

Table of Contents

Table of Contents

- ▲ User Guide
 - 1.0 Introduction
 - ▲ 2.0 Setup and Operation
 - Assembly
 - RASPBERRY PI Packages
 - SQL Configuration
 - SmartSafe Systems Installation
 - Camera and RASPI 4
 - I2C Setup
 - SmartSafe Systems Setup
 - User Setup
 - Using the SmartSafe
 - Violations
 - ▲ 3.0 Specifications
 - Power Consumption
 - ▲ Technical Guide
 - 4.0 System Diagram
 - 5.0 System Description
 - 6.0 Circuit Description
 - ▲ 7.0 Software Description
 - AVR Embedded SmartSafe program
 - SmartSafe System
 - SmartSafe Webserver
 - StartUp
 - Async AVR Input
 - Webserver Application
 - 8.0 Troubleshooting
 - ▲ Appendix
 - A. Illustrations
 - C. Bill of Materials

User Guide

1.0 Introduction

My project proposal is for a **Two-Step Authentication Lock System**.

The idea for this project came as a recommendation by our teacher, and I improved on the functionality such that the finish product resembles an **ID validated security/surveillance system** found in certain areas where restricted access is permitted to a few registered individuals. I also wanted to challenge myself with the low-level languages we used during the program, such as Assembly and C, by using them on most of my project.

The system would allow for a low-cost implementation of these features and requires a System Administrator to perform its initial configurations, make future adjustments and to view its reports.

These configurations are not complicated and should allow both non-technical and technical users alike to set it up.

I chose to title this project SmartSafe Systems because it sounds like something the marketing department would be pleased with!

The system will consist of:

- **A magnetic solenoid** that is activated by a mechanism controlled via a microcontroller, in this case the ATMEGA8515;
- A user input system which is comprised of a **twelve(12) digit keypad**, which is also controlled by the ATMEGA8515, and a **capacitive touch fingerprint scanner**, controlled by an Arduino Nano;
- A validation system, being also the administrator interface, comprised of a **Linux desktop application & locally hosted Webserver**;
- A **16x2 character LCD Screen**, again controlled by the AVR microchip;
- A **5MP 1080p OV5647** Raspberry Pi Camera Module;
- A **Raspberry Pi 4b**;
- An **Arduino Nano 33 IoT**.

A Raspberry Pi 4 Model B will receive two types of inputs in the following order:

First, the user will be asked by the built in display to utilize the **capacitive touch fingerprint reader** to authenticate their identity.

Following that, the user will have to input a 6-digit code into a numerical keypad and wait for its confirmation. This code will serve as the user's ID number.

After this, the lock would be released, and would lock automatically back after a period of about three(3) seconds.

The system will be programmable through a web interface accessible from the LAN that requires root level access to enter.

By using the web interface, we can add users, register or delete fingerprints as well as change the 6-digit code.

We can also view system generated reports such as ID & fingerprint combination mismatches, and pictures taken by the surveillance system.

Through the use of a web camera, the system will take a picture of the person after a successful unlock, or after 3 unsuccessful ones.

This picture will be stored locally and can be viewed through the Web interface.

The system will require two power sources, one from a 12V AC/DC voltage regulator circuit, and another from the Raspberry Pi 4B's own power supply.

2.0 Setup and Operation

Assembly

The circuit is modular by design. The PCB will come with a small number of soldered components, and the rest will be connected through IC sockets, female and male pin connectors, and screw terminals.

Once fully assembled, the user can begin the process of configuration of the system.

RASPBERRY PI Packages

The following new packages need to be installed on the Raspberry Pi device you will use as the SmartSafe System's Server.

The below packages can be installed by performing the command:

```
#sudo apt-get install <pkgname> -Y
```

Where you would replace **pkgname** with each of the following:

- apache2
- libmariadb-dev and libmariadb-dev-compat
- mysql-common
- mariadb-server
- git
- wiringPi

Among all of the packages above, the only one requiring some configuration before utilization was Apache2.

Apache2 and Mariadb are services that run daemons, meaning they run on the background. Make sure you can detect their daemons by using the **service <service> status** command such as: **#service apache2 status**

IMAGE

If the output of this command did not indicate that the service is running, see [section 8.0 Troubleshooting](#) for further guidance.

To ensure the webserver is looking at the right directory structure, we must edit the

DocumentRoot field of the file located at:

nano /etc/apache2/sites-enabled/000-default.conf

The field is easy to find, being the first one inside the <VirtualHost> tag.

I changed it to the value of /home/gui/AVRserial/ss_server since this is where all of my webserver file structure resides.

It is important to know the order of default execution of scripts for Apache2. Make sure you have the correct landing page labeled **index.php**.

```
<VirtualHost *:80>
# The ServerName directive sets the request scheme, hostname and port that
# the server uses to identify itself. This is used when creating
# redirection URLs. In the context of virtual hosts, the ServerName
# specifies what hostname must appear in the request's Host: header to
# match this virtual host. For the default virtual host (this file) the
# value is not decisive as it is used as a last resort host regardless.
# However, you must set it for any further virtual host explicitly.
#ServerName www.example.com

ServerAdmin webmaster@localhost
DocumentRoot /home/gui/AVRserial/ss_server
#DocumentRoot /var/www/html
```

SQL Configuration

The SQL structure of this program is specific for its application. A dev. might be able to make use of a different one, but they would need to change parameters for variables present in both the application and webserver codes.

Assuming you want the intended operation of this system, follow these steps:

1. Download and install **libmariadb-dev**, **libmariadb-dev-compat**, **mariadb-server** and **mysql-common** as instructed above.
2. Ensure the service is running with **#service mariadb status**.

If the output of this command did not indicate that the service is running, see [section 8.0 Troubleshooting](#) for further guidance.

3. Log into the database using **#mysql -u <username>**

4. Enter the following commands exactly:

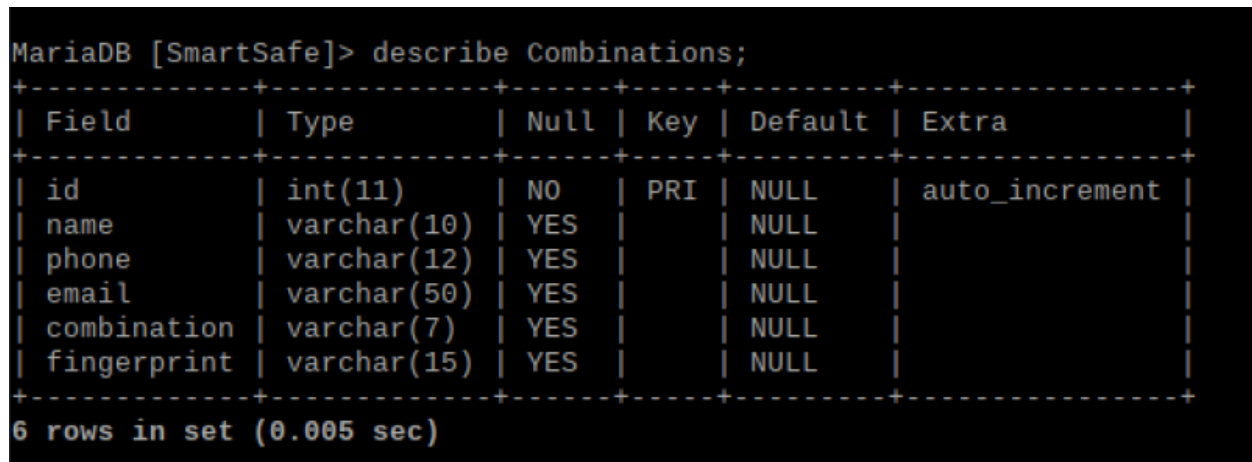
```
CREATE DATABASE SmartSafe;
```

```
CREATE TABLE Combinations (auto_increment primary_key id, varchar(10) name,  
varchar(12) phone, varchar(50) email, varchar(7) combination, varchar(15)  
fingerprint);
```

5. If done properly, you should be able to enter the command:

```
DESCRIBE Combinations;
```

And get the same result as the image below:



```
MariaDB [SmartSafe]> describe Combinations;
```

Field	Type	Null	Key	Default	Extra	
id	int(11)	NO	PRI	NULL	auto_increment	
name	varchar(10)	YES		NULL		
phone	varchar(12)	YES		NULL		
email	varchar(50)	YES		NULL		
combination	varchar(7)	YES		NULL		
fingerprint	varchar(15)	YES		NULL		

6 rows in set (0.005 sec)

Fig 4: SQL database format for appropriate SmartSystem operation

If you did not get the same table structure, review your commands, and fix any typos.

If the table description is exactly as the one above, we can move on.

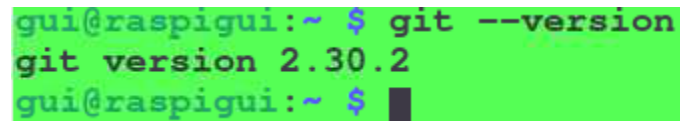
SmartSafe Systems Installation

The program application and webserver structures that I've created can be found in my Github account.

They are both freely available for anyone to download and install in their system.

Here are the steps to perform that installation:

1. Ensure that Git is correctly installed in your system.
 - a. Execute the command **#git --version**



```
gui@raspigui:~ $ git --version
git version 2.30.2
gui@raspigui:~ $
```

If the output of this command did not indicate the git version, see [section 8.0 Troubleshooting](#) for further guidance.

2. Go into your home directory and clone the git repository for the [WiringPi](#) module.

We will need that module to interface with the Arduino Nano board:

```
# cd
# git clone https://github.com/WiringPi/WiringPi.git
# cd wiringPi
# ./build
```

3. Now to download the SmartSafe Systems application:

```
# cd
# git clone https://github.com/PsychicPlant/computer_project.git
IMAGE
```

Camera and RASPI 4

The configuration of the Camera in the RASPI 4 device is simple.

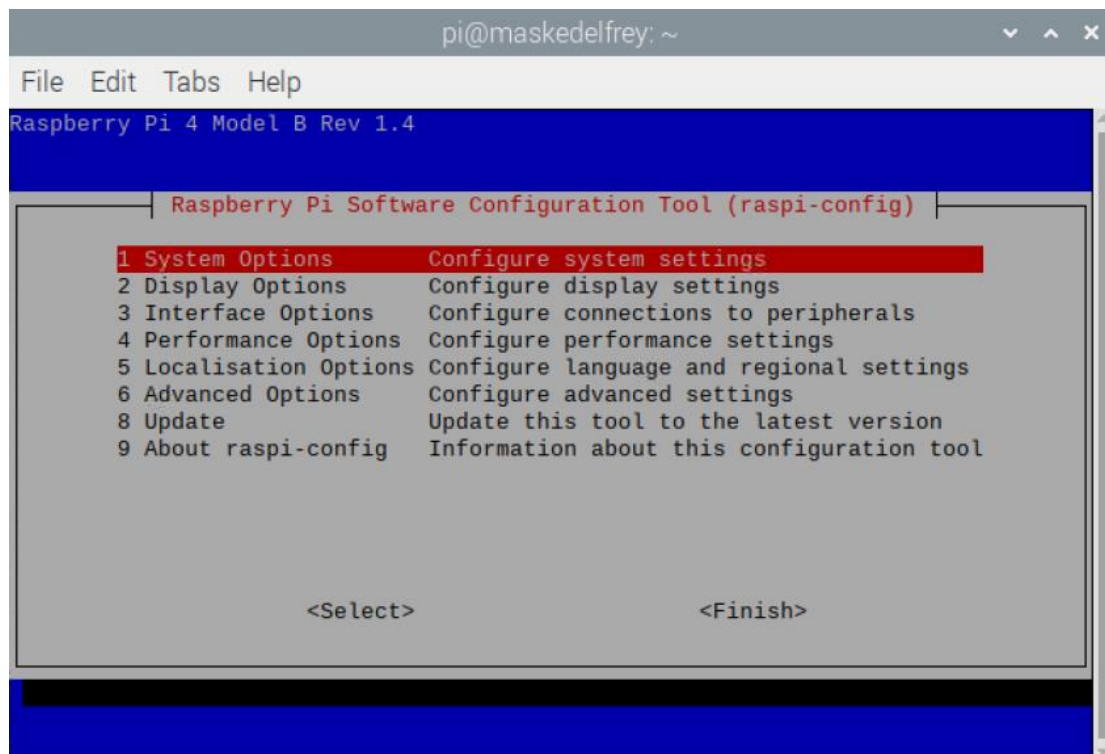
1. For the operation to be nominal, we must first update the whole Raspian OS by using the following command:

