**Multi-Band RFID Reader & Registration System**

Steven Bagdasarians

Julian Saunders

Dawit Amare

Josh Guenther

Table of Contents

|  |  |
| --- | --- |
| Introduction | 3 |
| Hardware | 3 |
| Initial Setup | 5 |
| * Raspberry Pi | 5 |
| * Atmega328 | 7 |
| 125kHz Module | 7 |
| * Raspberry Pi Code | 7 |
| * Atmega328 Code | 8 |
| 13MHz Module | 9 |
| Communication with Server | 10 |
| Usage | 13 |
|  |  |
|  |  |
|  |  |

**INTRODUCTION**

The Electronics Prototyping Lab (EPL) is a Rapid Manufacturing lab in FAB 84-10. There is a lot of expensive and mildly complicated equipment in the lab. Some people who don’t know how to use the equipment come in and try to use the equipment and then end up either hurting the equipment or wasting time. We would like to be able to use hardware to control access of the equipment and streamline waiver sign-up for lab usage. The overall process of use is: lab user uses their student ID or provided RFID tag to identify themselves and the system then checks a database to confirm they have been trained/checked out on the equipment and then gives some visual/auditory feedback and may deny them access to the equipment by prohibiting it from being powered up.

**HARDWARE**

|  |  |
| --- | --- |
|  | Image result for mfrc522 rfid reader pinout |

**Connections:**

**Board to Pi:**

This board uses a 2x20 female pin header that is soldered to the BOTTOM of the board so that it may easily plug directly onto a raspberry pi. This board should sit directly over the Pi like a cape. From the figure above you can see the 2x20 pin header that will sit directly over the Pi and the two 5V pins on the Pi line up with the 5V line on the custom board.

**13.56MHz Module:**

The MFRC522 13.56MHz reader has 8 pins connected to it. It only needs 7 of these pins connected. The 1x8 pin header is available to make connecting the module easy. The power pin is labeled on the custom board as 3.3V above pin 1 of the pin header. These pins connect directly to the module in order from there. This module is using SPI interface.

**125kHz Antenna:**

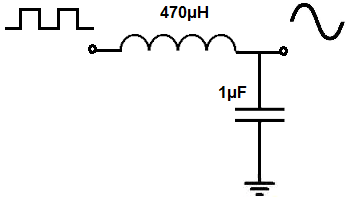
The 125kHz antenna is made from magnetic wire wrapped in a loop many times to act as an inductor. This loop antenna must be resonant with a capacitor (C3) to ground at our select frequency. Our antenna was measured and tuned to about 1.6mH which is resonant with a 1nF capacitor at 125kHz.

Note: It is easier to pick a capacitor value that you have and then tune the loop antenna to the correct inductance.

**Components:**

**PWM 125kHz Square to Sine Wave:**

To generate our 125kHz signal, we are using pwm from the on board ATMega328. This needs to be converted to a sine wave. By using a simple LC low pass filter (L2 and C18) resonant at our design frequency, we can filter out all the harmonics present in the square wave and be left with our fundamental frequency. Use the above equation for this as well.



These values are arbitrary and for another frequency.

Note: It may be easier to pick and inductor that you already have because we tend to have more capacitors than inductors. Our boards used 15uH and 100nF inductor and capacitor, respectfully.

**Power Supply:**

Our power supply circuit comes straight from the datasheet for the LM2596. It uses a 12V 2A supply from the wall and outputs 5V up to 3A. Our circuit only needs about 1-1.5A at 5V. The 33uH inductor (L1) must be rated to handle at least 1-1.5A of current. Our inductor uses 26 gauge magnetic wire wrapped about 23 times through a T50-2 red toroid core. For making your own inductor make sure it is at least 28 gauge or bigger.

Note: On current board layout, pins 2 and 4 on the LM2596 are switched. Modifications were made on current boards to correct this.

