MCU was selected to control the motors because designing the motor drivers would be simple.

The 28BYJ-48 motors have 5 pins, 4 of which each connect to a magnetic coil inside the motor, and the 5th connects to an external 5V power supply. When a pin goes high, it causes current to flow through the magnetic coil, creating a magnetic field which attracts the nearest teeth of the cogged wheel (Stepper Motor Basics). This turns the gear and therefore the motor. The sequence of pins determines the direction that the motor turns and the speed of the pulses determines the speed of the motor. The gear ratio of the motor is 1/64, so it takes 512 wave mode cycles for a full rotation (28BYJ-48 -- 5V Stepper Motor). We drove the motors in wave mode, where each pin goes high sequentially, as shown in Figure 7. Wave mode is the simplest mode but it also provides high torque. Due to the fact that the timing of these pulses is extremely important, I directly used statements in the code to set the bits of the ODR register to write the pin high. This solved an issue with timing when using the function DigitalWrite.

Figure 7. Logic Analyser of Stepper Motor Signals

The MCU controls the motors, takes in signals from the buttons, and sends signals to the FPGA. The MCU checks every 1ms if a button is pressed. If it detects a signal, it waits another 1ms and checks again, which ensures that there are no false signals and the signal is debounced. Upon detecting a button press, it determines the corresponding motor and begins driving the motor in wave mode to turn it. The motors are rated at 100Hz, so we used a 2ms delay and slowed down the clock to ensure the gear turns fully before the next pin goes high. When a button is pressed, it also sends a 100ms pulse to the FPGA, which has a slower clock. The motors turn 612 rotations, which we determined through mechanical testing to be an appropriate amount to dispense an item. Once the motor has turned fully, it sends another 100ms pulse to the FPGA to signal that the motor is done. The MCU then resets all of the pins and begins looking for another button press input.

FPGA Design

The FPGA was used to drive the LCD Character Display. We used the FPGA for this sensor since there were specific timing constraints that needed to be met. These timing constraints are easiest to meet on the FPGA using a state machine.

To meet these timing constraints we generated a slow clock by dividing our 16MHz by 262144 to create a 61 Hz clock. A diagram of how we created our slow clock is shown in Figure 8. In our code, we set clk_divide to an 19-bit binary number. Since our clock runs at 61 Hz we have 16ms in between each clock cycle. This meets the longest timing requirement which requires us to wait 15 ms after powering the device on to input data.

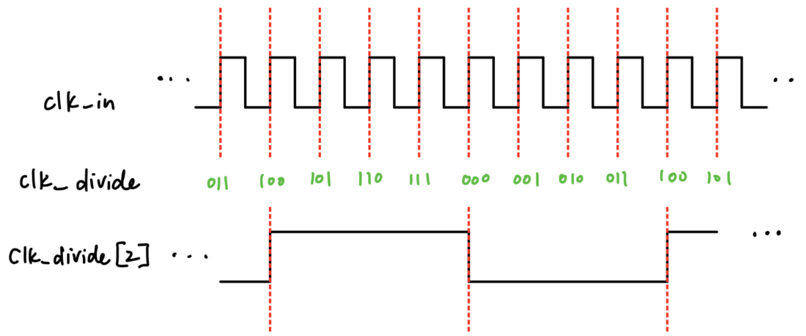


Figure 8. Slow clock signal

At every positive slow clock edge the FPGA moves to the next state and the new values for register select, read/write, and data_bits [7:0] are shifted into the registers. The state machine can be seen in Figure 9.

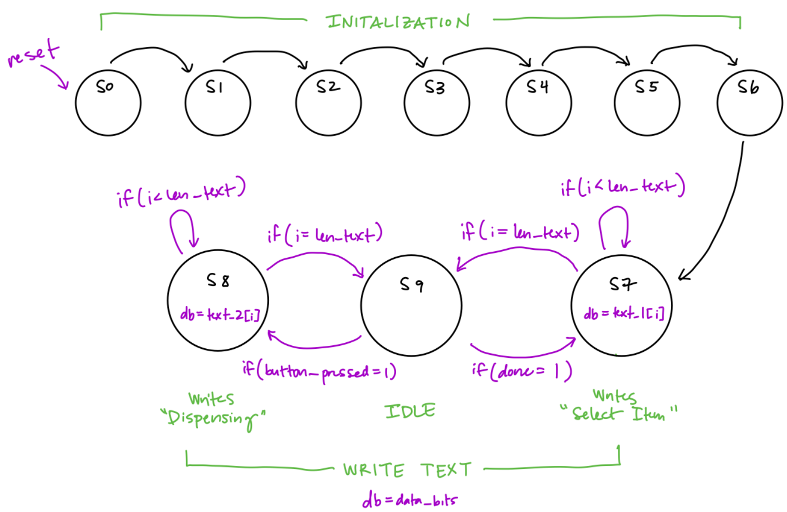


Figure 9. Finite State Machine for LCD Character Display

State 0 through 7 step through the initialization commands. In state 7, the FPGA writes "SELECT ITEM" to the display and steps into S9. S9 is the idle state. The FPGA waits in this state until it receives either a button pressed or done input signal. If the a button is pressed the button_pressed signal will go HIGH and the FPGA will step to S8 where "Dispensing. . ." gets written to the display. It then waits in S9 until it receives the "done" signal from the MCU indicating that the motors have turned off and the item has been dispensed. Once the FPGA receives the done signal it jumps to S7 to write "Select Item" on the display before waiting for the next button to be pressed.

One of the tricky parts of this sensor is setting the enable pin correctly so that it meets the timing constraints. A high level overview of the process is as follows:

1. Set register select, read/write, data bits and wait at least 40 ms to let them settle
2. Bring enable high and hold for at least 230ns
3. Bring enable low and leave data stable for at least 10 ns
4. Wait a minimum of 40µs (for character commands) or 1.64ms (for instruction commands) before entering the next byte.

To do this I set the enable pin high for one fast clock cycle. This can be seen in the simulated signals in Figure 10.

