International University of Sarajevo

Faculty of Engineering and Natural Sciences

EE325 Final Project Report

“Car parking monitoring and management system”

Prepared by:
1. Harun Tucaković ID: 180302046
2. Amar Halilović ID: 160302107

Professor:
Dr. Izudin Džafić

May, 2019

Contents

Overview

Car parking monitoring and management system project is trying to provide autonomous parking system intended for specific group of people, such as tenants of specific building, or parking of the specific institution. Parking system that is developed in this project is able to, with no additional human interference, function and manage whole parking. Meaning, it can let correct users in and out of the parking based on the database of all allowed users provided. Then, it keeps track who entered, who exited parking, how many users and which users are currently on the parking.

For all this, system has GUI application which allow manager of the system to supervise the system. Manager is able, through the GUI application, to monitor all allowed users, all active users (users that are currently on the parking), to add or delete users from the database. Manager is also able to manually control hardware parts of the system (ramps).

System that is developed also has features that prevent malicious use of the parking. For example, it will not open entrance ramp for the user whose tag is recorded to be on the parking. Also, it won't open exit ramp for user whose tag is recorded as absent from the parking.

Hardware

Car parking monitoring and management system project consist out of five hardware components. The main component in the project is Raspberry Pi, core of the system. Other components are servo motors, proximity sensors, Wiegand RFID readers with button pad and RFID tags, ESP8266 microcontrollers.

Raspberry Pi

"The Raspberry Pi is a low cost, credit-card sized computer that plugs into a computer monitor or TV and uses a standard keyboard and mouse."[1]

The Raspberry Pi is basically a small computer that runs the Raspbian operating system. The Raspberry Pi has 40 GPIO pins. GPIO stands for General-Purpose Input/Output. These pins are used for direct communication between raspberry pi and the outside world.

In the project is used latest version of Raspberry Pi, Raspberry Pi 3 Model B+. Raspberry Pi is the core of the project. The main part of the code runs on Raspberry Pi. The code with server that communicate with GUI application and with microcontrollers is on Raspberry Pi. Servo motors and proximity sensors are connected to and controlled from Raspberry Pi over GPIO pins.

[1] https://www.raspberrypi.org/help/what-%20is-a-raspberry-pi/

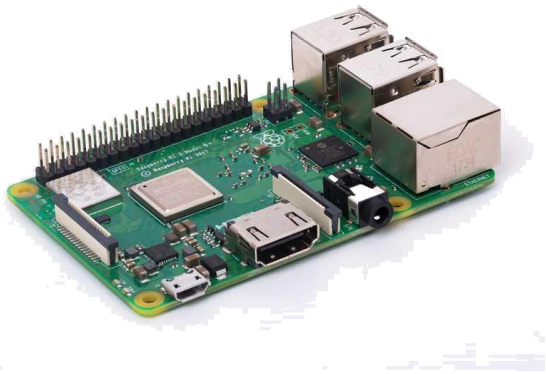


Figure 1 Raspberry Pi [2]

Servo motors

"A servo motor is a rotary actuator or motor that allows for a precise control in terms of angular position, acceleration and velocity, capabilities that a regular motor does not have."[3]

The servo motors are moving in specified angels. Possible movement is 180^{o}. The servo motors are controlled with pulse width modulation. The motor has neutral position, where it is possible to rotate to each side by 90^{o}. The minimum pulse is used when motor needs to rotate counterclockwise and maximum pulse rotates motor the clockwise.

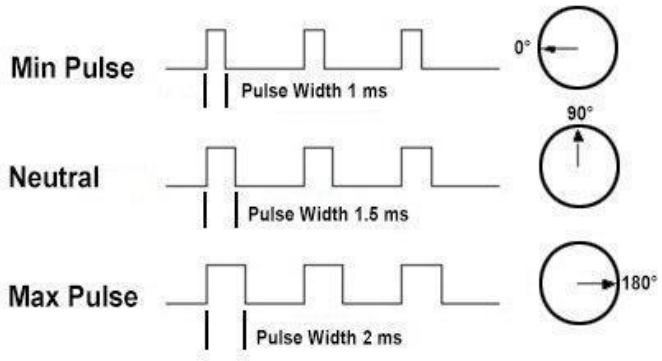


Figure 2 Variable Pulse width control servo position [4]

Two servo motors are used in this project and they are responsible for controlling the ramps, one for the entrance and one for the exit. They are connected to Raspberry Pi and opened and closed with signal that is sent from Raspberry Pi. They are opened when valid RFID tags are read or by manager. When opened, ramps stay opened for 8 seconds if no car was detected by IR proximity sensor under the ramp, if car is detected by

[2] https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/
[3] https://www.techopedia.com/definition/13274/servo-motor
[4] https://www.jameco.com/jameco/workshop/howitworks/how-servo-motors-work.html

the sensor (car passes) ramp will be closed as soon as IR proximity sensor sends signal that there is nothing under the ramp.

Figure 3 Servo motor [5]

IR Proximity sensors

An IR proximity sensor is used for detecting a presence of object without physical contact. The IR proximity sensor works by sending an infrared light and getting reflection in the case where there is presence of the object.