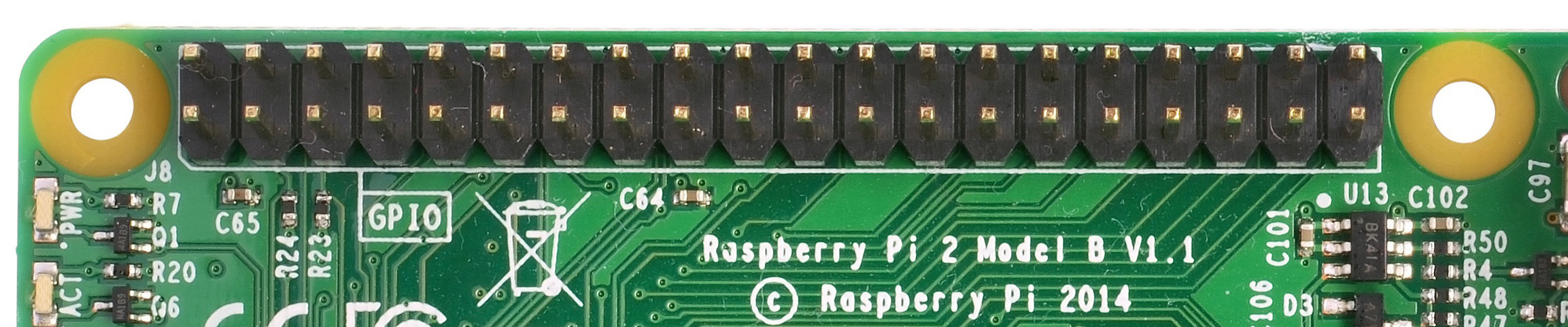
**Other Unlabeled Parts:**

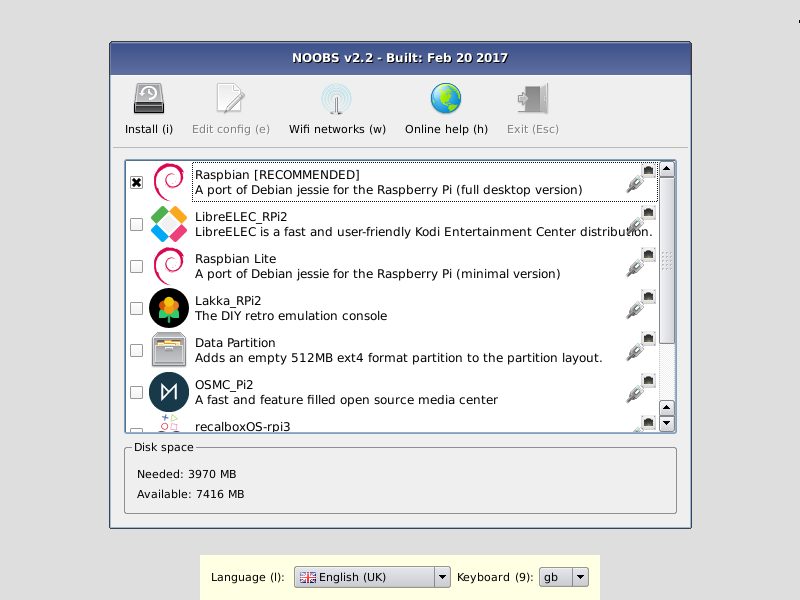
The schottky diode (D2) from the data sheet of the LM2596 was a 1N5824. That diode costs $47. Any other 30V 5A schottky diode will work.

The zener diode (D1) can be any 12V zener. Our board uses 1N759A from the EPL store.

**INITIAL SETUP**

**Raspberry Pi**



The Raspberry Pi 3 is a small single-board computer that can run Linux and various other low-power operating systems. In order for the custom board to attach to the Pi, it will need a 2x20-pin strip dual male header through-hole mounted onto the Pi. This should already be done on most current Raspberry Pi boards, but if not, it will need to be soldered on.

Before installing the custom board onto the Pi, it’ll need to have the proper operating system and scripts running on it.

These instructions are tested to work on Raspian Jessie. Initial Setup is to have a stable operating system installed on Raspberry pi.

Default User Name is pi, and Password is raspberry

1. Obtain a micro SD card.

2. Download Raspian Jessie, this link may be used:

http://downloads.raspberrypi.org/raspbian/images/raspbian-2017-07-05/

3. Type CTRL + ALT + T to open terminal.

4. Type sudo raspi-config

5. Select "Interfacing Options" then "SPI" on next dialogue box.

6. Enable SPI and I2C, then hit OK.

7. Reboot.

**Libraries to Include:**

These are the libraries needed to run this code with appropriate function calls. This list assumes Raspian Jessie operating system. Here are these libraries, use sudo apt-get install XXXX to install these, where XXXX is the library name as it appears in the designated repository. Make file takes care of linking of these libraries, however, -lXXXX of each library may be needed to run each library separately or together without a make file. In other words, linking libraries is required at compilation time. Then sudo privileges are needed to run.