#sudo apt-get update && sudo apt-get upgrade -Y

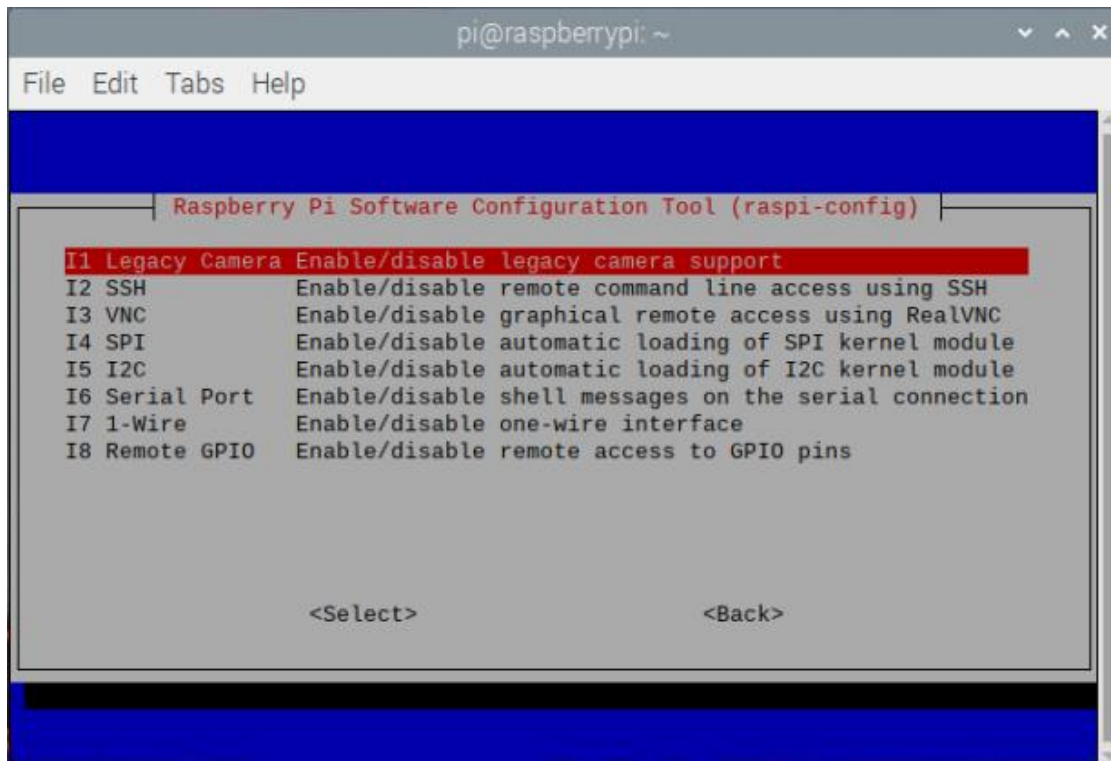
2. Then, to enable the interface the Camera will be using, we must enter the Raspian configuration interface:

#sudo raspi-config

This brings us to the following GUI:



3. Entering into the 3rd option of **Interface Options**, we see the following menu:



4. Then we choose the first option, the **Legacy Camera** and enable it. Upon rebooting the system, it is now configured to use the Camera.

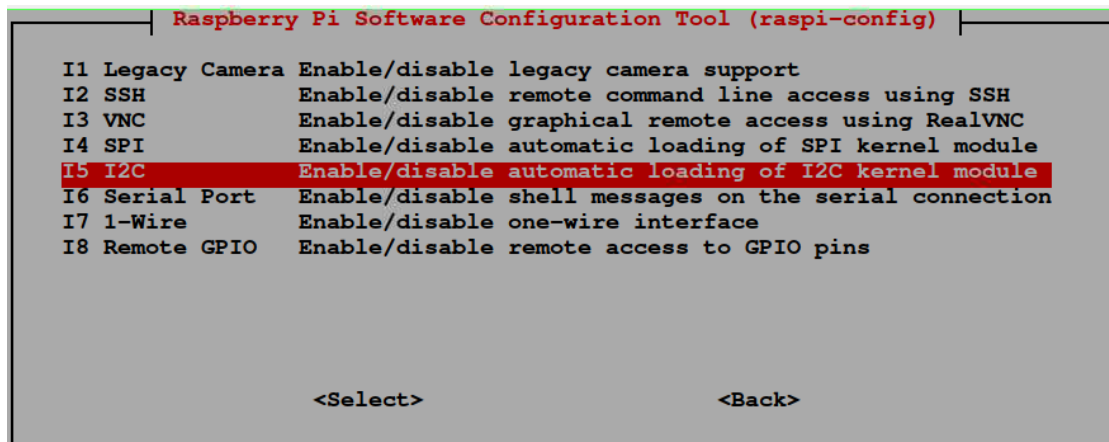
The OS comes pre-installed with the **raspistill** program, which we will use to capture still images through a script later on.

I2C Setup

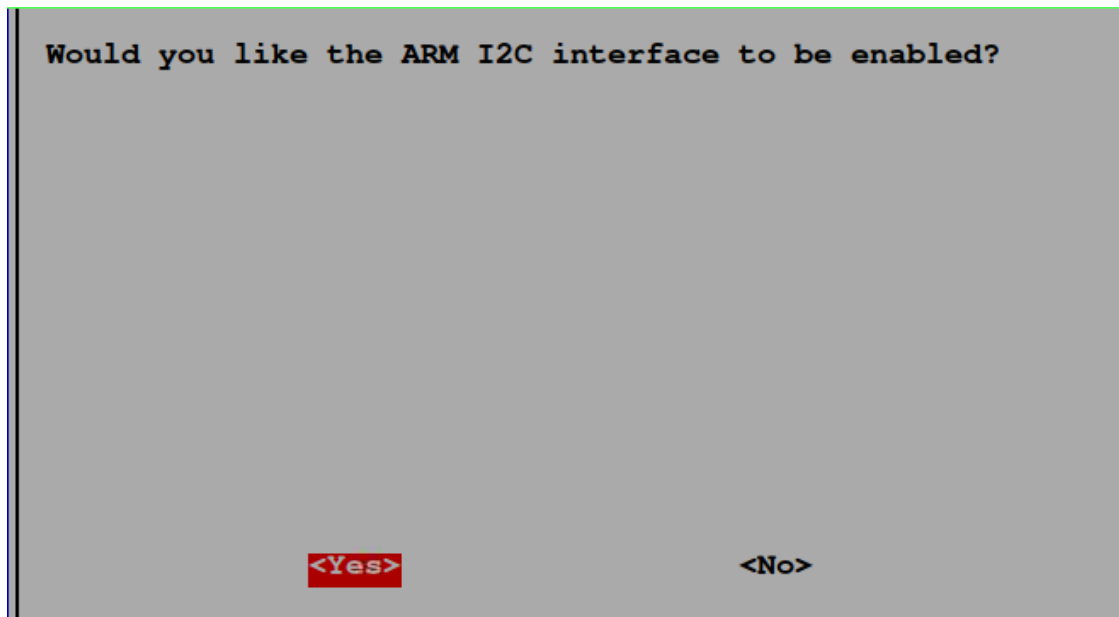
Now, similar to the camera setup, we need to enable a special type of interface on the Raspberry Pi to be able to use the system.

This interface is known as the I2C Bus, and it will allow us to asynchronously communicate with the Arduino board.

1. Again, type the command **#sudo raspi-config**
2. Again, go into **Interface Options**
3. This time, pick the option 5 **I2C**



4. Press the **Yes** button



The system may ask you to reboot.

If it does, obey!

SmartSafe Systems Setup

After all of the above steps have been taken, the real application should be able to run without any complaints.

The below steps are required to setup the system, but after that they will become optional and further configurations will only be necessary when adding/updating/removing a user to the system.

1. Power up the hardware. This is done by simply connecting the 2.5mm connector provided to the barrel receptacle located at the power module.

There is really only one way to insert it, and only one connector that fits it, so it should be evident as to where it is.

The specifications for the voltages will be found in [section 3.0 Specifications](#).

The powering up sequence will make the LCD display turn on and display its banner message.

IMAGE

Wait until the system has booted up and entered its default state, as shown below:

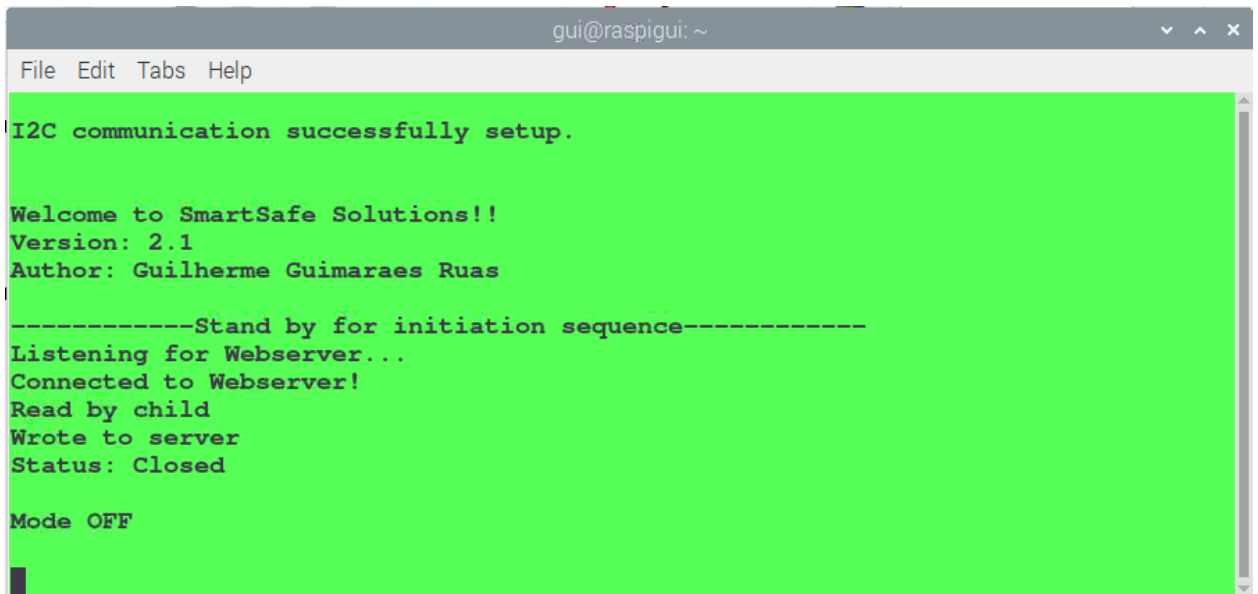
IMAGE

2. Start up the SmartSafe Systems application on the Raspberry Pi device.

Go into the installation folder of the application and run the executable called [avrserial.exe](#):

```
# ./avrserial
```

After executing this command, you will be met with the following interface:



```
gui@raspigui: ~
File Edit Tabs Help

I2C communication successfully setup.

Welcome to SmartSafe Solutions!!
Version: 2.1
Author: Guilherme Guimaraes Ruas

-----Stand by for initiation sequence-----
Listening for Webserver...
Connected to Webserver!
Read by child
Wrote to server
Status: Closed

Mode OFF
```