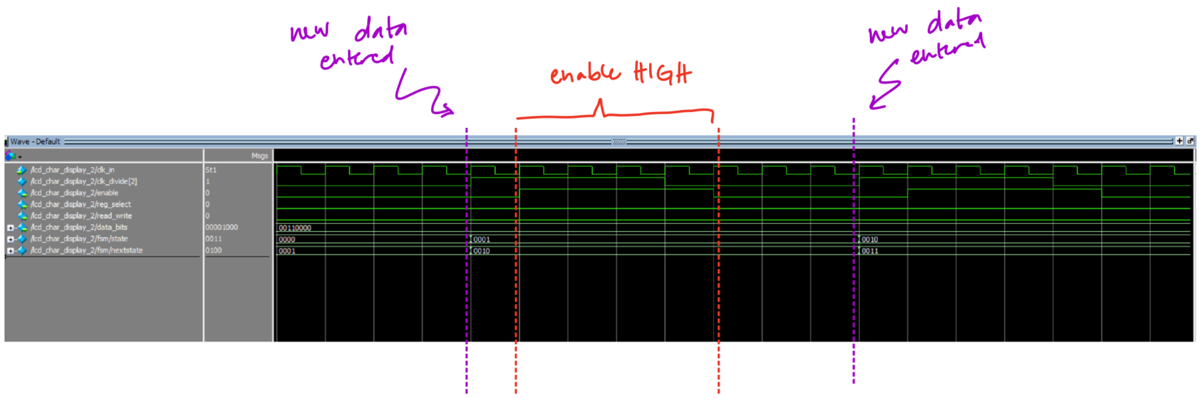


Figure 10. ModelSim signals for LCD FSM

With RFID

Here's a general overview of how the RFID would be added to our system

- The user would first press a button selecting their item.
- The display would then show "Scan tag" prompting the user to scan their RFID tag
- The RFID sensor would read the tag and check whether or not the UID of the tag was on the list of acceptable ids.
- If so, the MCU would write the "tag_accepted" signal high and send the signal to the FPGA.
- The FPGA would take the tag_accepted input and use that to determine what text to display on the LCD

A block diagram of the FPGA can be seen in Figure 11.

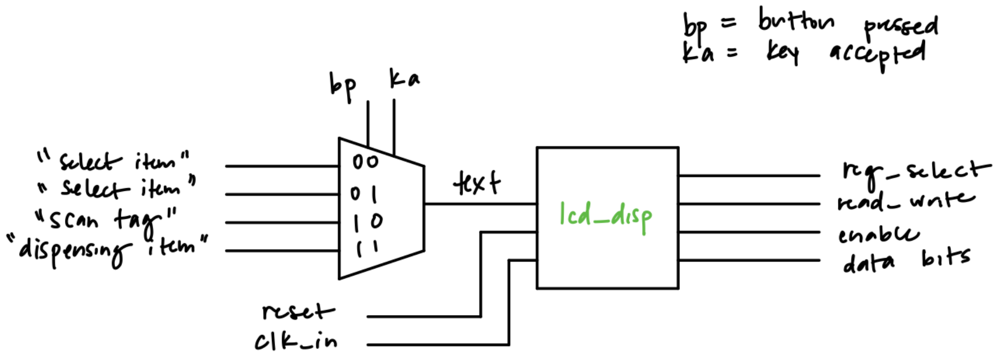


Figure 11. Block Diagram of FPGA

Mechanical Design

The vending machine is designed to be aesthetically pleasing and easy to use. The front of the machine catches the user's eye with a large window to view the items on the left and the LCD and buttons on the right. The user simply follows the instructions on the machine to "SELECT ITEM" by pressing a button, then the item begins dispensing and the LCD displays "DISPENSING…". The spiral dispensers turn and the item the user selects falls to the bottom of the left side, where the user can reach into a small opening and pick it up.

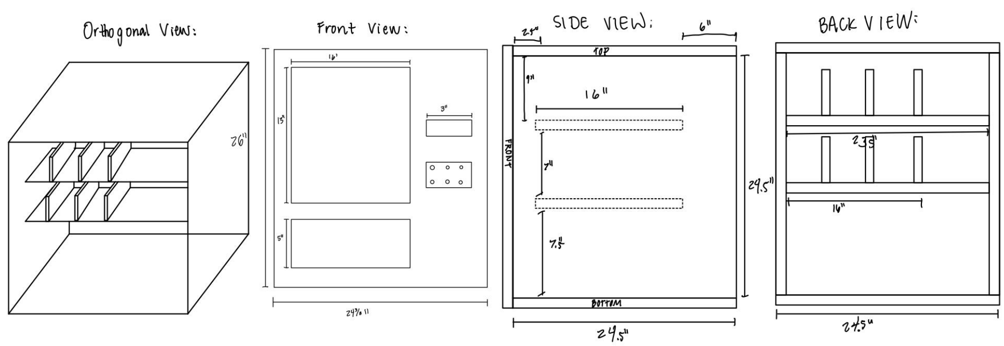


Figure 12. Vending Machine Housing Drawing

Primarily made of particle board, the box and shelves that house the electronics and dispensers is approximately 2'x2'x2'. The 6 shelves inside are designed to dispense items up to 4"x6". The spiral dispensers are 12" long with 5-6 locations in each spiral dispenser for the operator to place items. The dispensers are made of ¼" copper wire and mounted to a cardboard disk with hot glue. The center of the disk is glued to the motors so that the spiral turns as the motor turns. The front of the vending machine is cardboard with holes cut out for the window, LCD display, buttons, and slot to retrieve items.