Libraries:

PiGPIO (Needed for 125KHz module)

1. rm pigpio.tar
2. sudo rm -rf PIGPIO
3. wget abyz.me.uk/rpi/pigpio/pigpio.tar
4. tar xf pigpio.tar
5. cd PIGPIO
6. make
7. sudo make install

Wiring Pi (needed for 13.56MHz module)

1. sudo apt-get install git-core
2. git clone git://git.drogon.net/wiringPi
3. tar xfz wiringPi-98bcb20.tar.gz
4. cd wiringPi-98bcb20
5. ./build

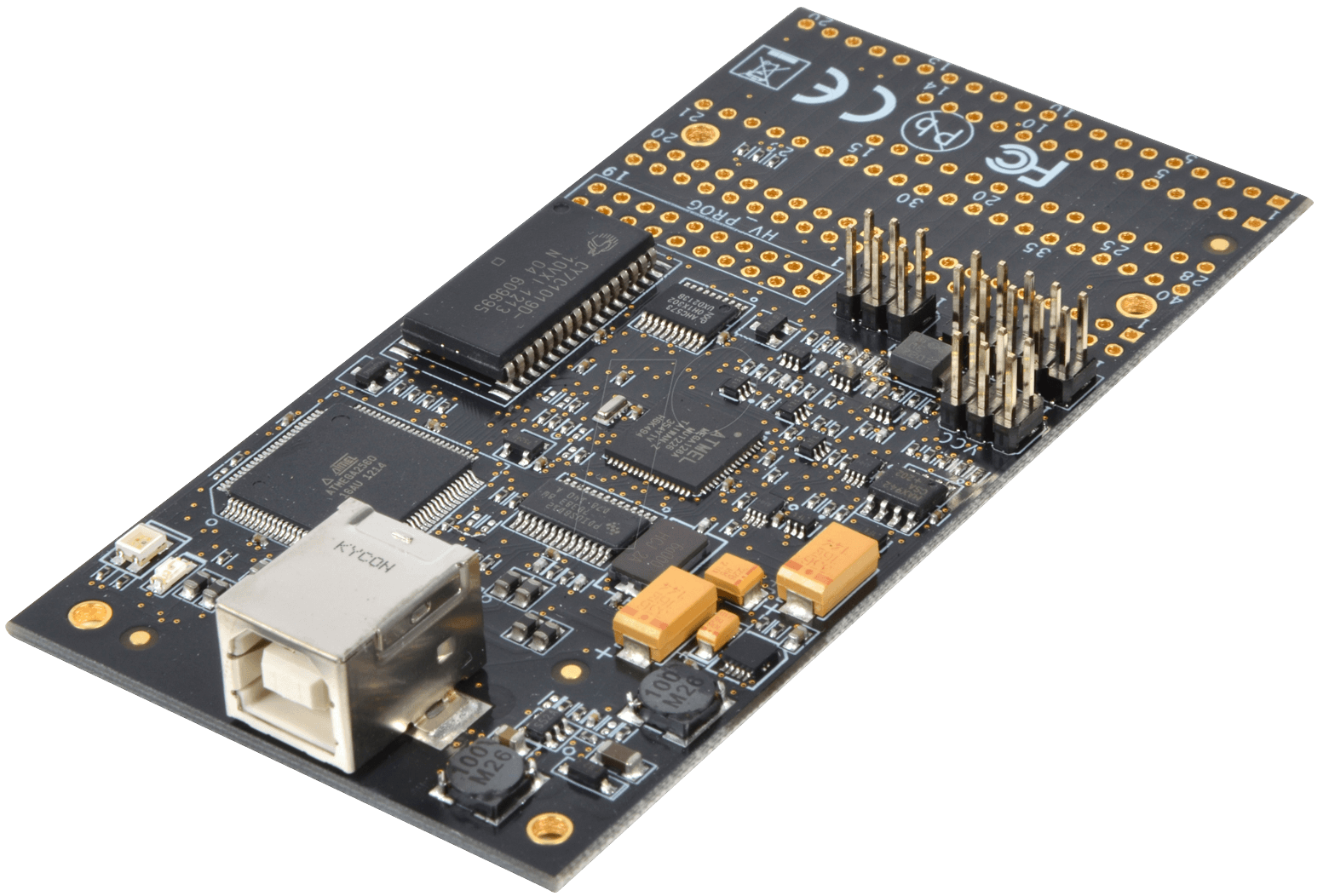
Curl

1. Sudo apt-get install curl (or sudo apt-get install libcurl4-openssl-dev)

BCM2835 (needed for 13.56MHz module)

1. Please see below for instructions

Reboot and sudo apt-get update and check if all libraries are installed properly.