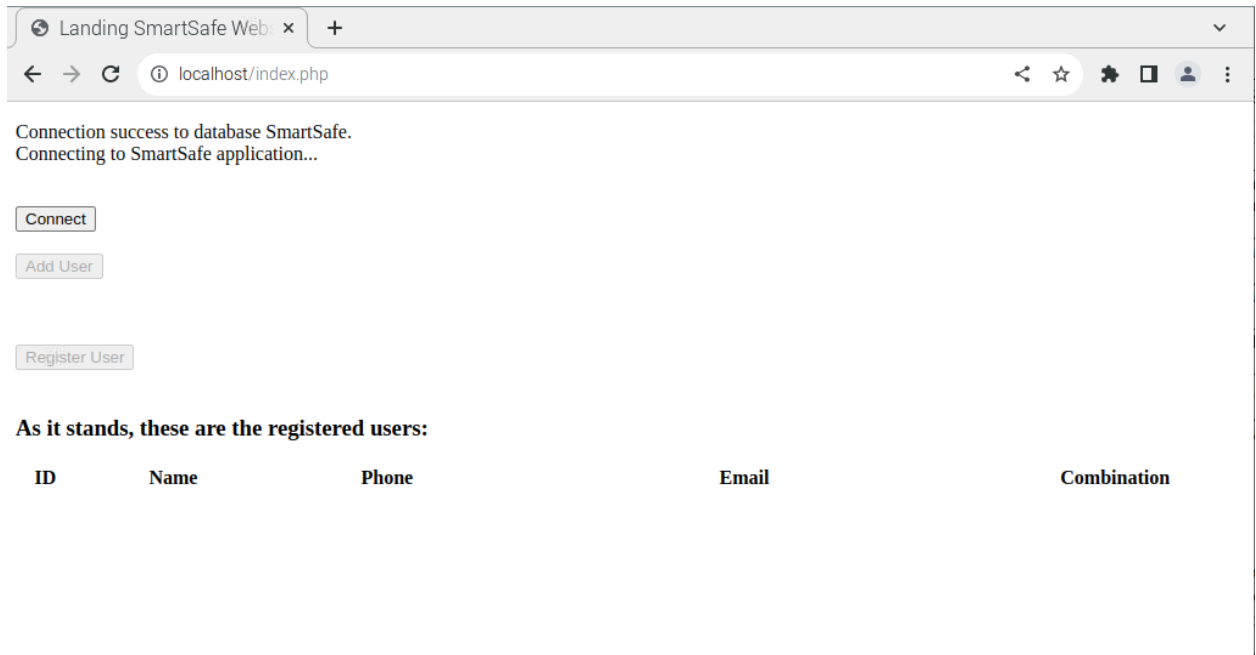
The output you see above is what you will see on your screen given the correct installation of the system via the previous instructions.

This screen will present you with some diagnostic messages in case something goes wrong.

The details will be explained further in the Technical Guide, but for now if you see this output, you can move on to the next step.

3. Open your web browser and navigate to **localhost** by typing <http://localhost> on the URL field.

You should be presented with the following landing page:



If you see this landing page, then the system setup was successful and we can move on to the last step on the configuration, which is configuring the users to be validated.

User Setup

If you made it this far, it means you have successfully configured the rest of the system.

Now, without leaving the landing page on the web browser, we will set up the users.

1. Click on the Connect button, which is the only available button to begin with.
You should see the web page change to this:

Connection success to database SmartSafe.
Connecting to SmartSafe application...

FIFO write pipe /home/gui/AVRserial/pipe1 opened.
Opening read pipe.

FIFO read pipe /home/gui/AVRserial/pipe2 opened.
Waiting for application response...

OK!

As it stands, these are the registered users:

ID	Name	Phone	Email	Combination
----	------	-------	-------	-------------

If you did not see this output, and instead saw a message about connection timing out, see [section 8.0 Troubleshooting](#) for further guidance.

2. Click on Add User.
You should be redirected to the following page:

SmartSafe New User x +

localhost/adduser.php

Add SmartSafe User

Use the below form to add a new user to the SmartSafe database.

Name [Maximum of 10 characters]

Contact Information

Phone Number: Email Address(Optional):

Fill in the information for the user in the appropriate fields.

Warning: The username should not exceed 10 characters long as instructed.

3. After adding the users, you will be directed again to the landing page. Click connect once more and you should see a drop-down menu with the name(s) of the user(s) you just added.



Connection success to database SmartSafe.
Connecting to SmartSafe application...

FIFO write pipe /home/gui/AVRserial/pipe1 opened.
Opening read pipe.

FIFO read pipe /home/gui/AVRserial/pipe2 opened.
Waiting for application response...

OK!

As it stands, these are the registered users:

ID	Name	Phone	Email	Combination
1	Guilherme	5145984301	mmorpg124@hotmail.com	
2	Mononoke	5656565656	kitty@cat.cute	
3	Mamae	123321546	teamo@prasempre.beijos	

Pick a user and press on the Register button.

You should see the text change into the word LOADING:

LOADING

As it stands, these are the registered users:

And you should notice the LCD changing to display a message Registering User, Input Combination.

4. The system will then go into input mode automatically:

You can now enter the combination for this user.

As previously stated, the combination is a 6 digit numerical code that you will use the system's **keypad** to input asynchronously.

After pressing the 6th digit, the combination will be sent to the application and you will receive a registration successful message on the LCD.

Back on the webserver, you can confirm the combination by seeing it being displayed on the webserver, where the LOADING text used to be.

Register User

111111

As it stands, these are the registered users:

5. By refreshing the page, you can now see that the **Combination** field of the table on the bottom was filled in with your new combination!

If you made a mistake, simply select the user, and perform the registration again.

If you do not see the user's combination on the table, see [section 8.0 Troubleshooting](#) for further guidance.

6. Now, press on the Scan User button.

Again, you will see a loading text appear and instructions to move to the **fingerprint scanner**.

Use your finger of choice and register it. Pay attention to the webserver as it will indicate success or failure.

Repeat this process until you have successfully registered your fingerprint to your username.

Using the SmartSafe

Now, we are finally set up to utilize the product.

The system is meant to stay on at all times, providing asynchronous access to the registered users. As such, simply move away from the computer and observe the LCD screen.

There are two steps to unlocking the mechanism:

1. First, place the registered finger on the sensor.

The LCD display should tell you if it detects that the fingerprint exists in its database or not.

For safety reasons, it will not mention your username and assumes you know it.

2. Next, the LCD will indicate that the system has gone into user input mode. Now simply type in your 6 digit combination and, given the correct combination of code and fingerprint, the solenoid holding the door will be retracted.

The solenoid will be released back after about 3 seconds, so locking the door is as simple as pushing it back into place.

Violations

The system was designed to be used on multiple doors at the same time. For that reason, it contains security features that will warn the owner/manager of the system about any suspicious activity.

Things that could trigger a violation are:

1. Putting an inexistant combination.
2. Incorrectly putting a combination for a given fingerprint.
3. Incorrectly scanning a fingerprint for a given combination.

There are two types of violations possible:

1. Warning: This happens when you perform one of the above-mentioned actions. This will be written to a report viewable on the system's webpage.

2. Report: This happens when you perform one or more of the above-mentioned actions 3 consecutive times.

- 3.

This causes the surveillance system to activate the **camera**.

The camera will take a still frame of the moment the violation occurs and it will be stored locally.

These images can be viewed by the administrator on the web interface.

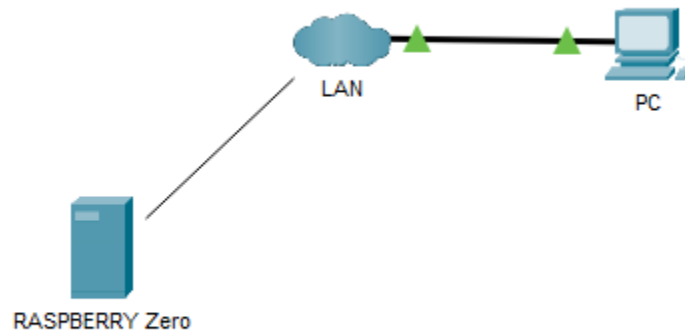
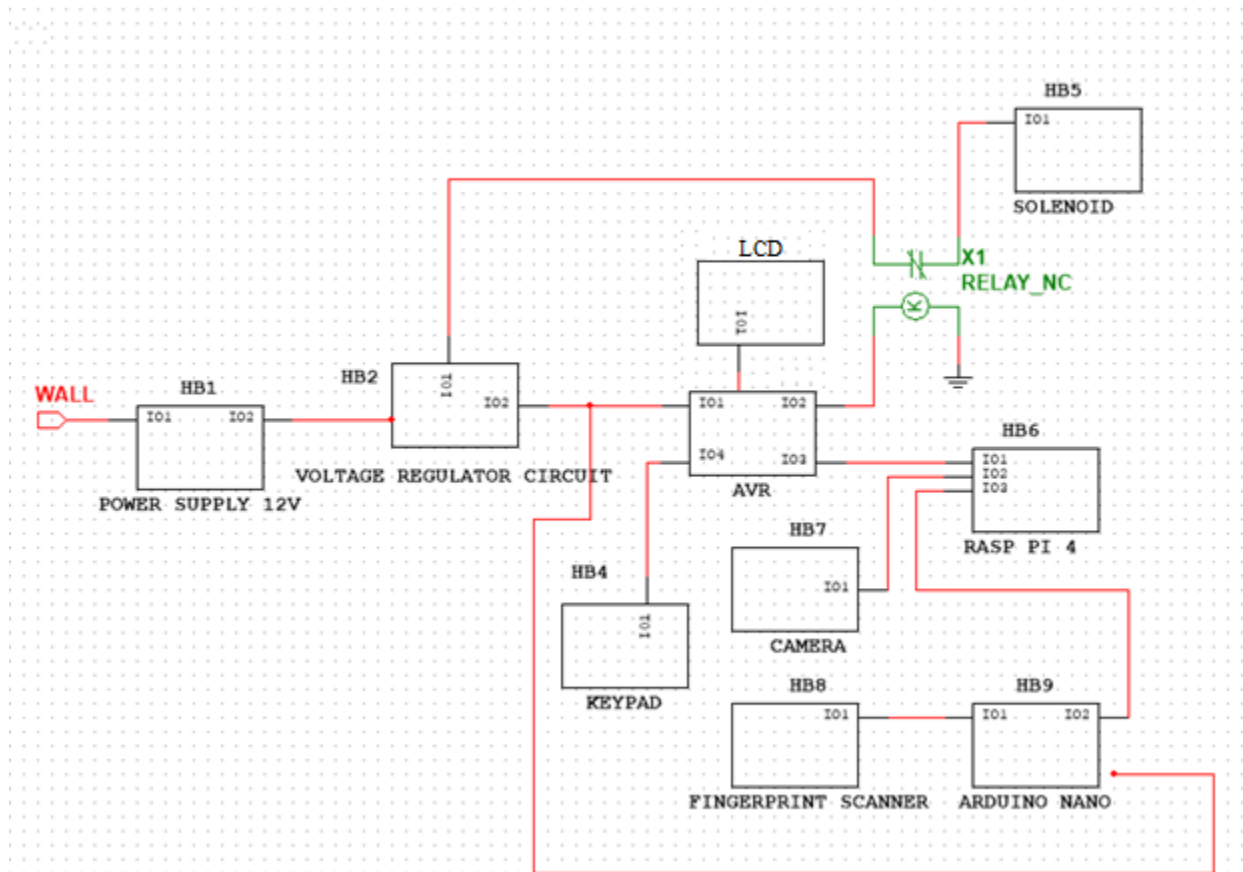
3.0 Specifications

Power Consumption

Device	Voltage Consumption	Current Consumption (max estimate)
AVR	5V	Negligible
Arduino Nano 33	5V	100mA
Solenoid	12V	400mA
Finger Scanner	5V	120mA

Technical Guide

4.0 System Diagram



5.0 System Description

The **Power Supply** module takes care of delivering power to all components in the system, except the Raspberry Pi 4B. This module is composed of the ACM18US12 Wall Mount Adapter.