Proximity IR sensors in this project are placed under the ramp in the place where car must go over it. The sensors are used to prevent ramp from hitting the car in the case where car is staying, for some reason, under the ramp. Sensors are also used to check did car actually entere the parking.

Figure 4 IR Proximity sensor [6]

ESP8266 microcontrollers

"A microcontroller is a computer present in a single integrated circuit which is dedicated to perform one task and execute one specific application."[7] Microcontrollers have processor and programmable input/output pins used for communication and control of sensors and other devices.

[5] https://robu.in/product/towerpro-sg90-9gm-1-2kg-180-degree-rotation-servo-motor-good-quality/
[6] https://www.makerlab-electronics.com/product/ir-proximity-sensor/
[7] https://www.techopedia.com/definition/3641/microcontroller

In this project two ESP8266 microcontrollers are used. They are connected with Wiegand RFID readers which are reading numbers entered with keypad or number that is read from RFID tags directly. Microcontrollers are connected to the server on the Raspberry Pi and are sending 7-digit number from RFID readers to the Raspberry Pi.

Figure 5 ESP8266 microcontrollers [8]

Wiegand RFID readers and tags

RFID stands for Radio Frequency Identification. As name said RFID reader is a component which sends radio waves using antenna that is part of RFID reader to identify a RFID tag. In the RFID tag there is a chip that can perform some logical operations. When tag receives signal from reader it starts to return a radio signal back. "The Wiegand interface is a de facto standard commonly used to connect a card reader or keypad to an electronic entry system."[9]

In the project are used two Wiegand RFID readers that are connected to ESP8266 microcontrollers. RFID readers are sending its input to microcontroller which sends processed signal to the server on Raspberry Pi. There are two ways that RFID can read data is a button pad on the reader. One way is by entering number via button pad, another way is by scanning RFID tags.

Figure 6 Wiegand RFID readers with keypad [10]

[8] https://medium.com/@rxseger/esp8266-first-project-home-automation-with-relays-switches-pwm-and-an-adc-ad25f317c74f
[9] https://github.com/monkeyboard/Wiegand-Protocol-Library-for-Arduino
[10] https://eworkaccesscontrol.en.made-in-china.com/product/osPmfiIAvxhW/China-Access-Control-Keypad-RFID-Reader-Smart-Card-Reader-Access-Control-System.html

Physical connections

Each of components, RFID readers, servo motors and IR Proximity sensors, need 3 connection with Raspberry Pi or microcontroller.

Physical connection on Raspberry Pi

Raspberry Pi is connected with two servo motors and two IR proximity sensors. Those connections take twelve pins on Raspberry Pi. Servo motors are connected on a power supplies of 3.3 volts. Those are pins 1 and 17. Signals used for control of the motors are sent from GPIO 5 for exit motor (pin position 29) and from GPIO 20 for entrance motor (pin position 38). For motors ground any Raspberry Pi ground can be used, there are 8 ground pins on the Raspberry Pi.

IR Proximity sensors are using 5 volts pins, those are pins at positions 2 and 4. Entrance sensor is using GPIO 18(pin position 12) for signal transfer and exit sensor is using GPIO 17(pin position 11). They are also using any free ground pins on Raspberry Pi.