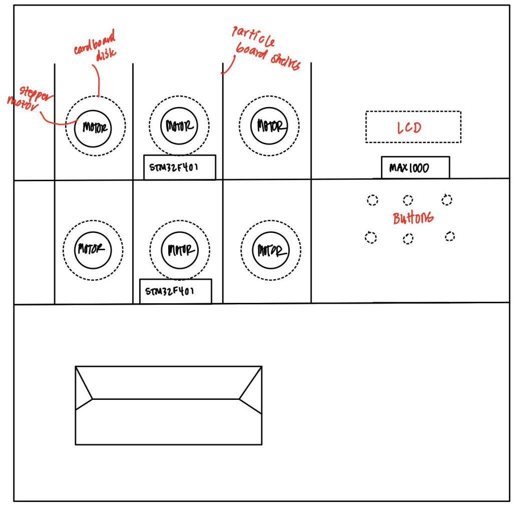


Figure 13. Vending Machine Setup

Results

Our final product is a working vending machine with 6 buttons, 5 motors, and an LCD display. The user simply presses a button and the vending machine dispenses an item while the LCD displays, “Dispensing…”, and once the motor stops turning the LCD returns to displaying “Select Item”. The RFID sensor was not functional at checkoff time but with more time we hope to figure out how to utilize the contactless UART connection to communicate the PICC and PCD and integrate it into our system.

The LCD display proved to be especially tricky. We created and debugged an FSM. The FSM forced us to think about the hardware implementation of our logic. This especially tripped us up when we had to write a flip flop for the i variable which we used to iterate over the characters in the string. This FSM gave us great practice at using ModelSim to debug our errors. Another tricky part of this sensor was making sure all the timing constraints were met and that the data was stable before writing the enable signal HIGH. This was done by utilizing a combination of a slow and fast clock.

The RFID sensor gave us great practice with debugging an SPI connection using a logic analyzer. Through this debugging process we were able to solidify our understanding of clock phase and polarity. Overall we were able to successfully communicate between the RFID sensor and the MCU. We eventually ran out of time and were unable to successfully communicate between the PICC and the PCD. We believe this was due to missing steps in our initialization of the RC522.

We originally planned to use an enable for each motor and use the same 4 GPIO pins to power the magnetic coils on each motor. This would use less GPIO pins and only require one MCU. However, after testing with some transistors, we destroyed an MCU by accidentally drawing too much current by trying to power all of the motors at once. Instead, we could use an H-bridge to select between motors and reduce the number of pins required.

The 28BYJ-48 motors provide the exact amount of torque necessary to turn the spiral dispensers. For the final demonstration, we decided to dispense paper so that the motors are not overpowered and unable to turn due to the extra weight. To improve this, we might increase the voltage of the motors or add a gear system to increase the torque applied to the spirals.

The mechanical design of the vending machine could use some improvements. The front of the vending machine is cardboard, but a particle board front would be sturdier and more secure. This could be attached with hinges and a latch that locks. Most of the motors and electronics are mounted with tape, but we would like to use screws for a more permanent and reliable design. The cardboard disk that connects the motor and spiral is hot glued together but this could be improved with a plastic gear that doesn't bend and a perfect slot for the motor dowel. The bottom slot to get the item could have a board to block the user from reaching up to grab an item from the machine.

References

Heymsfeld, Ralph. "Adding a Character LCD to an FPGA Project." *The Robot Diaries*, 21 March 2019, http://robotics.hobbizine.com/fpgalcd.html.

LCM Module TC1602A-09T Datasheet. 4 6 2009. *TC1602A-09T SpecV00 2009-06-04*, Tinsharp Industrial Co., Ltd., https://cdn-shop.adafruit.com/product-files/181/p181.pdf.

Stepper Motor Basics. *6, 1-6*, Industrial Circuits, https://www.geeetech.com/Documents/Stepper%20motor%20basic.pdf.

28BYJ-48 -- 5V Stepper Motor. *28BYJ 48 Stepper Motorx Motor Manual*, https://usermanual.wiki/Pdf/Stepper20Motor20Manual.1122402138/view.

ULN2003 Datasheet. December 1976. *High Voltage High Current Darlington Transistor Arrays*, Texas Instruments, https://www.geeetech.com/Documents/ULN2003%20datasheet.pdf.

Bill of Materials

Item	Description	Quantity	Vendor	Price
28BYJ-48 Motors and ULN2003 Drivers	Motors and Transistor Array	6	Amazon	$12.99
Mifare RC522 RF IC Card Sensor Module	RFID Sensor	6	Amazon	$5.49
NFC Smart Card tag Tags 1k S50 IC 13.56MHz	RFID Cards	10	Amazon	$7.99
LCD Screen	LCD Screen	1	HMC Digital Lab	-
8'x4' Particle Board	Box and Shelving	1	Lowes	$25
Pushbutton Switches	Buttons	6	Amazon	$7.99
¼" Metal Dowels	Shelf Support	4	HMC Stockroom	-
Wood Glue	To bond shelves and housing	N/A	HMC Machine Shop	-
1.2kOhm Resistors	Pulldown Resistors for Buttons	6	HMC Digital Lab	-
Extra Large Cardboard Box	Vending Machine front and wire mount	1	HMC Recycling Bin	-
Tape and Hot Glue	For mounting electronics and dispensers	N/A	HMC Makerspace	-
2"x8" Breadboard	2 for motors, 1 for LCD	3	Pre-owned	-
10' Copper Wire, 1'4" diameter	Spiral Dispensers	1	Lowes	$8.99
STM32F401RE	Microcontroller	2	E155 Kit	-
MAX1000	FPGA	1	E155 Kit	-
Miles and Miles of Wire	Wires	N/A	HMC Digital Lab and Stockroom	-
TOTAL				**$68.45**

Appendix A: LCD Character Display

A.1 Character Encodings

Higher 4bit / Lower 4bit	0000	0010	0011	0100	0101	0110	0111	1010	1011	1100	1101	1110	1111
××××0000			0	@	P	`	p		ー	タ	ミ	α	p
××××0001		!	1	A	Q	a	q	。	ア	チ	ム	ä	q
××××0010		"	2	B	R	b	r	「	イ	ツ	メ	β	θ
××××0011		#	3	C	S	c	s	」	ウ	テ	モ	ε	∞
××××0100		$	4	D	T	d	t	、	エ	ト	ヤ	μ	Ω
××××0101		%	5	E	U	e	u	・	オ	ナ	ユ	σ	ü
××××0110		&	6	F	V	f	v	ヲ	カ	ニ	ヨ	ρ	Σ
××××0111		'	7	G	W	g	w	ァ	キ	ヌ	ラ	g	π
××××1000		(	8	H	X	h	x	ィ	ク	ネ	リ	√	x̄
××××1001		)	9	I	Y	i	y	ゥ	ケ	ノ	ル	⁻¹	y
××××1010		*	:	J	Z	j	z	ェ	コ	ハ	レ	j	千
××××1011		+	;	K	[	k	{	ォ	サ	ヒ	ロ	ˣ	万
××××1100		,	<	L	¥	l			ャ	シ	フ	ワ	¢	円
××××1101		-	=	M	]	m	}	ュ	ス	ヘ	ン	£	÷
××××1110		.	>	N	^	n	→	ョ	セ	ホ	゛	ñ	
××××1111		/	?	O	_	o	←	ッ	ソ	マ	゜	ö	█