**Atmega328**

Atmega328 is used to generate a 125KHz carrier signal into the antenna and collect incoming data from the demodulating circuit. To process these, we first need to program the chip by loading the Arduino bootloader. There are several methods of doing this, such as using the AVR programmers found in Capstone Lab via the ISP pin headers, using your own USBasp cable, or alternatively you can use an already bootloaded chip on an arduino. Then the following code is written to ensure functionality for the 125KHz module.

**125kHz Module**

**Raspberry Pi Code:**

The code on the Raspberry Pi is responsible for the remaining portion of the 125KHz reading and all of the 13.5 MHz reading. It also handles sending out HTTP requests to the server to validate any badge numbers that have been read by either frequency reader. The return value from the HTTP request is used to determine what action should be taken like lighting LED’s and sounding a buzzer.

**funcs.c:**

**read\_station\_info(char\* station\_id):**

This function handles the retrieval of the current stations identification information. It first attempts to open “station\_info.txt” to read the information stored. If the file exists then the information is retrieved. If the file does not exist then the user is prompted to create a name for the station and it is stored in the newly created text file.

**i2c.py:**

This python script handles the incoming I2C communication from the ATmega328 to the Raspberry Pi. The script sets up the Rpi as as slave device and begins listening. When a value is received, the data is written to the “data.txt” file and increments the count. This continues until the desired number of bits has been read.

**NOTE:** This code needs to be replaced with a pure C version of I2C at some point as calling it from C in with the python interpreter does not properly release the memory it uses resulting in the program crashing after 1-2 reads.

**read\_125.c:**

**read\_125():**

This function handles the information received from the 125KHz reader. Using the python interpreter, it launches the i2c.py script above and wait for it to complete. Next it opens the file the python script wrote to and reads in all the 1’s and 0’s that correspond to the swiped badges binary card number. The data is then searched for a bit pattern that represents the start of the badge number. Once found, the function returns the 24 bits corresponding to a badge number to the top.

**main.c:**

This is the main file that generates the station executable. The program starts by initializing all the required resources for both 125KHz and 13.5MHz readers. It then reads in the station information and stores it in the variable “station\_id.” Next it starts the 125MHz and 13.5MHz functions via threading to have concurrent reading. When one of the functions returns a card number, it is sent along with the station id to the server for verification. If the badge was good, the green LED lights up, otherwise the red LED lights.

**Station\_reset.c:**

This file generates a standalone executable that allows the user to change the station id stored in station\_info.txt.

**ATmega328 Code:**

The code on the ATmega328 is for the 125KHz reader and takes care of three main responsibilities. Generating the carrier wave to power the antenna, reading incoming waves from the antenna circuit, converting and sending data to the Raspberry Pi.

**Setup():**

This function handles initializing all register values on the chip along with pin modes and I2C setup. For signal output, registers are initialised to use Timer2 in phase correct mode with no prescaler and output compare mode. Output compare registers A & B are set to give us a 50% duty cycle with a resulting 125 KHz square wave. For signal input, registers are initialised to use the input capture register ICR1 with noise cancelling and rising edge detection.

**ISR(TIMER1\_CAPT\_vect):**

This is an interrupt service routine function that is called whenever there is a change from logic low to logic high on the signal input. The time in microseconds in the Timer1 register is recorded into the “sec” array at the current index handled by “read\_counter.” . Timer1 is then reset to 0 and the function checks the value it just read. The values we consider valid are 64 or 80 microseconds +/- 5 for accuracy. If the current value read is not within the acceptable range then we treat it as noise and set the read\_counter variable back to 0, otherwise we increment the read\_counter variable for the next read. This helps filter out all of the obvious logic changes due to noise on the input line.