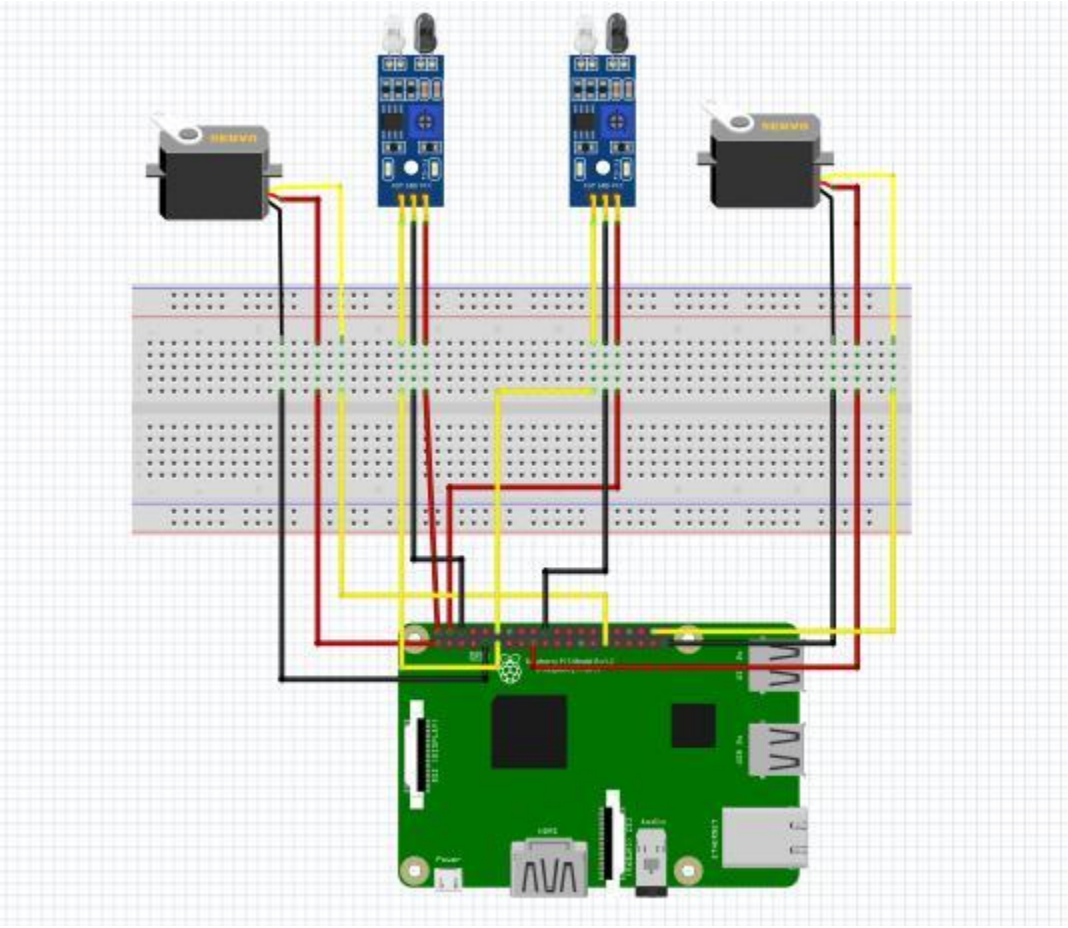


Figure 7 Physical connection with Raspberry Pi

Physical connection on ESP8266 microcontrollers

ESP8266 microcontrollers have only one component connected on them, RFID readers. Those readers need two pins on the microcontroller for data transfer. For data transfer GPIO5 and GPIO4 are used on ESP8266 microcontrollers. Wiegand reader is power of the 9V DC source, and its ground should be connected to the ground of the microcontroller for correct signal transmission.

Picture below shows how to connect Wiegand reader to the Arduino Uno. It is same process for connecting it on ESP8266 just use GPIO5 for Data1 and GPIO4 for Data0 on microcontroller.

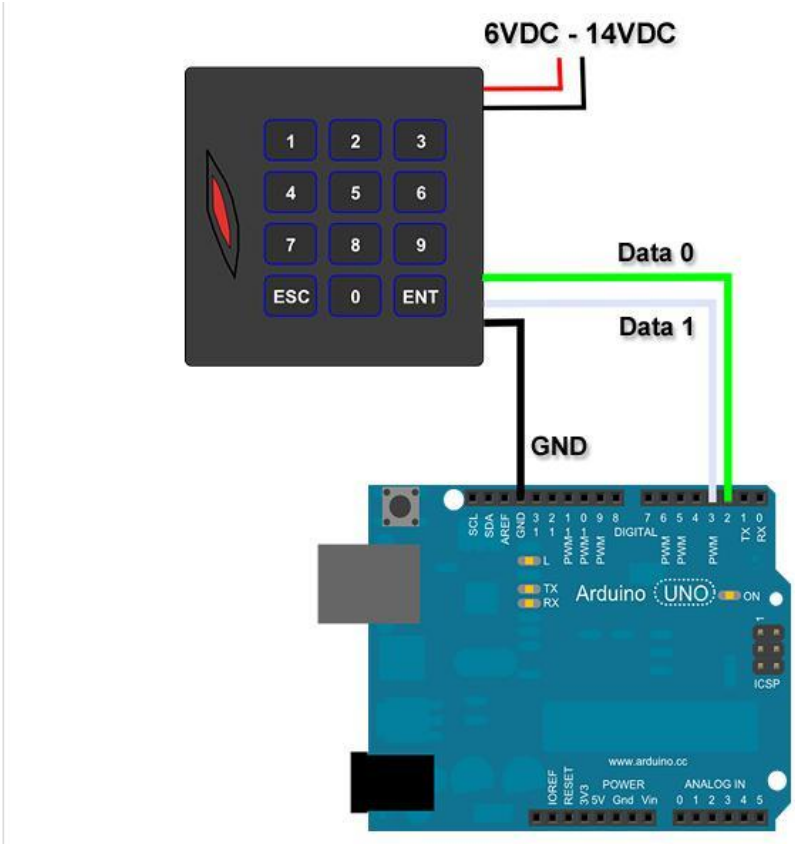


Figure 8 Physical connection with ESP8266 microcontroller[11]

[11] https://github.com/monkeyboard/Wiegand-Protocol-Library-for-Arduino

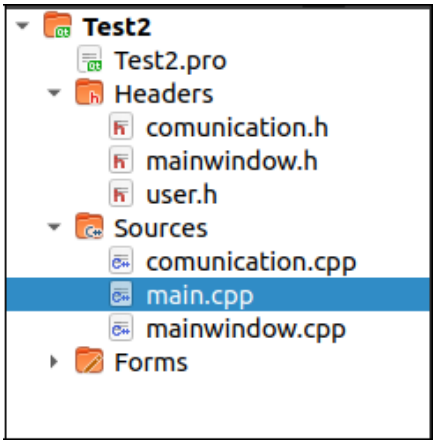
Software

This project is for the most part written in C programming language, though some parts are programmed in C++ and Qt. Platforms and software that was used for development of this project is: Visual Studio Code, Vim text editor, Qt Creator and PlatformIO IDE.