Appendix B: Verilog Code

```
/*
 E155 Final Project
 Name: Ava Sherry & Leila Wiberg
 Date: 11/10/21
*/

module lcd_char_display(
                       input logic clk_in,
                       input logic reset,
                       input logic button_pressed, done,
                       output logic [7:0] data_bits,
                       output logic reg_select, read_write, enable
                       );

   logic [18:0] clk_divide;
   always @(posedge clk_in)
       if (reset)
           begin
           clk_divide <= 18'b0;
           enable <= 0;
           end
       else
           if (clk_divide[18])
               begin
                   enable <= 1;
                   clk_divide <= clk_divide + 1;
               end
           else
               begin
                   enable <= 0;
                   clk_divide <= clk_divide + 1;
               end

   lcd_fsm fsm(clk_divide[18], reset, button_pressed, done, reg_select, read_write,
data_bits);


endmodule
```

```
module lcd_fsm(
                   input logic clk_in,
                   input logic reset,
                   input logic button_pressed, done,
                   output logic reg_select, read_write,
                   output logic [7:0] data_bits
                   );

   logic [7:0] i;
   logic [7:0] text[15:0];
   logic [7:0] text_1[15:0] = '{" ", " "," ", "M","E","T","I", "
","T","C","E","L","E","S"," ", " "};
   logic [7:0] text_2[15:0] = '{" ", " ",".", ".",
".","G","N","I","S","N","E","P","S","I","D"," "};
   logic [7:0] len_text = 8'b00010000;
   logic [3:0] state, nextstate;

   parameter S0 = 4'b0000;
   parameter S1 = 4'b0001;
   parameter S2 = 4'b0010;
   parameter S3 = 4'b0011;
   parameter S4 = 4'b0100;
   parameter S5 = 4'b0101;
   parameter S6 = 4'b0110;
   parameter S7 = 4'b0111;
   parameter S8 = 4'b1000;
   parameter S9 = 4'b1001;


   // State Register
   always_ff @(posedge clk_in or posedge reset)
       if (reset)  state <= S0;
       else            state <= nextstate;

   // i Flip-Flop
   always_ff @(posedge clk_in)
       if (state == S7 && i != len_text) i <= i + 1;
       else if (state == S9 && i == len_text) i <= 0;
       else if (state == S8 && i != len_text) i <= i + 1;
       else i <= 0;
```

```
// Next State Logic
always_comb
    case (state)
        // Initialize LCD Display (S0-S6)
        S0:     begin                                       // ** INITIALIZE **
                    reg_select <= 0;                        // instruction register
                    read_write <= 0;                        // to write data
                    data_bits <= 8'b00110000;               // initialization
                    nextstate <= S1;
                end
        S1:     begin                                       // ** INITIALIZE **
                    reg_select <= 0;                        // instruction register
                    read_write <= 0;                        // to write data
                    data_bits <= 8'b00110000;               // initilization
                    nextstate <= S2;
               end
       S2:      begin                                       // ** INITIALIZE **
                    reg_select <= 0;                        // instruction register
                    read_write <= 0;                        // to write data
                    data_bits <= 8'b00110000;               // initialization
                    nextstate <= S3;
               end
       S3:      begin                                       // ** TURN DISPLAY OFF **
                    reg_select <= 0;                        // instruction register
                    read_write <= 0;                        // to write data
                    data_bits <= 8'b00001000;               // display off
                    nextstate <= S4;
               end
       S4:      begin                                       // ** TO CLEAR DISPLAY **
                    reg_select <= 0;                        // instruction register
                    read_write <= 0;                        // to write data
                    data_bits <= 8'b00000001;               // clear display
                    nextstate <= S5;
               end
        S5: begin                                           // ** ENTRY MODE **
                    reg_select <= 0;                        // instruction register
                    read_write <= 0;                        // to write data
                    data_bits <= 8'b00000110;               // entry mode, assign cursor
moving direction (D)
                    nextstate <= S6;
               end
```

```
           S6: begin                                        // ** TURN DISPLAY ON **
                       reg_select <= 0;                      // instruction register
                       read_write <= 0;                      // to write data
                       data_bits <= 8'b00001100;             // turn display on and set
display
                       nextstate <= S7;
                  end
           // Write "Select Item"
           S7: begin
                   if (i == len_text)
                       begin                                // ** RETURN HOME **
                           reg_select <= 0;                  // instruction register
                           read_write <= 0;                  // to write data
                           data_bits <= 8'b00000010;         // return home
                           nextstate <= S9;
                       end
                   else
                       begin                                // ** WRITE DATA **
                           reg_select <= 1;                  // data register
                           read_write <= 0;                  // to write data
                           data_bits <= text_1[i];           // write char
                           nextstate <= S7;
                       end
                   end
           // Write "Dispensing..."
           S8: begin
                   if (i == len_text)
                       begin                                // ** RETURN HOME **
                           reg_select <= 0;                  // instruction register
                           read_write <= 0;                  // to write data
                           data_bits <= 8'b00000010;         // return home
                           nextstate <= S9;
                       end
                   else
                       begin                                // ** WRITE DATA **
                           reg_select <= 1;                  // data register
                           read_write <= 0;                  // to write data
                           data_bits <= text_2[i];           // write char
                           nextstate <= S8;
                       end
                   end
           // Idle
```

```
            S9:     begin
                    if (done == 1)
                        begin                           // ** CLEAR DISPLAY **
                            reg_select <= 0;            // instruction register
                            read_write <= 0;            // to write data
                            data_bits <= 8'b00000001;   // clear display
                            nextstate <= S7;
                        end
                    else if (button_pressed == 1)
                        begin                           // ** CLEAR DISPLAY **
                            reg_select <= 0;            // instruction register
                            read_write <= 0;            // to write data
                            data_bits <= 8'b00000001;   // clear display
                            nextstate <= S8;
                        end
                    else
                        begin                           // ** RETURN HOME **
                            reg_select <= 0;            // instruction register
                            read_write <= 0;            // to write data
                            data_bits <= 8'b00000010;   // return home
                            nextstate <= S9;
                        end
                    end
            default:
                    nextstate <= S0;

        endcase
endmodule
```