**Loop():**

This is the main function in the code that repeats indefinitely. The beginning of the loop initialises/resets the variables needed along with enabling interrupts for the input capture register. The function then waits until the desired number of read operations have occurred by checking if the read\_counter is greater than the desired LEN length defined at the top of the code. Once the desired number of reads have occured, the input capture timer is disabled so the information can be processed. Next the function iterates through indexed value of the “sec” variable. The value is divided by two due to the clock running twice as fast because of prescaler limitations for Timer1. The value is the compared to the “comp” variable which is intended to be set between 64 and 80. The value can be changed to fine tune the accuracy of the circuit as needed. If the returned value is above the compare value then a 1 is written to the current index of the “badge” variable, and a zero is written otherwise. The value in badge is then sent via I2C to the Raspberry Pi before continuing to the next iteration. Once all values of “sec” have been processed the function returns to the top and resets to begin a new read.

**13MHz Module**

**Description**

This part of the instruction shows the setup to install and run 13.56 MHz MIFARE protocol tags. The datasheet describes that The MFRC522 supports all variants of the MIFARE Mini, MIFARE 1K, MIFARE 4K, MIFARE Ultralight, MIFARE DESFire EV1 and MIFARE Plus RF identification protocols. This is a proximity tag reader in the HF range upto 4 Inches and is suitable for products such as public transport, eGov, Banking, access, NFC phones.

The following is instruction on how to set up the 13.56 MHz Module. This program is modified from source obtained from the following github repository by Paulvh.

1. Make sure you have SPI in raspi-config (advanced section)

2. Make sure that Device tree is enabled in raspi-config (advanced): dtparam=spi=on

cd /home/pi (assuming this is your home directory)

BCM2835 library (library 1.52 was the current version at the time of writing)

Install latest from BCM2835 from : <http://www.airspayce.com/mikem/bcm2835/>

1. wget http://www.airspayce.com/mikem/bcm2835/bcm2835-1.52.tar.gz

2. Current version is included in this git repo

3. tar -zxf bcm2835-1.52.tar.gz

4. cd bcm2835-1.52

5. ./configure

6. sudo make check

7. sudo make install

8. cd ..

RC522 utility

1. tar -xvf rc522.tar

2. cd rc522

3. ./mc # compile executable

4. sudo cp RC522.conf /etc

5. sudo chmod 666 /etc/RC522.conf

6. edit /etc/RC522.conf # if necessary

Reboot

cd rc522 and Run command as root : sudo ./rc522

**Communication with Server**

**HTTP** orHypertext Transfer Protocol is an application protocol designed to enable communication between a client and a server. In this particular case, the station modules are designed to send an HTTP request to an API designed by the CS Team, which will then make appropriate changes to the server database. Afterwards, the API returns a response to the station modules, which then decide the appropriate action to take. The modules send data to the server using one of the most common HTTP methods, POST, which allows information to be stored in the request body of the HTTP request. The modules send data in the form of:

|  |
| --- |
| POST HTTP/1.1  Host: https://ruby.cecs.pdx.edu:3001/  User-ID-String: XXXXX  Station-ID: XXXXX  Station-State: Enabled  /api/user-access |

The first line states the method type (POST) and version (1.1), while the 2nd line is where the data is being sent. The lines afterward are the headers (User-ID, Station-ID, etc) and the body of the request, which hold the data we are sending. The headers in particular are what is going to contain the User and Station information. More on how the data is formatted can be found in the CS Team’s [documentation](https://github.com/LUCCA-Capstone/LUCCA/wiki/LUCCA-API).

**HTTP Code:**

The section of code written for sending HTTP requests makes use of the libcurl library. Provided below are explanations on each function used chronically, however, further documentation on the library and it’s interface can be found on their website:<https://curl.haxx.se/libcurl/>.

**user\_access(char \*user, char \*station, char \*state):**

Once the User-ID is received from the RFID, it’s passed into this function to be parsed into the correct HTTP format and sent to the API. Each argument passed in is placed into a char string that is then concatenated into a linked list of strings. The following is a list of each libcurl function and how it’s used:

**curl\_easy\_init:** In order to use the libcurl’s interface, a “curl handle” must be initialized. Before using this, a pointer to a variable of type “CURL” must be declared. EX: CURL \*curl;

**curl\_slist\_append:** Appends a string to a linked list of strings. In this case, it takes every argument passed in and appends it to “headers.” This contains the aforementioned headers in the boxed POST format above.