Visual Studio Code is used as main text editor for writing project code. Vim text editor is used to modify and change code while connected to the Raspberry Pi over the SSH connection. Qt Creator is used for GUI application development. And PlatformIO IDE is used for ESP8266 microcontroller development and upload of the code to the microcontrollers.

Software part of this project can be divided in three main parts: GUI application, code that runs ESP8266 microcontrollers and code that runs on Raspberry Pi, core of this project.

GUI Application



GUI application for this project is developed in C++ and Qt and it is eveloped to run on Linux OS. GUI application code consists out of six files, three .c and three .h files.

File main.cpp just contains main method in which QApplication and MainWindow objects are created and made visible.

File user.h contains struct definition which is designed to store data about single user. User struct has instance variables of user id (QString), name (QString) and surname (QString), constructor to initialize these instance variables, and friend helper functions to simplify data stream from struct to files and from files to struct.

Network programming

File communication.cpp contains all functions that are needed for network communication of the GUI application with core application on Raspberry Pi. Network communication is programmed using C socket programming. File communication.h is just a standard header file which contains base data and function signatures.

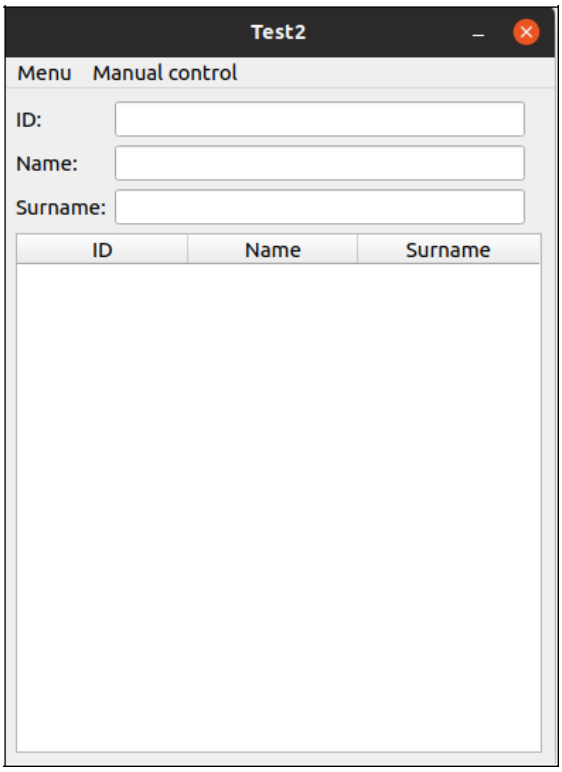
Main logic behind network programming for GUI application is to create client which will connect to main server on Raspberry Pi and send one byte header to the server. Server on the Raspberry Pi will read the header and execute appropriate command, or respond back to the GUI application with appropriate data.

```
if ((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
{
    perror("socket");
}

srv.sin_family = AF_INET;
srv.sin_port = htons(PORT);
srv.sin_addr.s_addr = inet_addr(ADDRESS);

if (connect(fd, (sockaddr *) &srv, sizeof(srv)) < 0)
{
    perror("connect");
}

if (write(fd, writeBuffer, 2) < 0 ) {
        QMessageBox messageBox;
        messageBox.critical(0,"Error","Write error!");
        messageBox.setFixedSize(500,200);
}

size = sizeof(readBuffer);
n = 0;
p = readBuffer;
do {
    if ((n = read(fd, p, size)) < 0) {
        QMessageBox messageBox;
        messageBox.critical(0,"Error","Read error!");
        messageBox.setFixedSize(500,200);
    }
    p = p + n;
    size -= n;
    total += n;

} while (n > 0);
```

In the code snippet above we can se how network client is programmed for the GUI application. Above code is just a part of much larger code, but it is accurate representation of the network programming idea. We created socket, populated server struct, connected to the server and simply written and read from the server.

Main GUI code

Main code for GUI application is in mainwindow.cpp and mainwindow.h files.

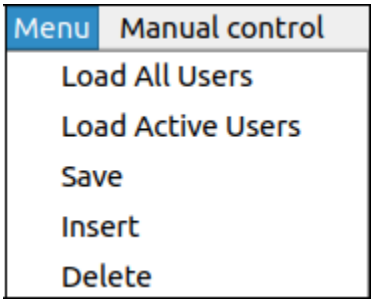


As it can be seen from the Figure to the right, GUI application consists out of QLabels, QLineEdits, QTableView and QMenus.

User has option to input data into QLineEdits and to manipulate that data from QMenu with options that are provided.

QTableView is there to provide clear representation of data form the system. User will have two options in QMenu to choose which data will be presented in this table.

GUI application provides user with many options for data manipulation or system control, and those options are accessible form two menus that application has: “Menu” and “Manual control”.