Appendix C: Microcontroller Code

C.1 Stepper Motors

The complete software include RCC.h, RCC.c, GPIO.h, GPIO.c, and main.c can be viewed at https://github.com/lwiberg/vending-machine. The files besides main.c are from previous labs found on the class github.

Main.c

```
#include "STM32F401RE_RCC.h"
#include "STM32F401RE_GPIO.h"
```

```
#define NUM_STEPS  612  //number of steps to dispense (512=full rotation)
#define MS_DELAY    2   //delay between steps

#define BUTTON_1 10
#define BUTTON_2 1
#define BUTTON_3 4
#define BUTTON_PRESSED 0 //GPIOA, Analog Pin 1
#define DONE 0 //GPIOB, Analog Pin 4

void initializeGPIO()
{
    //Set Up Clock
    RCC->CFGR.PPRE2 = 0b000;   //APB High Speed Prescaler = 0
    RCC->CFGR.HPRE  = 0b1001;  //AHP Prescaler = 4
    RCC->AHB1ENR.GPIOAEN = 1;  //turn on clock to GPIOA
    RCC->AHB1ENR.GPIOBEN = 1;

    //Set pins to output mode
    //MOTOR 1
    pinMode(GPIOA, 6, GPIO_OUTPUT);
    pinMode(GPIOA, 7, GPIO_OUTPUT);
    pinMode(GPIOA, 8, GPIO_OUTPUT);
    pinMode(GPIOA, 9, GPIO_OUTPUT);

    //MOTOR 2
    pinMode(GPIOB, 3, GPIO_OUTPUT);
    pinMode(GPIOB, 4, GPIO_OUTPUT);
    pinMode(GPIOB, 5, GPIO_OUTPUT);
    pinMode(GPIOB, 6, GPIO_OUTPUT);

    //MOTOR 3
    pinMode(GPIOA, 5, GPIO_OUTPUT);
    pinMode(GPIOB, 8, GPIO_OUTPUT);
    pinMode(GPIOB, 9, GPIO_OUTPUT);
    pinMode(GPIOB, 10, GPIO_OUTPUT);

    //Buttons
    pinMode(GPIOA, 10, GPIO_INPUT); //Button 1
    pinMode(GPIOA, 1, GPIO_INPUT); //Button 2
    pinMode(GPIOA, 4, GPIO_INPUT); //Button 3

    //Signals to FPGA
    pinMode(GPIOB, 0, GPIO_OUTPUT); //Done to FPGA
    pinMode(GPIOA, 0, GPIO_OUTPUT); //Button Pressed to FPGA

}

void ms_delay(int ms) {
   while (ms-- > 0) {
```

```
      volatile int x=1000;
      while (x-- > 0)
         __asm("nop");}
}

int get_button_press(){
    ms_delay(2); //debounce
    if (digitalRead(GPIOA, BUTTON_1)>0) return 1;
    if (digitalRead(GPIOA, BUTTON_2)>0) return 2;
    if (digitalRead(GPIOA, BUTTON_3)>0) return 3;

    return 0;
}

void one_step_1(){
    //wave mode
    GPIOA->ODR &= (0x7<<6); //reset pins to 0
    GPIOA->ODR |= (0x1<<6); //pin pa6 high (in 1 to motor 1)
    ms_delay(MS_DELAY);
    GPIOA->ODR &= (0xE<<6);
    GPIOA->ODR |= (0x1<<7);
    ms_delay(MS_DELAY);
    GPIOA->ODR &= (0xD<<6);
    GPIOA->ODR |= (0x1<<8);
    ms_delay(MS_DELAY);
    GPIOA->ODR &= (0xB<<6);
    GPIOA->ODR |= (0x1<<9);
    ms_delay(MS_DELAY);
}

void one_step_2(){
    //wave mode
    GPIOB->ODR &= (0x7<<3); //reset pins to 0
    GPIOB->ODR |= (0x1<<3); //set pb3 high (in 1 to motor 2)
    ms_delay(MS_DELAY);
    GPIOB->ODR &= (0xE<<3);
    GPIOB->ODR |= (0x1<<4);
    ms_delay(MS_DELAY);
    GPIOB->ODR &= (0xD<<3);
    GPIOB->ODR |= (0x1<<5);
    ms_delay(MS_DELAY);
    GPIOB->ODR &= (0xB<<3);
    GPIOB->ODR |= (0x1<<6);
    ms_delay(MS_DELAY);
}

void one_step_3(){
    //wave mode
    GPIOB->ODR &= (0x7<<7); //reset GPIOB pins to 0
```

```c
    GPIOA->ODR &= (0x7<<5); //reset GPIOA pin to 0
    GPIOA->ODR |= (0x1<<5); //set pa5 high (in 1 to motor 3)
    ms_delay(MS_DELAY);
    GPIOB->ODR &= (0xE<<7);
    GPIOA->ODR &= (0xE<<5);
    GPIOB->ODR |= (0x1<<8);
    ms_delay(MS_DELAY);
    GPIOB->ODR &= (0xD<<7);
    GPIOA->ODR &= (0xE<<5);
    GPIOB->ODR |= (0x1<<9);
    ms_delay(MS_DELAY);
    GPIOB->ODR &= (0xB<<7);
    GPIOA->ODR &= (0xE<<5);
    GPIOB->ODR |= (0x1<<10);
    ms_delay(MS_DELAY);
}

void dispense(int motor){
    for(volatile int i; i<NUM_STEPS; i = i+1){
        if (motor == 1) one_step_1();
        if (motor == 2) one_step_2();
        if (motor == 3) one_step_3();
    }
    digitalWrite(GPIOB, DONE, 1);
    ms_delay(100);
}

int main(void)
{
    initializeGPIO();

    while (1){
        digitalWrite(GPIOB, DONE, 0); //reset
        digitalWrite(GPIOA, BUTTON_PRESSED, 0); //reset
        int motor;
        motor = get_button_press();
        if (motor > 0){
            digitalWrite(GPIOA, BUTTON_PRESSED, 1); //send signal to FPGA
            ms_delay(100);
            digitalWrite(GPIOA, BUTTON_PRESSED, 0); //send signal to FPGA
            if (motor == 1) dispense(1); //turn on motor 1
            if (motor == 2) dispense(2); //turn on motor 2
            if (motor == 3) dispense(3); //turn on motor 3
        }
        motor = 0;
    }
}
```