**curl\_easy\_setopt:** The main function used for applying the appropriate settings for the data transfer. All settings are set by including the curl handle, a “behavior option,” and a parameter. EX: curl\_easy\_setopt(CURL \*curl, CURLoption, parameter)

|  |  |
| --- | --- |
| **Behavior Option** | **Description** |
| CURLOPT\_HTTPHEADER | This passes in a pointer to the list of header strings that was concatenated. |
| CURLOPT\_URL | This passes in a pointer to the URL for the server that data is being sent to. |
| CURLOPT\_SSL\_VERIFYPEER | This determines if CURL checks the authenticity of the API’s certificate. Set this to 1 if the API has a legitimate certificate. |
| CURLOPT\_SSL\_VERIFYHOST | Checks if the server host name provided in the certificate matches the one being sent to. Again, set this 1 if the API has a legitimate certificate. |
| CURLOPT\_POSTFIELDS | This is where the data being sent to the server is entered. |
| CURLOPT\_POSTFIELDSIZE | This sets the size of the data that is being sent in the CURLOPT\_POSTFIELDS option. |
| CURLOPT\_SSLCERTTYPE | This is needed to specify certificate type. Supported formats include PEM and DER. |
| CURLOPT\_SSLCERT | This will apply the certificate provided. |
| CURLOPT\_HEADERFUNCTION | This is how the program receives header values from the API. |

**curl\_easy\_perform**: Once ‘*curl\_easy\_init’*is called andall the behavior options have been applied via ‘*curl\_easy\_setopt’*, this function will perform the data transfer. The function will return a 0 if the transfer was successful, and will return a non-zero if an error has occurred.

**curl\_easy\_getinfo:** This function extracts information from the curl handle, though in this particular case, we only use it to get the response code (CURLINFO\_RESPONSE\_CODE). Within the user\_access function, this response code is what determines if a user was in the database. A list of response codes that may be sent back from the server API are provided below:

|  |  |
| --- | --- |
| 200 OK | Indicates that the server has found the user in the database |
| 400 Bad Request | Indicates that the request was sent incorrectly, such as a missing required header, or an unrecognized route. |
| 401 Unauthorized | Indicates that no authentication certificate was found |
| 403 Forbidden | Indicates that an unrecognized Station-ID was sent |
| 500+ | Indicates that the module is unable to connect to server |

The user\_access function is currently set up so that any response code other than 200 will be a 0 or fail.

**curl\_slist\_free\_all**: Just for clean up purposes, this function clears the headers that were passed in.

**curl\_easy\_cleanup**: This function will end the current curl handle. In order to do another transfer, the curl handle will need to be initialized again via **curl\_easy\_init.**

**void heartbeat(char \*station):**

This function is provided simply to allow the station modules to send a heartbeat packet to the API. It was specified within the CS Team documentation as a method to keep track of stations states. It works similarly to user\_access, however it only requires the station state and station ID. Assuming the pulse was successful, the API will send back a “200 OK” response code, as well as a response header that bears the current date and time.

**void init\_string(struct string \*s)**

This function sets up the buffer (struct string) that is used in the callback function “writefunc().”

All it does is takes the pointer from the struct passed in and allocates memory for the string. What this buffer is used for is explained below.

**size\_t writefunc(void \*ptr, size\_t size, size\_t nmemb, struct string \*s)**

This function is for allocating memory and storing the data that is sent back from the API. This is not required for the response code, but anything else received, such as headers (used for the heartbeat function), will need a dynamically allocated buffer. This function is used within “curl\_easy\_setopt(curl, CURLOPT\_HEADERFUNCTION, writefunc)” to place the headers into the buffer.

**Usage**

The usage is as simple as scanning a card or tag and the device communicates with the server. Once the server receives the data sent from the station modules, it’ll return one of the response codes listed above.

The hardware is currently set up so that the green LED on the board will blink and the buzzer will go off momentarily if the response code “200” is sent back. As mentioned in the code table above, this implies that the API not only successfully received the data properly, but the User-ID that was sent is also found in their database.

The code can be amended to provide more variations of LED and buzzer patterns in order to take into account all response codes. However, it is currently set up to only light up the red LED and buzz for an extended period of time should anything other than “200” be received.

That said, the terminal on the RPI’s OS will list off the response code when running the HTTP script, so debugging will be possible as long as the terminal is open. Using the code table above as a reference should make it relatively easy to fix any issues that may arise.