In the “Menu” menu user has five options that can be seen in the Figure on the left.

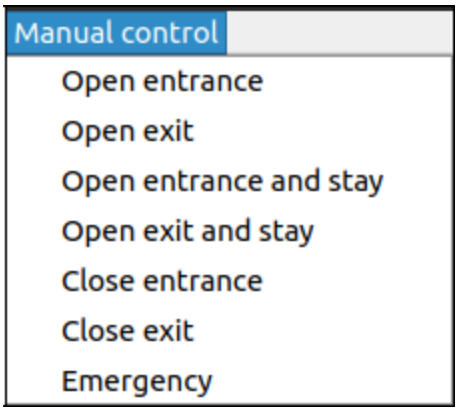
“Load All Users” option will call function that will try to connect to the server on the Raspberry Pi and ask for data of all users that have access to the parking. Data that is received from Raspberry Pi will be shown in QTableView table.

“Load Active Users” option will call function that will try to connect to the Raspberry Pi server and ask for data of users that are currently present on the parking. Again, data that is received from Raspberry Pi will be shown in QTableView table.

“Save” option will call function that will save all data from QTableView table to the file on the disk.

“Insert” option will call function that will take data form QLineEdits (input) and put it into QTableView table, and also send that data to the server on Raspberry Pi for it to add new data in database of all users of the parking. There are restrictions on the input from QLineEdits. In “ID” field only numbers can be entered. In “Name” and “Surname” fields only letters can be entered and there is maximum number of characters that can be entered.

“Delete” option will call function that will take data selected in QTableView table and delete it form the table, and also send that data to the Raspberry Pi server for it to remove that user form database of all users. If user is currently present on the parking it will raise flag for that user, and when he/she leaves the parking their data will be deleted.



In the "Manual control" menu user has seven options that can be seen in the Figure on the left.

"Open entrance" option will call function that will connect to the server and send header that will signal to the Raspberry Pi code that entrance ramp should be opened regularly (meaning that it should be closed automatically).

"Open exit" option will call function that will connect to the server and send header that will signal to the Raspberry Pi code that exit ramp should be opened regularly.

"Open entrance and stay" option will send header to the Raspberry Pi server that will signal that entrance ramp should be opened and left in that state. It will stay opened until manually closed. If it is already opened nothing will happen.

"Open exit and stay" option will also send header to the Raspberry Pi server that will signal that this time exit ramp should be opened and left in that state. It will stay opened until manually closed. If it is already opened nothing will happen.

"Close entrance" option will send header to the Raspberry Pi server that will signal that entrance ramp has to be closed. If it is already closed nothing will happen.

"Close exit" option will send header to the Raspberry Pi server that will signal that exit ramp has to be closed. If it is already closed nothing will happen.

"Emergency" option will send header to the Raspberry Pi that will signal that emergency situation is raised and both ramps should be raised. It will raise both ramps if they are not already opened and they will stay opened until closed manually.

In the code below it is possible to se how all widgets are placed to the main window of our GUI application. In the first part of the code we are creating menus, adding menu options and connecting them to the functions. Then we are adding widgets (line edits, labels and table view) to the window.

```
    auto menu = menuBar()->addMenu("Menu");
    menu->addAction("Load All Users", this, &Mainwindow::loadData);
    menu->addAction("Load Active Users", this,
&Mainwindow::loadActiveUsers);
    menu->addAction("Save", this, &Mainwindow::onSave);
    menu->addAction("Insert", this, &Mainwindow::onInsert);
    menu->addAction("Delete", this, &Mainwindow::onDelete);

    auto menu2 = menuBar()->addMenu("Manual control");
    menu2->addAction("Open entrance", this, &Mainwindow::openEntrance);
    menu2->addAction("Open exit", this, &Mainwindow::openExit);
    menu2->addAction("Open entrance and stay", this,
&Mainwindow::openEntranceAndStay);
    menu2->addAction("Open exit and stay", this,
&Mainwindow::openExitAndStay);
    menu2->addAction("Close entrance", this, &Mainwindow::closeEntrance);
    menu2->addAction("Close exit", this, &Mainwindow::closeExit);
    menu2->addAction("Emergency", this, &Mainwindow::emergency);
```

```
    _layA.addWidget(&_lblID, 0, 0);
    _layA.addWidget(&_id, 0, 1);

    _layA.addWidget(&_lblName, 1, 0);
    _layA.addWidget(&_name, 1, 1);
    _layA.addWidget(&_lblSurname, 2, 0);
    _layA.addWidget(&_surname, 2, 1);

    _layA.addWidget(&_view, 3, 0, 1, 4);


    _centralWidget->setLayout(&_layA);
    setCentralWidget(_centralWidget);
```

In the two functions below table view is connected with the function that puts the data into the table. Table view is connected with the function sung connect() function.

```
    void MainWindow::makeConnects()
    {
        connect(&_view, &QTableView::activated, this,
&MainWindow::putDataIntoWidgets);
    }

    void MainWindow::putDataIntoWidgets(const QModelIndex& index)
    {
        auto user = _data.at(index.row());
        _id.setText(user._id);
        _name.setText(user._name);
        _surname.setText(user._surname);
    }
```

ESP8266 Code

Code that controls ESP8266 microcontrollers is written in C++ programming language. It is developed and uploaded to controllers using PlatformIO IDE. ESP8266 controllers in this project are used to control Wiegand RFID reads and button pads.