C.2 RFID Sensor

MIFARE_RC522.c

```
// MIFARE_RC522.c
// RC522 function declarations

#include "STM32F401RE_SPI.h"
#include "MIFARE_RC522.h"
#include "STM32F401RE_GPIO.h"


void rc522Init() {
   // perform hard reset
   pinMode(GPIOA, 1, GPIO_OUTPUT);          // PA4 --> OUTPUT
   digitalWrite(GPIOA, 1, 0);
   delay(1);
   digitalWrite(GPIOA, 1, 1);
   delay(50);

   // Reset baud rates
   writeRegister(TxModeReg, 0x00);
   writeRegister(RxModeReg, 0x00);

   // Reset ModWidthReg
   writeRegister(ModWidthReg, 0x26);
   writeRegister(TModeReg, 0x80);      // TAuto=1; timer starts automatically at the
end of the transmission in all communication modes at all speeds
   writeRegister(TPrescalerReg, 0xA9); // TPreScaler = TModeReg[3..0]:TPrescalerReg,
ie 0x0A9 = 169 => f_timer=40kHz, ie a timer period of 25μs.
   writeRegister(TReloadRegH, 0x03);   // Reload timer with 0x3E8 = 1000, ie 25ms
before timeout.
   writeRegister(TReloadRegL, 0xE8);
   writeRegister(TxASKReg, 0x40);      // Default 0x00. Force a 100 % ASK modulation
independent of the ModGsPReg register setting
   writeRegister(ModeReg, 0x3D);       // Default 0x3F. Set the preset value for the
CRC coprocessor for the CalcCRC command to 0x6363 (ISO 14443-3 part 6.2.4)
   writeRegister(CommandReg, 0x00);    // Switch analog reciever on
   antennaOn();                        // Enable the antenna driver pins TX1 and TX2
(they were disabled by the reset)
}


void antennaOn() {
```

```
   uint8_t value = readRegister(TxControlReg);
   if ((value & 0x03) != 0x03) {
       writeRegister(TxControlReg, value | 0x03);
   }
}


uint8_t readRegister(uint8_t reg) {
   digitalWrite(GPIOA, 4, 0);                         // write NSS pin low
   spiSendReceive(0x80 | (reg << 1));                 // Address must be in form 1xxxxxx0
for read mode where xxxxxx is the address
   uint8_t value = spiSendReceive(0xAA);              // Read the value
   digitalWrite(GPIOA, 4, 1);                         // write NSS pin high
   return value;
   delay(50);
}


void readRegisterMulti(uint8_t reg, uint8_t count, uint8_t *values) {
   digitalWrite(GPIOA, 4, 0);                                 // write NSS pin low
   spiSendReceive(0x80 | (reg << 1));  // Address must be in form 1xxxxxx0 for read
mode where xxxxxx is the address
   for (uint8_t index = 0; index < count +1; index++) {
       values[index] = spiSendReceive(0xAA);
   }
   digitalWrite(GPIOA, 4, 1);                                 // write NSS pin high
   delay(50);
}


void writeRegister(uint8_t reg, uint8_t value) {
   digitalWrite(GPIOA, 4, 0);              // write NSS pin low
   spiSendReceive(0x00 | (reg << 1));      // Address must be in form 0xxxxxx0 for
write mode where xxxxxx is the address
   spiSendReceive(value);                  // write value to register
   digitalWrite(GPIOA, 4, 1);              // write NSS pin high
   delay(50);
}


void writeRegisterMulti(uint8_t reg, uint8_t count, uint8_t *values) {
   digitalWrite(GPIOA, 4, 0);              // write NSS pin low
```

```
   spiSendReceive(0x00 | (reg << 1));      // Address must be in form 0xxxxxx0 for
write mode where xxxxxx is the address
   for (uint8_t index = 0; index < count; index++) {
       spiSendReceive(values[index]);
   }
   digitalWrite(GPIOA, 4, 1);               // write NSS pin high
   delay(50);
}


// sets specific bits of register value without altering rest of bits
// xxxxxxxx | 00000101 --> xxxxx1x1     [mask = 00000101]
void setRegisterBitMask(uint8_t reg, uint8_t mask) {
   uint8_t tmp;
   tmp = readRegister(reg);
   writeRegister(reg, tmp | mask);
}


// clears specific bits of register value without altering rest of bits
// xxxxxxxx & (~00000101) --> xxxxx0x0  [mask = 00000101]
void clearRegisterBitMask(uint8_t reg, uint8_t mask) {
   uint8_t tmp;
   tmp = readRegister(reg);
   writeRegister(reg, tmp & (~mask));
}


// Transmits WAKE UP command, Type A. Puts tags in state IDLE or HALT into state READY
uint8_t wakeUpTag() {
   uint8_t fifo;
   clearRegisterBitMask(CollReg, 0x80);             // Sets ValuesAfterColl = 0
   writeRegister(FIFODataReg, 0x52);                // Tag WAKE UP command
(PICC_CMD_WUPA)
   writeRegister(CommandReg, PCD_TRANSCEIVE);       // Transmit data from FIFO buffer
   setRegisterBitMask(BitFramingReg, 0x80);         // StartSend=1, transmission of
data starts
   fifo = readRegister(FIFODataReg);