It's installed together with the Voltage Regulator circuit to provide two voltage levels: 5VDC and 12VDC.

The **Voltage Regulator** circuit serves to split the 12VDC into 12V and 5V.

Alone, it does nothing, so it connects with the main board via wires attached to screw terminals on each board.

The **Keypad** and **AVR** blocks work closely together. The AVR block consists of the ATMEGA8515 programmable microcontroller and that is because it is low power, easy to use and already present in my inventory.

The keypad is a 3 column 4 row switch pad which shorts GPIO pins on the AVR's PORTB interface to create a low-active description of a pressed key.

The **LCD** block is composed of a 16x2 character LCD screen. It has a high refresh rate and low power consumption.

The LCD is controlled by the AVR by means of hard-coded character sequences which are sent to the LCD given user input/state update.

The **Solenoid** part of the circuit is, as the name implies, a magnetic solenoid that will be manipulated through the microprocessor in response to some actions.

These actions are either entering the correct combination of keystrokes and fingerprint scan, or a manual activation through the SmartSafe application accessible by the system's admin.

It is controlled by the AVR.

The **Rasp Pi 4** block is composed of the Raspberry Pi 4 Model B microcomputer.

I chose this device because it had already been purchased for past semesters.

This device will be responsible for hosting the web server, for hosting the SQL database for the programmable 6-digit codes and their respective finger print scans, and for using the Camera to take pictures and storing them locally.

More importantly, it performs validation of all user input and allows for the intercommunication of various devices and applications without further controller circuitry.

The **Camera** block is composed of a OV5647 Sensor Module camera specifically built for Raspberry Pis.

These cameras are low powered and take good pictures. On top of that, they are extremely simple to configure and use because of readily available Linux packages such as **libcamera-still**.

The **Arduino Nano** block is composed of an Arduino Nano 33 IoT microcontroller.

This device will interface with the fingerprint scanner to perform asynchronous scans and deliver the results to the Raspberry Pi for validation by means of the I2C bus.

This device will be powered by the wall adapter.

The **Fingerprint Scanner** block contains a capacitive fingerprint scanner. These scanners come with lengthy but comprehensive user manuals. The devices also come with some flash memory that stores finger captures.

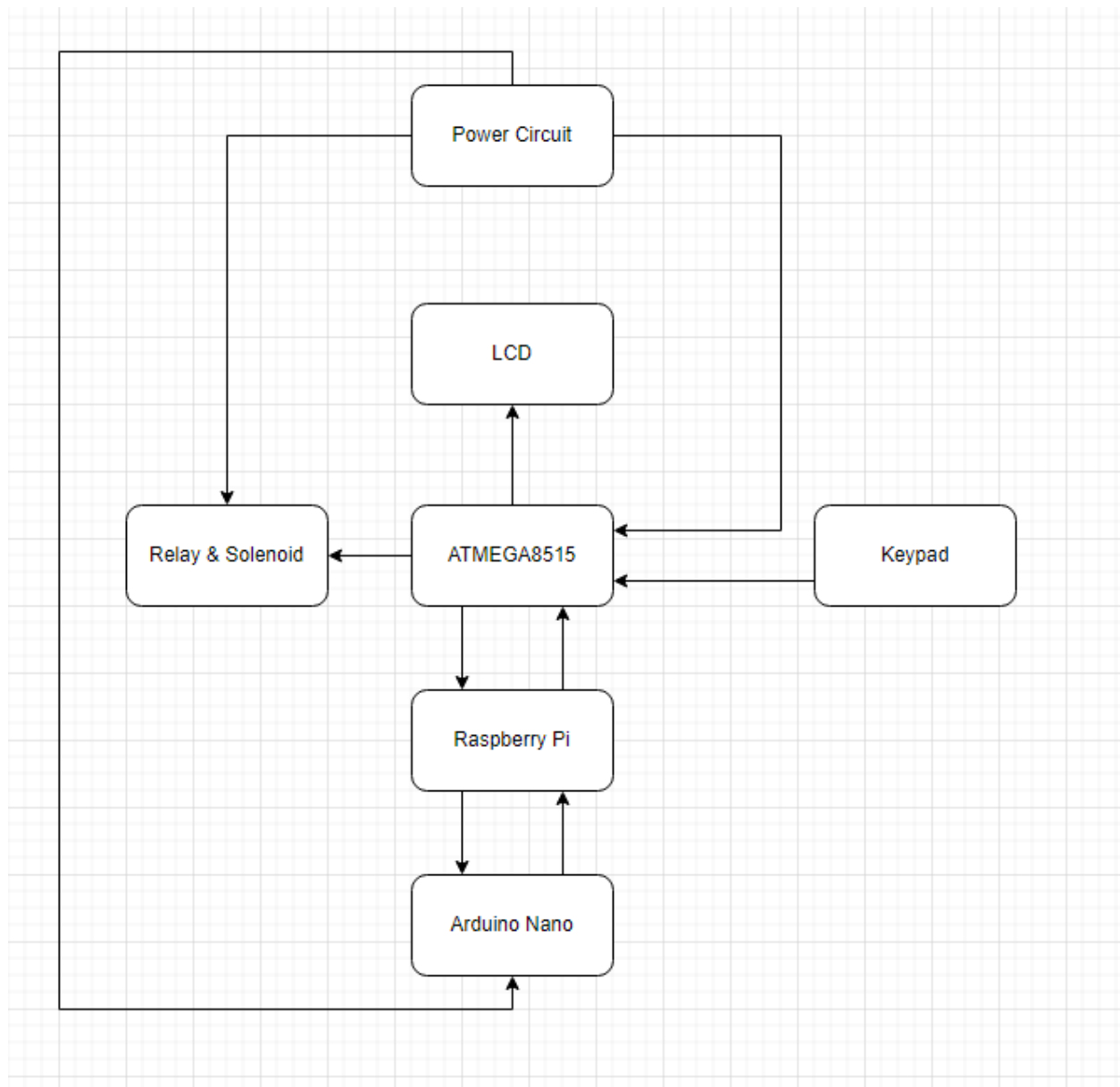
The capacitive scanner used in this project can store up to 500 scans, but my system can only handle up to 127 users.

The **Web Interface** will be accessed by the system administrator.

This interface will be hosted by the RaspBerry Pi.

The interface will make it easy to change the keypad code, store and delete fingerprint images, view logged events and show image captures from violations

6.0 Circuit Description



Above we can observe a general block diagram of the circuit.

The following section will explain the connection between them in a format that goes as follows:

“Block title 1” -> “Block title 2”

The arrow indicates the direction of data and/or signals.

ATMEGA8515 -> LCD:

The connection between these devices is done via the copper tracks on the circuit's top and bottom faces.

Originally, the LCD was planned to be controlled by a multiplexer circuit, but due to shortage of time, I had to move it to the regular AVR parallel Address:Data bus.

The bus created by the AVR takes ports A and C; port D cannot be used because it contains the special pins for UART and INT0, so I am using PORTB.

PORTB contains the SPI pins, but having the keypad wired to them is not a problem as long as only the SPI can send a prolonged !RESET signal. This indicates Serial Programming mode to the AVR.

The orientation of the pins on PORTB is also a mirror of PORTC, so the planned Keypad connection also had to be mirrored.

This only means that I must keep in mind the orientation of the Keypad itself. The keypad will be connected with the keys facing the AVR.

The LCD will be Memory mapped to **addresses \$2000 to \$2FFF** and **repeats every two increments of the MS byte**.

The AVR can configure the LCD by writing to **address \$2000 and write to its RAM via address \$2100**.

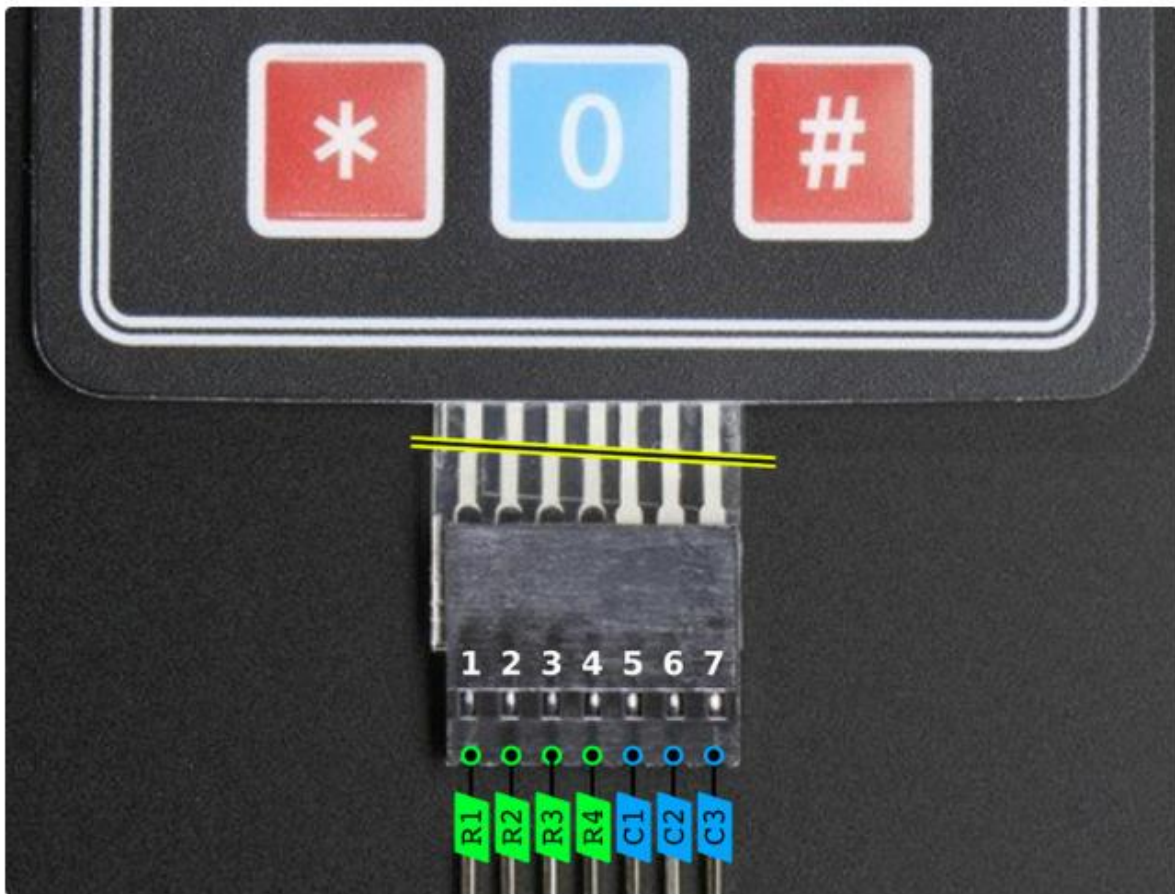
Pin 14 of the AVR is not used for anything but needed to be grounded to allow the ground plane to reach the underside of the IC.

Don't worry! That pin is only important when either using PORTD, or when running UART in synchronous mode, neither of which we are doing.

Keypad -> ATMEGA8515:

④ Membrane 3x4 Matrix Keypad (PID 419)

Pinout



The keypad is connected to the AVR microcontroller via an 8-pin female header present in the board.

The system utilizes a state driven architecture, where a user must press the # key before being able to enter any input.

This is done to avoid accidental input to the Raspberry Pi and to allow me to block user input from the keypad while data is being processed.

The system uses the AVR's internal pull up resistors to create an **active-low** input bias, where a **column pin is driven LOW** and the program reads the PORTB to determine which row pin was pulled low, if any at all.

To that end, I designed the circuit with pins 1 and 4 of the keypad branching out on the copper board towards a **74LS02N NOR gate chip**.

These two pins serve as the input to a NOR gate, which when driven low simultaneously, will create a high pulse output.

That output is connected to **the AVR's INT0** via copper tracks.