For this purpose, code uses two very important libraries that provide all necessary functions. Those libraries are “Wiegand.h” and “ESP8266WiFi.h”. Second library is used for network configuration and communication, because ESP8266 communicates with Raspberry Pi over the network.

Reading RFID tags and button pad

ESP8266 microcontroller has one job, to read data form Wiegand RFID reader or button pad and to send that data to the Raspberry Pi. Reading RFID tags is done via getCode() function that gets ID of the RFID tag that was read.

```
void loop() {

  if(wg.available())
    {
      unsigned long input = wg.getCode();
      Serial.print(input);
      Serial.println();

      if (input>99)
      {
```

```
            String cardid = String(input, DEC);
            sendDataToServer(cardid);
          }
          else
          {
            if (input < 10)
            {
              addNode(&keypad, (int)input);
            }
            else if(input == 13)
            {
              sendDataToServer((String)convertToString());
                   for (int i = 0; i < 7; i++)
                         addNode(&keypad, 0);
            }
          }
        }
    }
```

In the code above we firstly got the code from the RFID reader, and then checked if code taken from RFID is from the scanned tag or from button pad. If it is 7-digit number its RFID tag scanned. If it is 2-digit number it is from button pad. Reading 7-digit PIN from button pad is done by using linked list. Every digit entered from button pad is placed in linked list. When there are 7 digits in the linked list, every other is added to the back and first is removed so there are always only 7 digits. When enter is pressed digits from linked list are converted to the one 7-digit integer and sent to the Raspberry Pi. Raspberry Pi is performing checks to prevent entering/leaving multiple times with the same PIN.

Now, the code below is the function sendDataToServer(String) that connects to our server on the Raspberry Pi and sends code from RFID/button pad to the Raspberry Pi.

```
void sendDataToServer(String data)
{

    WiFiClient client;

    if (!client.connect(host, port)) {

        Serial.println("Connection to host failed");

        delay(1000);
        return;
    }

    Serial.println("Connected to server successful!");

    String _data = "2\n" + data;

    client.print(_data);
    Serial.println(_data);
    Serial.println("Disconnecting...");
    client.stop();

    delay(3000);
}
```

Main Raspberry Pi Code

Main code on the Raspberry Pi is programmed completely in C programming language. This code is core of the project for many reasons. In this code main server for communication is programmed, control of the ramps and IR sensors is programmed here, database is on the Raspberry Pi and overall all data form other parts of the project are directed to this code.

There are two main parts of the code on Raspberry Pi. First is server code and second is hardware and data control. These two parts of the code run simultaneously on two separated threads.

In the main function of the main code on the Raspberry Pi that we can se below we firstly initialized some variables and called couple setup custom functions that will prepare server for running. After that we initialize GPIO pins of the Raspberry Pi. Then we initialize the mutex. *Mutual exclusion object (mutex) is a program object that allows multiple program threads to share the same resource, such as file access, but not simultaneously.*[12] After all this, we are creating new thread on which server will run simultaneously with the main thread/code. Under this, in the infinite while loop we are checking for new data/codes from the RFID readers.

```
int main (void)
{
	n = 0;
	isEntranceOpened = false;
	isExitOpened = false;
	pthread_t server_thread;
	createSocket();
	populateSrv();
	bindAddress();
	listenOnSocket();
	database._counter = 0;
	if (gpioInitialise() < 0)
		return -1;

	loadDatabaseFromFile();

	if (pthread_mutex_init(&lock, NULL) != 0) {
	  perror("Mutex init failed\n");
	  return 1;
    }

	if (pthread_create(&server_thread, NULL, &server, NULL) != 0) {
		perror("pthread_create");
		return 1;
	}

	while(1)
	{
		idCheck(&database);
	}

	pthread_join(server_thread, NULL);
	pthread_mutex_destroy(&lock);

	return 0;
}
```

[12] https://www.webopedia.com/TERM/M/mutex.html

The code below is function that checks for new data from RFID readers. It is checking global variables and user data. Global variables are set by server when it gets new data from ESP microcontrollers. “data” variable that is checked in the code below is one of global variables set by server and it contains ID number that is send to server by ESP microcontrollers, and when read once it is set to 0. Therefore, nothing will happen until new data comes through server. Other global variable in the question is “header” and it contains one digit which tells from which RFID reader ID is sent, from entrance or from the exit.