   return fifo;
}

void selectTag(uint8_t *uid) {
```

```
   uint8_t buffer[9];                  // The SELECT/ANTICOLLISION commands uses a 7
byte standard frame + 2 bytes CRC_A
   uint8_t rxAlign;                    // Used in BitFramingReg. Defines the bit
position for the first bit received.
   uint8_t txLastBits;             // Used in BitFramingReg. The number of valid bits
in the last transmitted byte.
   uint8_t index = 2;

   // Description of buffer structure:
   //      Byte 0: SEL                 Indicates the Cascade Level: PICC_CMD_SEL_CL1,
PICC_CMD_SEL_CL2 or PICC_CMD_SEL_CL3
   //      Byte 1: NVB                 Number of Valid Bits (in complete command, not
just the UID): High nibble: complete bytes, Low nibble: Extra bits.
   //      Byte 2: UID-data or CT      See explanation below. CT means Cascade Tag.
   //      Byte 3: UID-data
   //      Byte 4: UID-data
   //      Byte 5: UID-data
   //      Byte 6: BCC                 Block Check Character - XOR of bytes 2-5
   //      Byte 7: CRC_A
   //      Byte 8: CRC_A
   // The BCC and CRC_A are only transmitted if we know all the UID bits of the
current Cascade Level.
   //
   // Description of bytes 2-5: (Section 6.5.4 of the ISO/IEC 14443-3 draft: UID
contents and cascade levels)
   //      UID size    Cascade level   Byte2   Byte3   Byte4   Byte5
   //      ========    =============   =====   =====   =====   =====
   //       4 bytes        1           uid0    uid1    uid2    uid3



   // Prepare MFRC522
   clearRegisterBitMask(CollReg, 0x80);        // ValuesAfterColl=1 => Bits received
after collision are cleared.
   buffer[0] = PICC_CMD_SEL_CL1;



   // Repeat anti collision loop until we can transmit all UID bits + BCC and receive
a SAK - max 32 iterations.
   // This is an ANTICOLLISION.
   txLastBits      = 0;
   buffer[1]       = (index << 4) + txLastBits;    // NVB - Number of Valid Bits
```

```
  // Set bit adjustments
  rxAlign = txLastBits;                                         // Having a
separate variable is overkill. But it makes the next line easier to read.
  writeRegister(BitFramingReg, (rxAlign << 4) + txLastBits);      // RxAlign =
BitFramingReg[6..4]. TxLastBits = BitFramingReg[2..0]

  // Transmit the buffer and receive the response.
  writeRegisterMulti(FIFODataReg, 9, buffer);
  writeRegister(CommandReg, PCD_TRANSCEIVE);      // Transmit data from FIFO buffer
  setRegisterBitMask(BitFramingReg, 0x80);        // StartSend=1, transmission of
data starts
  readRegisterMulti(FIFODataReg, 9, buffer);

  // Copy the found UID bytes from buffer[] to uid
  // index            = (buffer[2] == PICC_CMD_CT) ? 3 : 2; // source index in
buffer[]
  // bytesToCopy      = (buffer[2] == PICC_CMD_CT) ? 3 : 4;

  uint8_t bytesToCopy = 4;
  uint8_t uidArr[4];
  uint8_t count;
  for (count = 0; count < bytesToCopy; count++) {
    uidArr[count] = buffer[index++];
  }
  *uid = uidArr;
}


void haltTag() {
  uint8_t buffer[4];

  // Build command buffer
  buffer[0] = PICC_CMD_HLTA;
  buffer[1] = 0x00;
  // Calculate CRC_A
  calculateCRC(buffer, 2, &buffer[2]);

  writeRegisterMulti(FIFODataReg, 4, buffer);
  writeRegister(CommandReg, PCD_TRANSCEIVE);      // Transmit data from FIFO buffer
```

```
   setRegisterBitMask(BitFramingReg, 0x80);        // StartSend=1, transmission of
data starts

}



void calculateCRC(uint8_t *data, uint8_t length, uint8_t *result) {
   writeRegister(CommandReg, PCD_IDLE);        // Stop any active command.
   writeRegister(DivIrqReg, 0x04);             // Clear the CRCIRq interrupt request
bit
   writeRegister(FIFOLevelReg, 0x80);          // FlushBuffer = 1, FIFO initialization
   writeRegisterMulti(FIFODataReg, length, data);  // Write data to the FIFO
   writeRegister(CommandReg, PCD_CALCCRC);     // Start the calculation


   while(!(readRegister(DivIrqReg) & 0x04));       // wait for CRCIRq bit set - i.e
calculation done
   writeRegister(CommandReg, PCD_IDLE);    // Stop calculating CRC for new content in
the FIFO.
   result[0] = readRegister(CRCResultRegL);
   result[1] = readRegister(CRCResultRegH);

}

void reset() {
   writeRegister(CommandReg, PCD_RESET);   // Issue the SoftReset command.
   // Wait for the PowerDown bit in CommandReg to be cleared
   // while (readRegister(CommandReg) & (1 << 4));
}

void delay(int clkCycles){
   int i = 0;
   while(i<clkCycles){
       i++;
   }
}
```