To interpret the inputs from the keypad, I had to calculate the expected PORTB value for each key. These values can be seen below:

- Column 1 = X111 1011
- Column 2 = X111 1101
- Column 3 = X111 1110

Digit	Binary	Hex
SCAN C1	1111 1011	\$FB
SCAN C2	1111 1101	\$FD
SCAN C3	1111 1110	\$FE
1	1011 1011	\$BB
2	1011 1101	\$BD
3	1011 1110	\$BE
4	1101 1011	\$DB
5	1101 1101	\$DD
6	1101 1110	\$DE
7	1110 1011	\$EB
8	1110 1101	\$ED
9	1110 1110	\$EE
*	1111 0011	\$F3
0	1111 0101	\$F5
#	1111 0110	\$F6

These inputs would all be read in parallel through the AVR's PORTC, but the scanning was programmed in such a way that only one column can be read at a time.

The column pins should be output pins, and the row ones should be inputs.

Thus, the DDR code for PORTC(DDRC) will be \$07.

ATMEGA8515 -> Relay & Solenoid:

The connection between the AVR and the Relay is done via copper tracks.

The tracks go through a small switching transistor before entering the relay, which then allows current through the Solenoid.

Power Circuit -> Everything powered by rail

The voltage regulator circuit will have to create a 12VDC and a 5VDC outputs to be able to power both the Solenoid and the AVR.

The power supply I purchased is a 12V 1.5A AC to DC device:

Output Power	Output Voltage	Output Current	Total Regulation ⁽¹⁾	Efficiency ⁽²⁾	Model Number ^(1,2,3)
12.5 W	5.0 V	2500 mA	5%	82.5%	ACM18US05
18.0 W	9.0 V	2000 mA	5%	86.7%	ACM18US09
	12.0 V	1500 mA	5%	87.5%	ACM18US12
	15.0 V	1250 mA	5%	87.5%	ACM18US15

Fig 1: Output Power of the 12V power supply

Input	Voltage	Sleep	BareMinimum	Blink	Shake detector (IMU)	Basic Scan Networks (Wifi)	HTTPS GET (Wifi)	BLE usage
USB	5V	OK	OK	OK	OK	OK	OK	OK
3.3V pin	3.3V	6mA	18mA	22mA	19mA	112mA	110mA	47mA
Vin	4.5V	10mA	18mA	21mA	19mA	88mA	92mA	41mA
Vin	5V	16mA	25mA	26mA	25mA	82mA	84mA	39mA
Vin	6V	13mA	20mA	21mA	20mA	72mA	66mA	37mA
Vin	9V	<1mA	5mA	7mA	6mA	50mA	40mA	20mA
Vin	12V	<1mA	4mA	5mA	4mA	42mA	37mA	16mA
Vin	18V	<1mA	1mA	2mA	1mA	28mA	30mA	10mA

Fig 2: Arduino Nano 33 IoT power consumption chart

SPECIFICATION

- Operating Voltage: 12V
- Operating Current: 0.4A
- Electrify Time: unlimited
- Stroke: 10mm

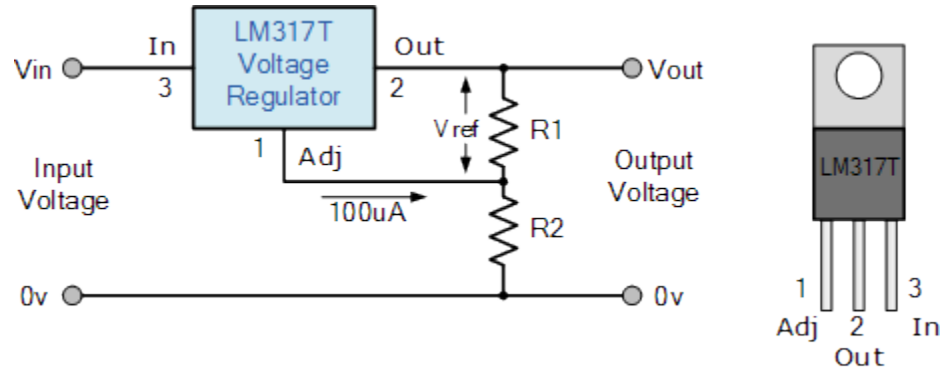
Fig 3: Solenoid power consumption characteristics

- Supply voltage: 3.6 - 6.0VDC
- Operating current: 120mA max

Fig 4: Fingerprint Scanner power consumption characteristics

The AVR's current consumption is really negligible for my application.

I needed a **power regulator** circuit. Thankfully, this is simple to implement, and all of the components needed to produce it are already available to me. Here is a simple schematic of it:



V_O is calculated as shown in Equation 1. I_{ADJ} is typically $50\mu A$ and negligible in most applications.

$$V_O = V_{REF} (1 + R_2 / R_1) + (I_{ADJ} \times R_2)$$

But $(I_{ADJ} \times R_2) = (0 \times R_2) = 0$, thus:

$$V_o = 5V$$

$$V_{ref} = 1.25$$

$$I_{adj} = N/A$$

$$R_1 = 1K$$

$$5 = 1.25 (1 + R_2/1000)$$

$$5 = 1.25 + 1.25(R_2)/1000$$

$$3.75 = R_2(1.25/1000)$$

$$R_2 = 3.75/(1.25/1000)$$

$$R_2 = 3000\Omega$$

Raspberry PI -> AVR:

The connection of these two components is done via copper tracks.

The signals from the Raspberry Pi reach the board via a 10pin male on-board header.

The AVR is a device that uses logic levels ranging from 0V to 5V.

However, the Raspberry Pi uses logic levels from 0V to 3.3V.

To accomplish successful communication between them, I designed a 12pin female header that has a **Logic Level Converter** installed on it.

This LLC serves as the bridge to the Raspberry Pi's TX and RX, and their respective counterparts on the AVR.

Arduino -> Raspberry PI:

The connection of these two devices is done via copper tracks.

Both of these devices operate at the same logic levels, so I designed the circuit with tracks going straight from the Arduino IC socket to the Raspberry Pi's on-board connector.

The communication between them is done via the I2C bus, so they are only connected by two pins.

7.0 Software Description

Let us start by the AVR program.

AVR Embedded SmartSafe program

The embedded program inside the AVR, as mentioned previously, is capable of running on its own. It will not crash in the absence of the SmartSafe application running on its serial port, nor will it cause the unintended operation of the solenoid device on PORTE.

This program was completely written (with 3 or 4 subroutines from Mr. Nick Markou's Embedded Systems Hardware program) by me using standard libraries in Assembly.

It is comprised of the files:

- Main.asm
- Rxmode.inc
- Txmode.inc
- Termio.inc

The program running on the AVR must be of an asynchronous nature.

Not only should it be capable of receiving input from the Keypad device at any point in time, but it should also be listening on its serial port for any incoming command sent from the SmartSafe program.

It should also be able to asynchronously reply to the program in the serial port.

Here is a comprehensive flow of how the program goes:

1. Initialization sequence: Takes care of setting up serial comm. values, initializing variables used in the program, and sending out a banner message.
2. **Main Loop:** The program defaults on a global variable called FLAG0 to the value of 0x00. The program enters in this small loop after it is set to constantly monitor the 3rd column of the keypad for the **# key** input.

This is because the program is in **STANDBY** mode, represented by the FLAG0 state 0x00. In this mode, the AVR will ignore all input (with the exception of the # key). The only way to exit this mode is to press the # key, which takes the user into **INPUT** mode, represented by FLAG0 state 0x01.

This flag is controlled either by commands sent by the SmartSafe program, or by an interrupt sbr called **isr0**, which is activated upon pressing the # key.

Depending on the flag state, the main loop will either allow the program to scan the keypad for input, or it will loop itself back to the top.

Independent of the mode in which the user is, this loop is always called since it contains the **receive_check** subroutine.

3. **Receive_check**: This part of the program polls the UCSRA for any received input in its buffer. This is a means to asynchronously listen for the SmartSafe program on the serial port. It will also perform all of the necessary operations to execute the commands it receives, such as Open, Close, Status, and Sync.
4. **Keypad**: If by the end of the main loop, the system detects that it is in INPUT mode, it will then jump to the **keypad** section.

With the use of calculations previously done, I determine the necessary values to output to and read from the PORTC interface such that the program individually scans the columns in search of a shorted pin.

If it finds a shorted pin, it jumps to the **keypad_in** section.
If it fails to find a shorted pin, it loops back to main loop.

5. **Keypad_in**: Uses the subroutine **txsendinput** to send the key pressed thru the serial port, encapsulated by the hex value 0xF1. This encapsulation technique was used everywhere in my project as a means of checking the integrity of a signal, and differing between debug messages, input or commands.
6. **Isr0**: This subroutine is called by pressing the # key on the keypad. It performs the switch from INPUT and STANDBY mode. It also sends status messages indicating what mode it is on to the SmartSafe application.

SmartSafe System

Now that we have looked in a broad scope what the embedded SmartSafe program does, let us look at the Linux application SmartSafe in an even broader scope, since this program is quite long and complex.

This program was also written entirely by me, using only standard C libraries with the exception for the mariadb/mysql.h libraries. This will be considered in the manual for the program dependencies.

This program is very complicated, and so instead of explaining it piecemeal like I did with the embedded version, I will explain it in broader terms, and will show examples later in the sections 6 and 7.

It is comprised by the files:

- Avrserial.c
- Avrbehaviour.h
- Childbehaviour.h
- Combinations.txt

INITIALIZATION (PARENT)

The program starts by defining a couple global and local variables. It then proceeds to check if the system contains three vital files: **pipe1**, **pipe2** and **combinations**.

The two first files are called named pipes and are known as FIFOs. The last one is a simple text file. The system will check for the existence of all those files and create them if they did not exist.

The system proceeds to creating regular pipes before calling a fork() operation.

I decided to use this capability to create a child process that handles interactions between the SmartSafe application and the other devices such as the SmartSafe webserver and the peripherals.

This design adds a lot of complexity to the program, but it also makes it more modular on top of reducing the failure domain of a problem in any of the given devices, allowing them to crash or be interrupted without corrupting the main program.

In safety terms, this means that the safe could still be operated if something went wrong with the other parts of the application. So long as the AVR/Keypad and the main SmartSafe app are running, the safe can be closed and open (opening requires Admin access to the CLI application).

The child will inherit a couple necessary variables, but will only communicate to the parent process via the pipes created. Each process is responsible for closing the appropriate sides of their pipes so we have in total 3 half-duplex connections between them.

TERMINAL

In the main process, the program will start the operations to create its own terminal, such that all user input (stdin) CLI input is redirected towards it, and all program output is redirected towards the Linux line terminal (stdout).

This terminal is very important because it is the thing that allows asynchronous communication between the **RASPI** and the **AVR**. The terminal is manually configured with the agreed upon parameters for communication such as parity, speed, stop bits, etc.

The terminal is created by means of **tcsetattr()**.

A reference to the default terminal attributes is kept for the exit sequence.

POLLING

One great method I found to make an asynchronous application that receives and sends out information to different devices is to use the **poll.h** library. It is standard POSIX.

To make a long story short, polling creates a struct object that holds pointers to open files and polls them for special events defined by the user.

For my purposes, and to reduce complexity, I am only using the POLLIN event so far, which happens when a file contains buffered data and is ready to be read from.

The main process and the child process both perform polling of various files.

The main process polls:

- The AVR
- The CLI (stdin)
- The child process (data)

The child process polls:

- The parent (data)
- The parent (cmd)
- The Webserver

There may need to be extra polling added for the Fingerprint Scanner, but this is what's defined as of yet.

This polling is performed on a loop on both processes, and they can be optimized by setting the time, in milliseconds, before another poll of the files is done. Each poll() call is done in parallel in all observed devices, so async operations can happen very fast.

The polling process creates a decision structure that allows the program to execute different pieces of code depending on which file was detected.

PROCESSING INPUT (PARENT)

The polling in the parent happens for three different input sources. They are described as:

- AVR

When input is detected from the AVR, the program will send it for processing.

The processing function will determine if the input sent was a command or a regular input.