We can also se that for bringing the ramps up multithreading is used which will be explained latter on in the document. Usage of the linked list “_database” will also be explained latter on in the document.

```
void idCheck(list* _database)
{
node* current = _database->_first_node;

while(current)
{
	if (current->_id == data)
	{
		pthread_t temp;

		if(current->_status == 0 && header == 1)
		{
			parameter._motor = MOTOR_RIGHT;
			parameter.__node = current;
			pthread_create(&temp, NULL, &motorUp, (void
*)&parameter);
			pthread_detach(temp);
		}

		else if(current->_status == 1 && header == 2)
		{
			parameter._motor = MOTOR_LEFT;
			parameter.__node = current;
			pthread_create(&temp, NULL, &motorUp, (void
*)&parameter);
			pthread_detach(temp);
		}
		else if(current->_status == 2 && header == 2)
		{
			parameter._motor = MOTOR_LEFT;
			parameter.__node = current;
			pthread_create(&temp, NULL, &motorUp, (void
*)&parameter);
			pthread_detach(temp);
		}
	}

	current = current->_next;
}
data = 0;
header = 0;
}
```

Server code

Server on the Raspberry Pi is developed using C socket programming. It is designed to accept connections from ESP8266 microcontrollers and handle them separately, or to accept any other connections assuming they are from GUI application, and handling these connections separately. It runs on separate thread from the main code.

```
void* server() {

    cli_len = sizeof(cli);
    char temp[32];
    while(1) {

      if ((newfd = accept(fd, (struct sockaddr*) &cli, &cli_len)) == -1) {
                  perror("accept");
                  continue;
      }
       inet_ntop(AF_INET, &(cli.sin_addr), temp, INET_ADDRSTRLEN);
      readMessage();

      if ((cli.sin_addr.s_addr == inet_addr(MICROCONTROLLER1)) ||
(cli.sin_addr.s_addr == inet_addr(MICROCONTROLLER2))) {
            handleMicrocontrollers();
      }

      else {
            handleQtConnection();
      }

      memset(readBuffer, 0, READ_BUFFER);
      memset(writeBuffer, 0, WRITE_BUFFER);
      close(newfd);

      }
}
```

If server gets connection from ESP8266 microcontrollers it firstly reads header and stores it to global variable for other thread to be able to recognize which controller sent the data (microcontroller on the entrance or on the exit). Then it reads 7-digit integer and also stores it in global variable. These variables will be accessed by the main thread and data will be appropriately processed, but server code is just in charge of accepting connections and reading data from microcontrollers.

If server gets connection that is not from microcontrollers, that means that server will assume connection is from GUI application. Server will firstly read just a header to determine what to do next. For the options/commands from “Manual control” menu in GUI application, server just reads header and calls appropriate function to handle the command.

For options/commands form the “Menu” menu in GUI application server has more work to do. For options “Load All Users” and “Load Active Users” server calls functions that will prepare that data into the write buffer, and then it sends it back to the GUI application. For “Insert” and “Delete” server need to read additional data (user ID, Name and Surname) and then sends that data to appropriate function that will insert/delete user into/from database.

Hardware and data control

Main thread of the core application on the Raspberry Pi is in control of the hardware parts connected to the Raspberry Pi (servo motors and IR proximity sensors) and in control of the data and database.

Hardware control

Raspberry Pi is in control of the servo motors (entrance and exit ramps) and IR proximity sensors that are located under the ramps. For the hardware control on Raspberry Pi library “pigpio.h” is used.

Servo motors are controlled via function gpioServo(GPIO pin, step) from the library “pigpio.h”. This function is called in while loop that has limits at minimum and maximum range of the motors, and it is called with the particular step until step reaches the limit.

Function below is the best example of the control over the motors. There are multiple similar functions that control servo motors in different ways and situations, but this function is the best example. It returns void pointer and takes as argument void pointer because it is called on the separate thread from the thread where it is called. It is used with multithreading because it is kind of the function that blocks the execution flow of the program and waits for the user input. Multithreading is used not to block whole system when waiting on user input.

```
void* motorUp(void* str)
{
	motorUpParam* __parameter;
	int motor;
	node* _node;
	__parameter = (motorUpParam *) str;
	motor = __parameter->_motor;
	_node = __parameter->__node;

	if (motor == MOTOR_RIGHT)
	{
		pthread_mutex_lock(&lock);
		if (isEntranceOpened) {
			pthread_mutex_unlock(&lock);
			return NULL;
		}
		else
			isEntranceOpened = true;
		pthread_mutex_unlock(&lock);
	}
	else
	{
		pthread_mutex_lock(&lock);
		if (isExitOpened) {
			pthread_mutex_unlock(&lock);
			return NULL;
		}
		else
			isExitOpened = true;
		pthread_mutex_unlock(&lock);
	}
	int width = MAX_WIDTH;
	while (width > MIN_WIDTH)
	{
	gpioServo(motor, width);
	width -= STEP;
```

```
        time_sleep(0.1);
        }

        if (motor == MOTOR_LEFT)
                irControlOfTheMotor(SENSOR_LEFT, motor, _node);
        else
                irControlOfTheMotor(SENSOR_RIGHT, motor, _node);

        return NULL;
}
```