MIFARE_RC522.h

```
// MIFARE_RC522.h
// Header for RC522 functions

#ifndef RC522_H
#define RC522_H

#include <stdint.h> // Include stdint header

// PCD (Proximity Coupling Device): MFRC522 Contactless Reader IC
// PICC (Proximity Integrated Circuit Card): card or tag

////////////////////////////////////////////////////////////////////////////////
// Bitfield structs
////////////////////////////////////////////////////////////////////////////////

// MF522 (PCD) command
#define PCD_IDLE              0x00  // no action, cancels current command execution
#define PCE_MEM               0x01  // stores 25 bytes into the internal buffer
#define PCD_GENRANDOMID       0x02  // generates a 10-byte random ID number
#define PCD_CALCCRC           0x03  // activates the CRC coprocessor or performs a
self test
#define PCD_TRANSMIT          0x04  // transmits data from the FIFO buffer
#define PCD_NOCMDCHANGE       0x07  // no command change, can be used to modify the
CommandReg register bits without affecting the command, for example, the PowerDown bit
#define PCD_RECEIVE           0x08  // activates the receiver circuits
#define PCD_TRANSCEIVE        0x0C  // transmits data from FIFO buffer to antenna and
automatically activates the receiver after transmission
#define PCD_AUTHENT           0x0E  // performs the MIFARE standard authentication as
a reader
#define PCD_RESET             0x0F  // resets the MFRC522


// card (PICC) command
#define PICC_REQIDL           0x26
#define PICC_READ             0x30  // Reads one 16 byte block from the authenticated
sector of the PICC
#define PICC_HALT             0x50
#define PICC_REQALL           0x52
#define PICC_AUTHENT1A        0x60  // Perform authentication with Key A
#define PICC_AUTHENT1B        0x61  // Perform authentication with Key B
#define PICC_ANTICOLL1        0x93
```

```
#define PICC_ANTICOLL2        0x95
#define PICC_ANTICOLL3        0x97
#define PICC_WRITE            0xA0  // Writes one 16 byte block to the authenticated
sector of the PICC
#define PICC_TRANSFER         0xB0  // Writes the contents of the internal data
register to a block
#define PICC_DECREMENT        0xC0
#define PICC_INCREMENT        0xC1
#define PICC_RESTORE          0xC2  // Reads the contents of a block into the internal
data register
#define PICC_CMD_WUPA         0x52  // Wake up command
#define PICC_CMD_SEL_CL1      0x93
#define PICC_CMD_CT           0x88  // Cascade Tag
#define PICC_CMD_HLTA         0x50  // HALT command, Type A. Instructs an ACTIVE PICC
to go to state HALT.

//MF522 (PCD) registers
#define     CommandReg             0x01
#define     ComIEnReg              0x02
#define     DivlEnReg              0x03
#define     ComIrqReg              0x04
#define     DivIrqReg              0x05
#define     ErrorReg               0x06
#define     Status1Reg             0x07
#define     Status2Reg             0x08
#define     FIFODataReg            0x09
#define     FIFOLevelReg           0x0A
#define     WaterLevelReg          0x0B
#define     ControlReg             0x0C
#define     BitFramingReg          0x0D
#define     CollReg                0x0E

#define     ModeReg                0x11
#define     TxModeReg              0x12
#define     RxModeReg              0x13
#define     TxControlReg           0x14
#define     TxASKReg               0x15
#define     TxSelReg               0x16
#define     RxSelReg               0x17
#define     RxThresholdReg         0x18
#define     DemodReg               0x19
//                                 0x1A
```

```
//                                  0x1B
#define     MifareTxReg             0x1C
#define     MifareRxReg             0x1D
//                                  0x1E
#define     SerialSpeedReg          0x1F

#define     CRCResultRegH           0x21
#define     CRCResultRegL           0x22
//                                  0x23
#define     ModWidthReg             0x24
//                                  0x25
#define     RFCfgReg                0x26
#define     GsNReg                  0x27
#define     CWGsCfgReg              0x28
#define     ModGsCfgReg             0x29
#define     TModeReg                0x2A
#define     TPrescalerReg           0x2B
#define     TReloadRegH             0x2C
#define     TReloadRegL             0x2D
#define     TCounterValueRegH       0x2E
#define     TCounterValueRegL       0x2F

#define     TestSel1Reg             0x31
#define     TestSel2Reg             0x32
#define     TestPinEnReg            0x33
#define     TestPinValueReg         0x34
#define     TestBusReg              0x35
#define     AutoTestReg             0x36
#define     VersionReg              0x37
#define     AnalogTestReg           0x38
#define     TestDAC1Reg             0x39
#define     TestDAC2Reg             0x3A
#define     TestADCReg              0x3B

//////////////////////////////////////////////////////////////////////
// MF522
//////////////////////////////////////////////////////////////////////
#define     TAG_OK                    0
#define     TAG_NOTAG                 (1)
#define     TAG_ERR                   (2)
#define     TAG_ERRCRC                (3)
#define     TAG_COLLISION             (4)
```

```c
typedef char tag_stat;

// A struct used for passing the UID of a PICC.
   typedef struct {
       uint8_t     size;           // Number of bytes in the UID. 4, 7 or 10.
       uint8_t     uidByte[10];
       uint8_t     sak;            // The SAK (Select acknowledge) byte returned from
the PICC after successful selection.
   } uid;

////////////////////////////////////////////////////////////////////////////////
// Function prototypes
////////////////////////////////////////////////////////////////////////////////

void rc522Init();
void antennaOn();
uint8_t readRegister(uint8_t reg);
void readRegisterMulti(uint8_t reg, uint8_t count, uint8_t *values);
void writeRegister(uint8_t reg, uint8_t value);
void writeRegisterMulti(uint8_t reg, uint8_t count, uint8_t *values);
void writeRegisterBitMask(uint8_t reg, uint8_t mask);
void clearRegisterBitMask(uint8_t reg, uint8_t mask);
uint8_t wakeUpTag();
void selectTag(uint8_t *uid);
void haltTag();
void calculateCRC(uint8_t *data, uint8_t length, uint8_t *result);


#endif
```

vending_machine_aslw.c

```
// card_reader_aslw.c
/*
Author: Ava Sherry
Email: asherry@hmc.edu
Date: 11/17/21
*/


#include "STM32F401RE_FLASH.h"
#include "STM32F401RE_RCC.h"
#include "STM32F401RE_GPIO.h"
#include "STM32F401RE_SPI.h"
#include "MIFARE_RC522.h"
#include <stdint.h> // for integer types (i.e., uint32_t)


void main(void) {
   // Configure flash and clock
   configureFlash();
   configureClock();            // Set system clock to 84 MHz

   // Configure SPI
   spiInit(16, 1, 1);

   // Configure MIFARE RC522
   rc522Init();

   while(1) {
       while(wakeUpTag() == 0x52);          // Wait until tag is present i.e FIFO
buffer changes
       uint8_t uid[4];
       selectTag(uid);
       haltTag();
   }



}
```