For regular inputs from the keypad, it will send this raw data to a lookup table that returns the assigned real value of that hex code received (see Table 0 from modulereport_1).

This operation is only done when the AVR is already in INPUT mode, so the program will interpret this input as the beginning of a request to evaluate a 6-digit combination.

The program evaluates the characters received and stores them in a file (in my case, it is called **combinations**, but this can be configured by changing the CMBTXT macro in the program code). This file exists for logging purposes.

The input processing function returns the combination acquired to the main program, which evaluates it against an SQL database, which must exist in the machine running the program and is configured in the avrbehaviour.h file.

Furthermore, the AVR processing is also responsible for inputting new combinations into the database, and its behavior is dependent on a global flag named **STATE**.

This flag changes the interpretation of the result from the AVR input processing procedure. This makes it easy for future development of the application and different input interpretations to be added.

For debugging messages sent from the AVR, the program recognizes it and immediately sends the message to **stdout** to be displayed.

- STDIN

When input comes from the user by means of CLI thru **stdin**, the program checks for its available command set.

If it does not find a corresponding command to the one typed, it will not do anything (or possibly ask for the user to retry).

The functions available to the CLI control directly the AVR or the application itself.

One of the main functions is the “q” command, which will ensure proper termination of the process by closing all open files, freeing allocated memory, and flushing any data hung on pipes. This is crucial, since the creation of a child process can lead to a **zombie** if not properly terminated with its parent process.

A comprehensive list of all commands will be available below in section 5.

- CHILD

When input comes from the child, the program evaluates it for either processing, or straight outputting the string.

This is a necessary step for the conversation between the main SmartSafe application and the SmartSafe webserver since they do not share a common pipe and can only communicate through the child. This is also a security feature.

One of the main jobs of this loop is to check for the request to register a new user to the database.

So far, this loop can successfully add/update a user’s combination and store it on the SQL database.

INITIALIZATION (CHILD)

The child program code is stored in the childbehaviour.h file. As part of its initialization, it closes the appropriate pipes to create half-duplex connectivity to the SmartSafe. What it also does it opens the named pipes mentioned above, one for reading and one for writing.

These pipes will be used to communicate directly with an open SmartSafe Webserver application. As a security feature, the child will not start unless it is able to connect to the server, or if it detects the server exists.

To this extent, the parent process automatically starts a PHP script which will connect to the child on startup. This PHP script is in a pre-defined location, and if it doesn't exist, it means the SmartSafe Webserver doesn't exist, and thus it cannot perform its functions.

It will timeout eventually if it does not send a status update to the parent, which sends a kill signal to the child.

PROCESSING INPUT (PARENT)

The polling in the child happens for three different input sources. They are described as:

- PARENT CMD

This input pipe is dedicated for the parent to send commands to the child. These commands can be expanded on, but as of right now they are "exit", "reg_ok" and "reg_err".

The exit command is sent when the parent has been given the quit command. This ensures the child performs its own closing of files, freeing of memory, etc.

The reg_ok and reg_err are status and result messages sent to the child to indicate the success or the failure of registering/updating a user.

- PARENT DATA

The eventual goal of this detection is to be able to send large portions of data from the parent to the child in bursts. This operation would be enabled by a command given by the child.

- WEBSERVER

This pipe can only be written to by the Webserver, and is necessary to perform the interfacing from web application to terminal application of the SmartSafe System.

As of now, it contains two operations: register/update user and check connectivity.

SmartSafe Webserver

The SmartSafe Webserver is a server running on Apache2 which is comprised of the following files:

- Index.php
- Adduser.php
- Inseruser.php
- Registeruser.php
- Connect_server.php

It is simpler to explain this file structure using a flow description, illustrated by a flow chart below. We start at **index.php**.

This is the landing page. The user can connect to this page by performing the correct Apache2 configurations (shown below in section 5) and going to **localhost** on the browser search bar.

This webpage can only be accessed locally and is meant only for the SmartSafe System Admin to operate.

The first operation of this page is to verify that the configured SQL database exists, and it can successfully connect to it.

It then plots the database onto a table such that the Admin can see all of the users, their credentials, their combinations and (eventually) the identifier to their fingerprint scan.

There are various options on how to manipulate the system from here, but first the Admin is required to press on a button named "Connect". The other input fields are disabled until this operation is completed. This is for security and to guarantee that the SmartSystem's child process is still alive and listening to the webserver.

The “Connect” button will begin the **connect_server.php** file, which will listen for the SmartSystem’s response. If successful, it will display a message indicating the successful connection, and it will enable all of the other functions of the page. If it cannot connect for 5 seconds, it will time out, and require the Admin to reload the page and try again.

This script is run async thru AJAX.

The “Add User” button sends a GET request to the **adduser.php** script, which creates a webpage containing a single form which requires the Admin to input credentials to identify the new user.

Upon clicking the “Submit” button, the **insertuser.php** script is called, which as you may have guessed, inserts the new user and its credentials into the database, and returns the user to the landing page.

The “Register User ” button runs the **registeruser.php** script. It takes as argument a dropdown selection of every available user in the SQL database, automatically populated by the script. After selecting a user, the Admin can begin the registration process. This process takes place in the SmartSystem application itself, as a security feature, meaning a hacker cannot remotely insert a combination into the database.

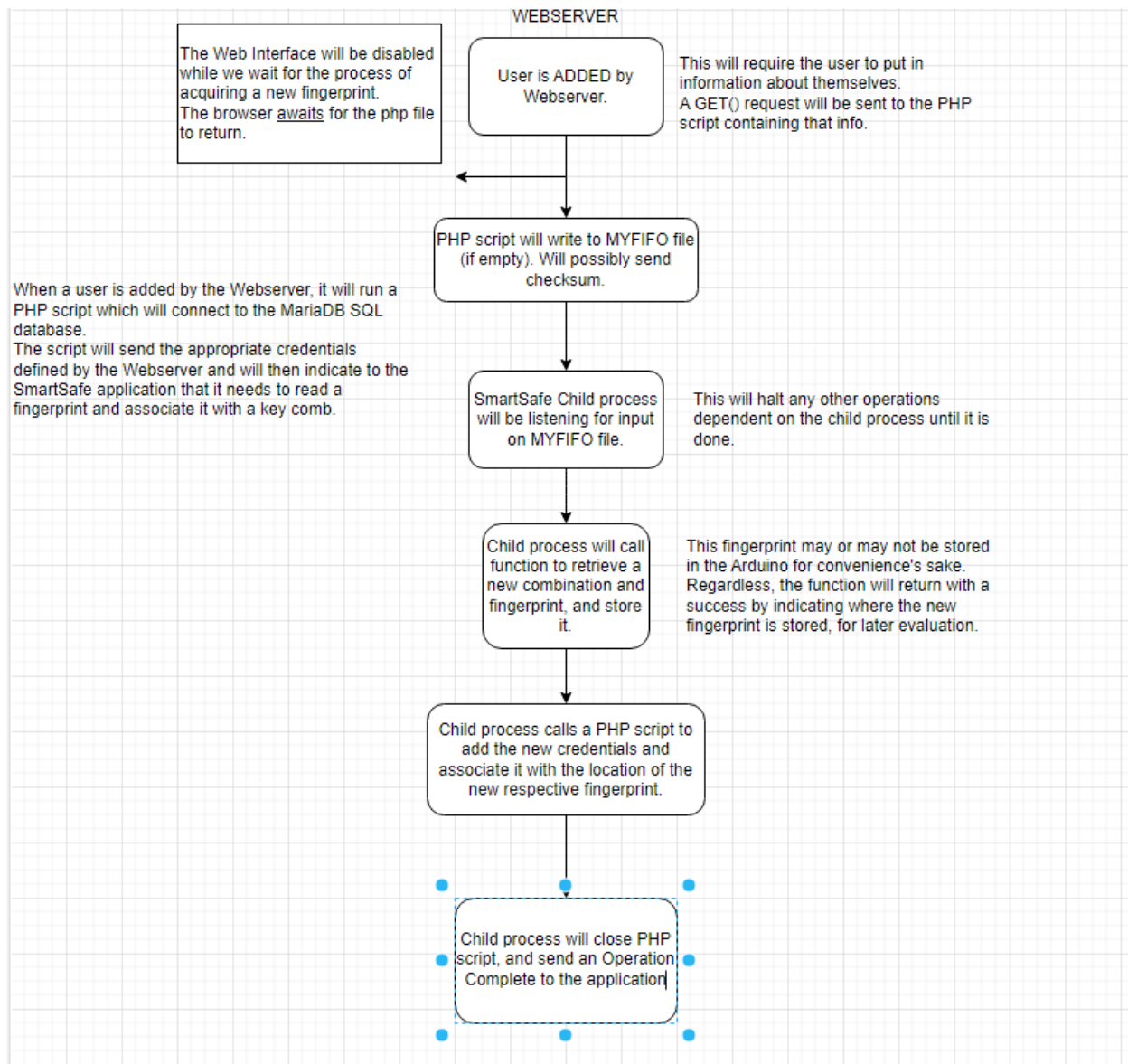
This script is run async thru AJAX, so as it runs it indicates to the Admin that the procedure now takes place manually on the safe itself.

After inserting a valid combination, this combination will be displayed on the webserver while simultaneously being registered into the SQL database.

This step **MUST** be done for new users, since not registering a combination will not bear any result when trying to open the safe. This step **CAN** be done for existing users, to change their combinations.

Keep in mind that the system will notice if more than one user with that combination exists, and when it comes time to open the safe it will treat it as an incorrect combination if more than one entry with it exists in the SQL database.

Given this fact, this function can also be used to delete a user by changing their combination to 000000. This combination is a default value that cannot be changed by regular means (aside from tampering with the database itself).



The above image is a conceptual flow chart I made when designing the Web application. I used this basic operation as the ground for the framework of the website.

Command	Function
q	Quit the program. Will properly send kill signals to the child process as well. This is the only way to properly terminate the program. Do not terminate it in any other way.
t	Test the connection between the parent and the child process. If the child process is hanging, this command will not return the status of child-parent pipes
o	Open the safe door. This sends a command to the AVR to forcibly draw the solenoid.
c	Close the safe door. This sends a command to the AVR to forcibly release the solenoid.
i	Force input mode on the AVR.
Test	Run test() function. This is for debugging purposes. The test() function is defined by a dev.

StartUp

In this section, we will go through the normal operation of the program, how it starts, how it evaluates a combination input, and how it quits.

This is a very limited scope of what the system is capable of, but it should give an idea of the design.

I will intentionally remove the **pipe** files so we can see the program behavior:

```

root@raspigui:/home/gui/AVRserial# rm pipe*
root@raspigui:/home/gui/AVRserial# ./avrserial
Making new FIFO file: pipe1
Making new FIFO file: pipe2
Listening for Webserver...
Connected to Webserver!

Read by child

Wrote to child

```


Let us go by sections of the output now.

As we can see above, the program detects that no named pipe files existed within the directory structure after running the **#rm pipe*** command. It then creates new pipes.

The code responsible for that behavior is:

```
umask(011);
//This statement checks the existence of a fifo file of name defined
//by MYFIFO1, and promotes its creation
//if it does not exist.
if(access(MYFIFO1, F_OK) != 0)
{
    if(mkfifo(MYFIFO1, S_IRWXO | S_IRWXG | S_IRWXU | 0777) == -1)
        {perror("Error making FIFO file."); exit(-1);}
    else
        printf("Making new FIFO file: %s\n", MYFIFO1);
}

//Same story with the following statement, but for file MYFIFO2.
if(access(MYFIFO2, F_OK) != 0)
{
    if(mkfifo(MYFIFO2, S_IRWXO | S_IRWXG | S_IRWXU | 0777) == -1)
        {perror("Error making FIFO file."); exit(-1);}
    else
        printf("Making new FIFO file: %s\n", MYFIFO2);
}
```

Then, we can observe the program saying it is listening to the webserver, and that it is connected finally to it.

This is the result of opening these FIFO files we just created in both the child process and the webserver application. Remember that, as explained above, the main SmartSafe application never talks directly to the webserver.