Function irControlOfTheMotor(int, int, node*) that is called at the end of the code above is in control of the IR sensors. It just reads the output that sensors give and wait for the car to pass. If it does not pass ramp will close automatically after 8 seconds. While sensor is activated by car ramp will not go down.

```
void irControlOfTheMotor(int ir_pin, int motor, node* _node)
{
    int i;
    int check = 0;
    bool didCarPass = false;

    for (i = 0; i <= 10; i++)
    {
      if (gpioRead(ir_pin) == check)
      {
            didCarPass = true;
            break;
      }
      time_sleep(1);
      if (i == 8)
      {
          motorDown(motor);
          return;
      }
    }

    while(gpioRead(ir_pin) == check)
    {
        time_sleep(0.3);
    }

    if (didCarPass)
    {
            pthread_mutex_lock(&lock);
            if (_node->_status == 1)
                  _node->_status = 0;
            else if(_node->_status == 0)
                  _node->_status = 1;
            else
                  deleteNode(_node->_id);

            pthread_mutex_unlock(&lock);
    }
    time_sleep(1);
    motorDown(motor);
}
```

From IR proximity sensors only input is taken. Input from this sensors is digital high/low input, and it is read via gpioRead() function form the library mentioned above. If sensors are outputting high signal there is nothing in front of them, but if they are outputting low signal there is something in front of them.

Hardware control and multithreading

Functions that are in control of the motors are function that block program execution flow, because ramps have to wait for the car to pass. Therefore, when such function is called rest of the program is frozen. Also meaning that only one ramp can be controlled at the time.

Because of these reasons, functions that control motors are called on the separate thread. When function needs to be called, new thread is created for that function and that thread is detached from the main thread. Thread is detached because there is no need for data transmission or additional communication between main and newly created thread. New thread just needs to bring up the ramp, wait for the user to pass (or wait for timeout) and to bring the ramp down. Then function returns NULL and thread is killed.

```
parameter._motor = MOTOR_RIGHT;
parameter.__node = current;
pthread_create(&temp, NULL, &motorUp, (void *)&parameter);
pthread_detach(temp);
```

Struct “parameter” is used to pass multiple arguments to the function called on the new thread.

Data control

System that was developed for this project has a database that has information about all users that have access to the parking. That database is a simple file “all_users.DAT” and it contains ID, name and surname of all users that have access to the parking. This database can be updated by user of the GUI application via “Insert” and “Delete” commands. Information about all users that have access to the parking and about their status (are they currently present on the parking) is data that is used a lot. Therefore, in the main code on Raspberry Pi there is linked list created that contains all data as the database with additional information (status of the user). When database is updated this linked list is also updated. But when program needs to check status of the user or whether user has access to the parking it does not need to open and search file but it just searches the linked list which is much faster process.

```
typedef struct __node {
        int _id;
        char _name[20];
        char _surname[25];
        short int _status;
        struct __node* _next;
} node;

typedef struct __linked_list {
        node* _first_node;
        int _counter;
} list;
```

Above code is just declaration of two structs needed for our linked list. And in the code below we will se how to add new data/node to the linked list.

```
void addNode(list* _database, int id, char* name, char* surname)
{

	if (database._counter == 0)
	{
		node* newNode = (node*) malloc(sizeof(node));
		newNode->_id = id;
		strcpy(newNode->_name, name);
		strcpy(newNode->_surname, surname);
		newNode->_status = 0;
		newNode->_next = 0;
		_database->_first_node = newNode;
		printf("\n");
		_database->_counter += 1;
		return;
	}
	else
	{
		node* current = _database->_first_node;

		while(current)
		{
			if (current->_id == id)
				return;

			if (!current->_next)
			{

				node* newNode = (node*) malloc(sizeof(node));
				newNode->_id = id;
				strcpy(newNode->_name, name);
				strcpy(newNode->_surname, surname);
				newNode->_status = 0;
				newNode->_next = 0;
				current->_next = newNode;
				printf("\n");
				_database->_counter += 1;
				return;
			}
			else
			{
				current = current->_next;
			}
		}
	}
}
```

With this kind of data structure it is much faster to search data than accessing file database every time we need to check something. Linked list also simplifies adding and deleting nodes/data from it.

Database is there to permanently store data and linked list is there to speed up the program. Every time program on Raspberry Pi is ran, at the start of the program linked list is filled with information from the database.

References

https://www.raspberrypi.org/help/what-%20is-a-raspberry-pi/

https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/

https://www.techopedia.com/definition/13274/servo-motor

https://www.jameco.com/jameco/workshop/howitworks/how-servo-motors-work.html

https://robu.in/product/towerpro-sg90-9gm-1-2kg-180-degree-rotation-servo-motor-good-

quality/ https://www.makerlab-electronics.com/product/ir-proximity-sensor/

https://www.techopedia.com/definition/3641/microcontroller

https://medium.com/@rxseger/esp8266-first-project-home-automation-with-relays-switches-pwm-and-an-adc-ad25f317c74f

https://github.com/monkeyboard/Wiegand-Protocol-Library-for-Arduino

https://eworkaccesscontrol.en.made-in-china.com/product/osPmfiIAvxhW/China-Access-Control-Keypad-RFID-Reader-Smart-Card-Reader-Access-Control-System.html

https://www.webopedia.com/TERM/M/mutex.html