We are seeing this output on the program terminal only because the child process is writing those messages to the parent upon succeeding to connect.

Note: The message is supposed to be "Read by child\nWrote to webserver".

This was a typo I later corrected.

```
int fifofd_r = open(MYFIFO1, O_RDONLY);
if (fifofd_r == -1)
    {perror("Error opening FIFO file for writting."); exit(-1);}
else
    dprintf(write_parent, "Listening for Webserver...\n");

int fifofd_w = open(MYFIFO2, O_WRONLY);
if (fifofd_w == -1)
    {perror("Error opening FIFO file for writting."); exit(-1);}
else
    dprintf(write_parent, "Connected to Webserver!\n");
```

This connection status is given by the fact that the child process can successfully open the pipe ends. This is a good indication of successful connection, since in C the commands open(), read() and write() are **blocking** for FIFO files.

Meaning that, if on the other end the files have not been properly opened, the program will hang indefinitely (or until it receives a kill signal).

The opening of the pipes is done by the PHP script aptly named **connect_server.php**.

This script is run on a fork created by the parent process:

```
//-----
//The code below forks a child and tells it to run a PHP script.
//This is essential to the init sequence and ensures that the file structure
//to support this service exists. This script will interact with
//the main child process to validate it is possible to communicate
//between application and webserver
if(fork() == 0)
{
    system("curl -s http://localhost/connect_server.php");
    exit(0);
}
else
//-----
```

The script called is written as follows:

```
if($MYFIF01 = fopen($filename1, "w"))
{
    echo "</br>FIFO write pipe $filename1 opened.</br>Opening read pipe.</br>";
    if($MYFIF02 = fopen($filename2, "r"))
    {
        echo "</br>FIFO read pipe $filename2 opened.</br>Waiting for application response...</br>";
        //-----
        //The following statements will send an encapsulated "!" character to the listening cl
        //which has just succeeded in opening the pipes. This will indicate to the child that
        //can begin communications.

        $tmp = chr(0xF3);
        fwrite($MYFIF01, "$tmp$tmp", 3);

        //-----
        //The script will then wait for an answer from the child. This answer is mostly symbo
        //as it is not parsed in this script, but it is nonetheless necessary and will cause
        //the script to hang unless it is properly done.
        //This reply is automatically programmed to be sent by the child and is defined in
        //childbehaviour.h.

        $answer = fread($MYFIF02, 4);
        echo "</br>$answer";

        //-----

        fclose($MYFIF01);
        fclose($MYFIF02);
    }
    else echo "Unable to open MYFIF02 file $filename2</br>";
}
```

This same script is called async thru AJAX on the landing page of the webserver.

The last two lines of output are due to the child being able to receive information through those newly opened pipes.

As we can see in the image above, the PHP script will send a "!" character encapsulated by the Webserver interpretation tags, which will be evaluated at the child process.

The child process, listening for any input from the webserver, reacts to this preprogrammed command character in the following code:

```

else if(child_pfds[2].revents & POLLIN)    //Reads commands sent from Webserver
{
    read(fifofd_r, &fifo_buf, 512);
    if(*fifo_buf == 0xF3)
    {
        //The switch case statements below serve as a lookup table for commands
        //sent specifically by the Webserver.
        switch(fifo_buf[1])
        {
            //The webserver sends the character encapsulated "!" character
            //if it succeeds in opening the FIFOs.
            case '!':
                dprintf(write_parent, "Read by child\n");
                write(fifofd_w, "OK!", 4);
                dprintf(write_parent, "Wrote to server\n");
                break;

            //The webserver sends this signal to indicate the registration/update
            //of a user. The command is acknowledged and sent to the parent.
            case '+':
                dprintf(write_parent, "Registering user metrics...\n");
                sleep(1);
                dprintf(write_parent, "+\n");
                break;

            default:
                break;
        }
    }
}

```

Async AVR Input

```
root@raspigui:/home/gui/AVRserial# ./avrserial
Listening for Webserver...
Connected to Webserver!

Read by child
Wrote to server

SmartSafe Systems Version 9.0

By Guilherme G. Ruas

Smile for the camera!

Stand By Mode...
```

The above image shows a different output from what we saw previously. Aside from the already analyzed output lines, the rest are debug messages sent from the AVR to the RASPI and getting displayed.

These messages are part of the AVR's start up sequence but will not always show up since it is common for the embedded system to constantly be running.

Right now, the program is in STANDBY mode and no input can be sent from the AVR with the exception of the # key, which enables input.

When pressing it, the mode will change:

```
Stand By Mode...
```

```
Mode ON
```

```
Beginning input sequence from keypad...
```

Now the AVR program is in INPUT mode. It runs a loop scanning the columns of a 3 x 4 keypad installed on PORTC.

The details of this scanning process will be explained further in the user manual.

I will now press a single button, the 1 key:

```
Mode ON
```

```
Beginning input sequence from keypad...
```

```
1
```

The input processing is complicated, so it will be omitted. But the output of each key is the same, just a regular echo of the key.

I will now input a non-existent combination. This means that there are no users in the SQL database that have a combination corresponding to the one entered **OR** that there are more than one users with that combination.

The distinction is not made in the program, since I found it unnecessary, and the outcome should be the same: no access is permitted.

```
1
2
3
9
8
7
!123987!      >>Combination acquired successfully!

!!ERROR PROCESSING YOUR COMBINATION!!
This message will be displayed if you either entered a non-registered combination, or if there exists more than one user with that combination.
Please retry, or check with the System Admin for further instructions.
You have got 3 more tries before this incident is reported to the System Admin.
```

We will now briefly discuss the chronological sequence of events.

Upon entering the 6th digit, the program will stop fetching input and process the buffered combination that was just entered.

The whole combination is displayed to the user, with a success status upon it being successfully acquired.

The processing of the AVR inputs is all done by one function, so upon fetching a full combination, the return value of that function will tell the main program that it should do one of two things: keep going, register the combination to a user or evaluate the combination against existing users.

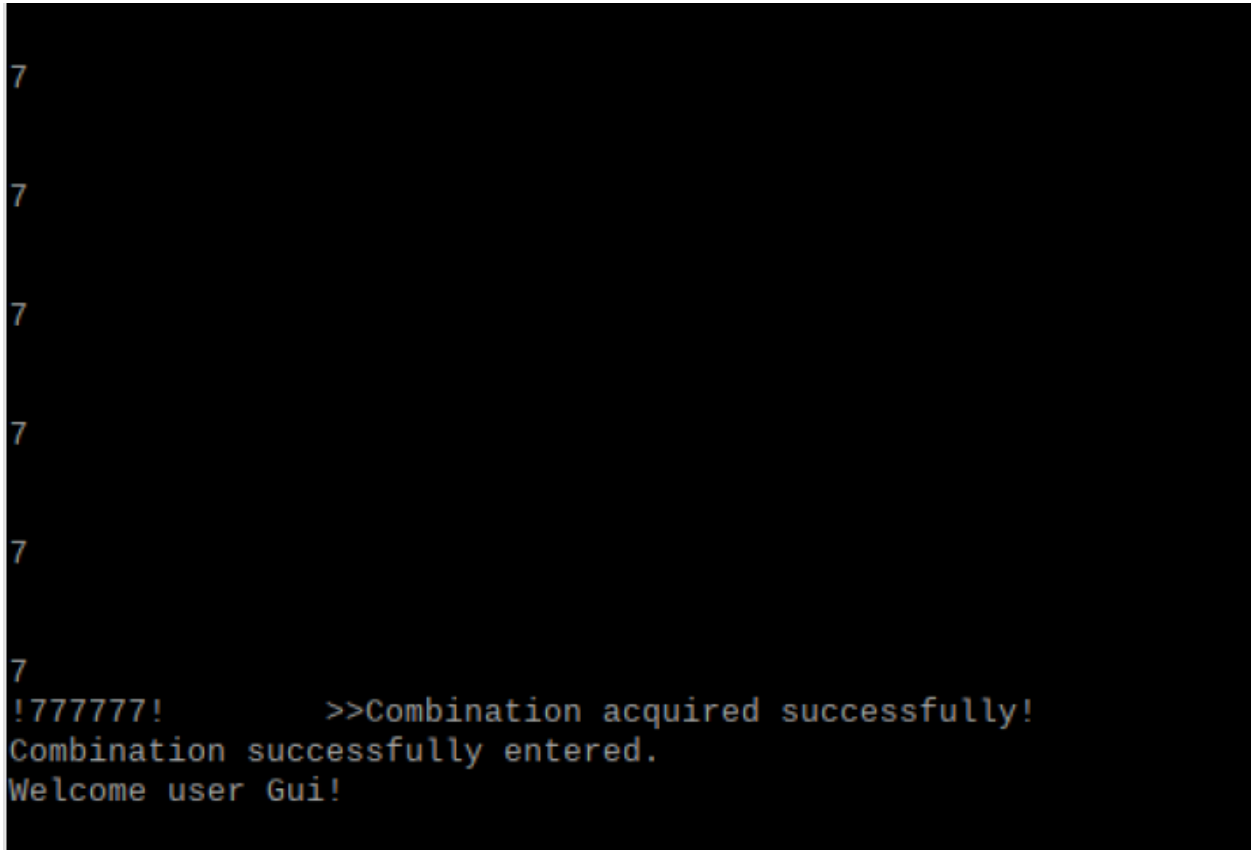
No matter what the case is, this combination is always logged into the file called **combinations.txt**.

Right now, it is used for debugging. But the final design will have a time stamp be written to the file with the fetched combination, as well as flagging combinations that resulted in violations

(will explain further in user manual). The Admin would then be able to directly inspect the file on the host machine, or through the Webserver application.

As we can see above, the combination failed, and it explains to the user the two possible causes for this error.

Now we will see an example where the combination exists, and I registered it under my own name:

A terminal window with a black background and orange text. It shows a series of seven '7' characters entered on separate lines. After the seventh '7', the program responds with three lines of text: '!777777! >>Combination acquired successfully!', 'Combination successfully entered.', and 'Welcome user Gui!'.

```
7
7
7
7
7
7
7
!777777! >>Combination acquired successfully!
Combination successfully entered.
Welcome user Gui!
```

The program dynamically fetched my username from the database and used it in its reply.

For now, this is all it does, as I am still debugging some things in preparation for integrating the Camera and Arduino/Fingerprint Scanner peripherals.

The code responsible for this part of the execution can be seen below (simplified):


```

if(pfds[0].revents & POLLIN)
{
    //printf("POLLIN\n");
    res = read(fd1, avr_buf, 255);
    avr_buf[res]= '\0';          /* set end of string, so we can printf */
    char *bufpt = &avr_buf[0];

    if (*bufpt == 0xF1)
    {
        avrresp = avrprocessing(bufpt, comb_buf);
        if(avrresp == 1)
        {
            //This checks if the AVRREG flag was set.
            //If it was, it will send the combination thru the pipe towards
            //the webserver using 0xF3 as an integrity check. The combination will still
            //be logged into the COMBTXT file.

            if(state == (STATE | AVRMODE | AVRREG))
            {
                //Sends reg_status to child, which sends to webserver
                dprintf(cmd_child_write, "reg_ok\n");
                dprintf(write_child, "%s", comb_buf);

                //Returns input state to default STATE
                state = (STATE | AVRMODE);
                printf("Registered user combination!\n");
            }
            else if(state == (STATE | AVRMODE))
            {
                user = comb_eval(comb_buf);
                if(user == NULL)
                {
                    int tries = 3;
                    printf("\n!!ERROR PROCESSING YOUR COMBINATION!!\n");
                    printf("This message will be displayed if you either entered a non-regi:");
                    printf("Please retry, or check with the System Admin for further instruc");
                    printf("You have got %d more tries before this incident is reported to ");
                }
                else
                {
                    printf("Combination successfully entered.\n");
                    printf("Welcome user %s!", user);
                }
            }
        }

        FILE * fd2 = fopen(COMBTXT, "a" );
        if (fd2 < 0) {perror(COMBTXT); exit(-1); }

        fprintf(fd2, "%s\n", comb_buf);
        fclose(fd2);
        memset(comb_buf, 0, sizeof comb_buf);
    }
}

```

Webserver Application

The webserver application part of this system is very important for any managerial functions. This interface offers a much easier-to-digest view of the system and allows the Admin to perform their tasks with less trouble.

So far, the application is a little bare bones. However, it is still complicated, and I will make it brief since this report is already huge.

Let us start at the landing page:

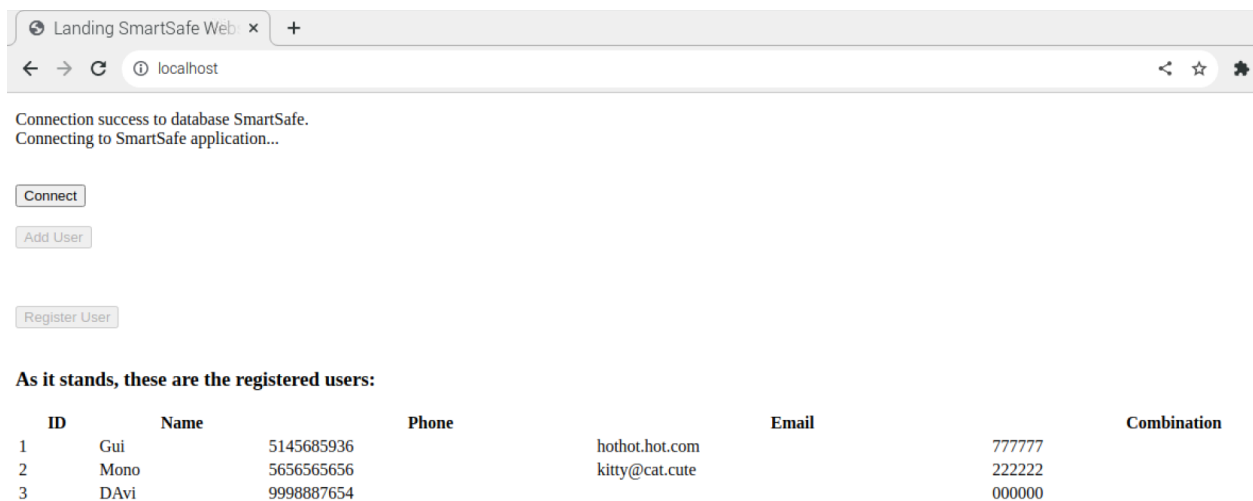


Fig 17: Landing page of SmartSafe System's Web application

We can observe in the image above the elements described in section 4, so I wont explain them again.

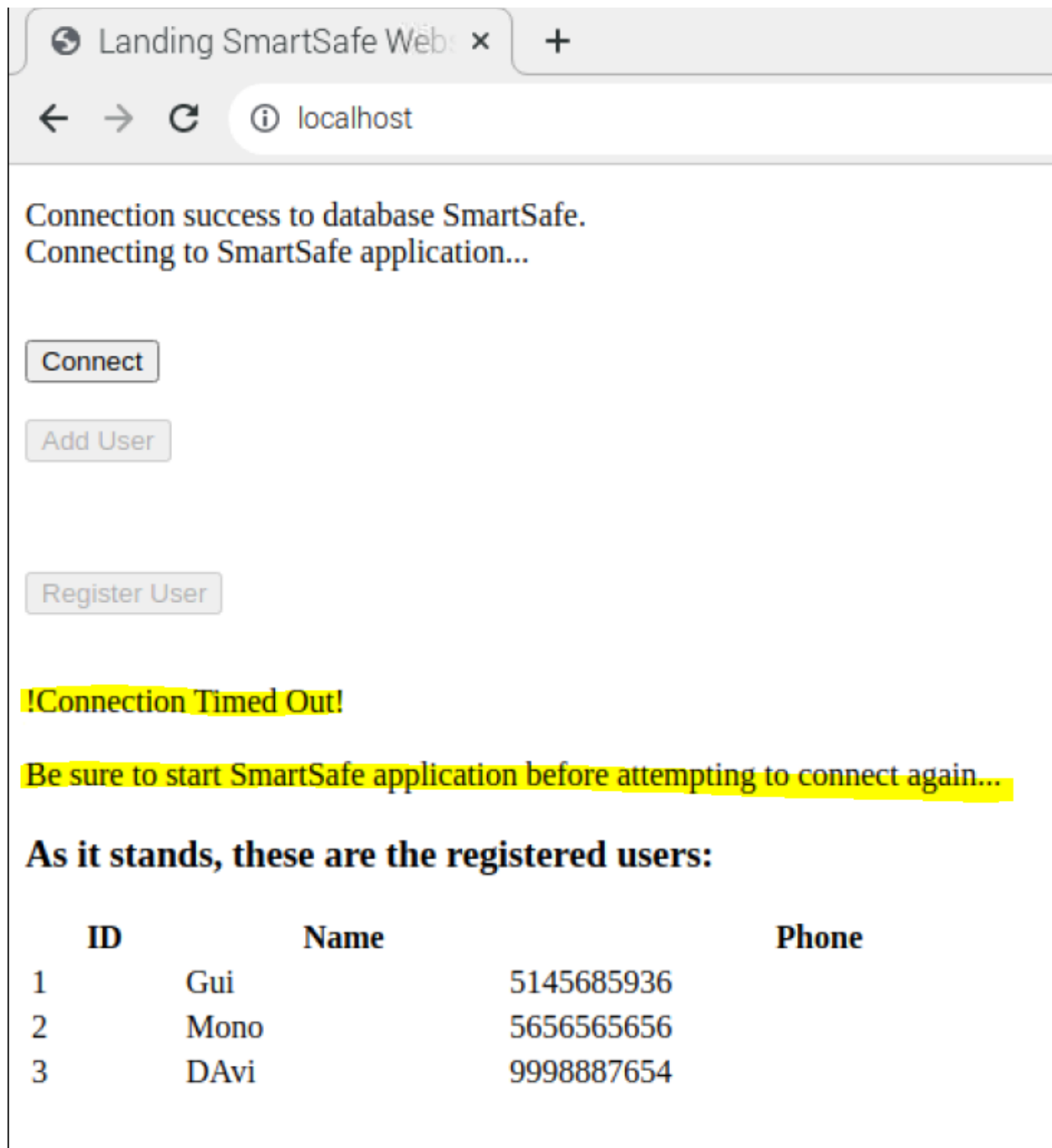


Fig 18: Landing page connection timed out

The above event happened because I pressed the connect button without having the SmartSafe application running. It took 3 seconds to time out.

Once I start the application and press the button again, we see:

Connection success to database SmartSafe.
Connecting to SmartSafe application...

Connect

Add User

Gui ▼

Register User

FIFO write pipe /home/gui/AVRserial/pipe1 opened.
Opening read pipe.

FIFO read pipe /home/gui/AVRserial/pipe2 opened.
Waiting for application response...

OK!

As it stands, these are the registered users:

ID	Name	Phone	Email
1	Gui	5145685936	hothot.hot.com
2	Mono	5656565656	kitty@cat.cute
3	DAvi	9998887654	

Fig 19: Landing page connection OK

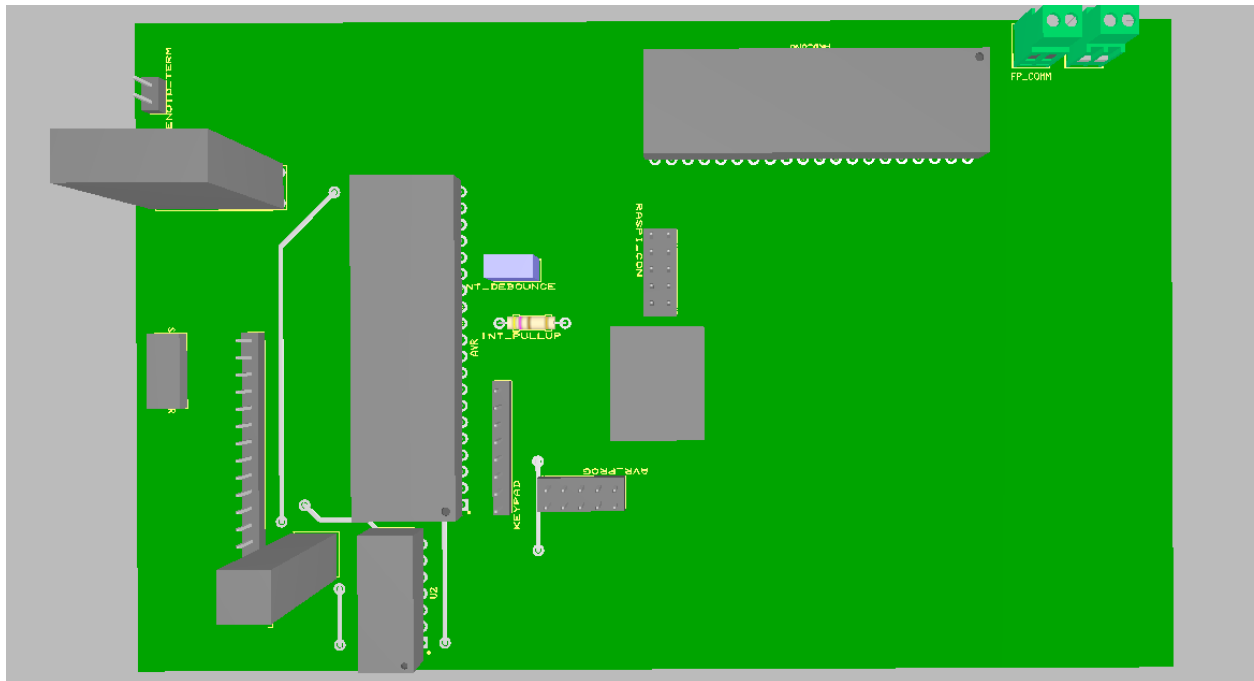
From here, the Admin can perform the operations already described in section 4.

I have refrained from putting any code into this document, since it is all too complicated and too numerous to even be worth it.

However, I thought I could include the landing page for the Webserver application such that the reader can understand a bit of the operations going on behind the making of such a simple landing page

Appendix

A. Illustrations



The above image is a 3D rendering of the board done by Ultiboard.

B. Code/Script Listings

There are three parts to this system:

1. Linux Application code: can be found on my Github page at [this link](#).
2. AVR embedded application code: can be provided upon request.
3. Arduino Sketch: can be provided upon request.

C. Bill of Materials

<i>Device</i>	<i>Quantity Ordered</i>	<i>Price (CAD\$)</i>
AC/DC WALL MOUNT ADAPTER 12V 18W	1	27.02
ULTRA-SLIM ROUND FINGERPRINT SEN	1	29.42
KEYPAD - 12 BUTTON	1	7.77
RELAY GEN PURPOSE SPST 5A 5V	3	4.71
FAN AXIAL 30X6.1MM 5VDC WIRE	1	16.69
ARDUINO NANO 33 IOT	1	37.75
INPUT PLUG US FOR ACM SERIES	2	6.26
CONN PWR JACK 2.5X5.5MM SOLDER	3	5.76
Raspberry Pi Camera, KEYESTUDIO Fisheye Wide- Angle Lens 5MP 1080p OV5647 Sensor Module	1	
<i>15 Pin 30cm 50cm 100cm FFC Ribbon Flexible Flat Cable for Raspberry Pi Module Camera</i>	3	11.49
RASPBERRY PI 4 MODEL B	1	N/A
ATMEGA8515	1	N/A
INCLINED ELECTROMAGNETIC LOCK-12 FIT0624	1	12.07
LOGIC LEVEL CONVERTER - BI- DIREC	2	10.08
LM317 VOLTAGE REGULATOR	2	N/A

The above items with the price labelled Not Applicable refers to devices I already have in my possession and were not purposefully bought for this project.