Hindawi
Mobile Information Systems
Volume 2020, Article ID 8875301, 15 pages
https://doi.org/10.1155/2020/8875301

Research Article

A Smart Parking System Based on Mini PC Platform and Mobile Application for Parking Space Detection

Vladimir Sobeslav and Josef Horalek

Department of Information Technologies, Faculty of Informatics and Management, University of Hradec Kralove, Rokitanskeho 62, Hradec Kralove 500 01, Czech Republic

Correspondence should be addressed to Vladimir Sobeslav; vladimir.sobeslav@uhk.cz

Received 24 June 2020; Revised 25 September 2020; Accepted 9 October 2020; Published 26 October 2020

Academic Editor: Peter Brida

Car parking is a major problem in urban areas in developed and also in developing countries. The growing number of vehicles creates a problem with parking spaces mainly in the city center and the surrounding streets. The local authorities have to react with regulations, and the current situation is unpleasant for many citizens. Therefore, the aim of this article is to propose a complex outdoor smart parking lot system based on the mini PC platform with the pilot implementation, which would provide a solution for the aforementioned problem. Current outdoor car park management is dependent on human personnel keeping track of the available parking lots or a sensor-based system that monitors the availability of each car. The proposed solution utilizes a modern IoT approach and technologies such as mini PC platform, sensors, and IQRF. When compared to a specialized and expensive system, it is a solution that is cost-effective and has the potential in its expansion and integration with other IoT services.

1. Introduction

The number of vehicles is constantly increasing, not only in the Czech Republic but also in other countries. According to the data from Central Auto-moto Club of the Czech Republic, more than 5.5 million cars are registered here, which means that their number has increased 2.4 times since 1989. With the increasing number of the vehicles, the problems with the parking also arise. First and foremost, these problems become prevalent during sport and cultural events, as well as in the proximity of administrative buildings or banks. Parking in the towns and cities during the traffic peak times also poses a significant problem. A lack of knowledge of current number of parking spaces can lead the drivers to fully occupied parking lots, which consequently leads to having to move to another location, along with searching for a spot on another parking lot. Therefore, not only do the drivers waste their time, but also fuel, and it all causes the deterioration of the traffic situation and negatively affects the environment.

The aim of this article is to propose a complex smart parking lot system based on the mini PC platform, along with its pilot implementation, which would provide a solution for the aforementioned problem. The solution aims to use the newest principles in the IoT area, mesh networks, and tools offered by the Android OS as for the whole solution to be affordable and quickly deployable.

The proposed solution contains a complete design and realization based on the mini PC platform, IQRF technology, including use of the DPA protocol. As a gateway, UpBoard with DK-EVAL-04A communication module has been used. Before the solution itself, an in-depth analysis of the currently used solutions and approaches has been performed. The findings of this analysis are provided below.

2. Related Works

Firstly, an analysis of the approach to the smart parking solution must be performed. The article IoT-Based Smart Parking System [1] is a prominent input in this area, which is similarly to [2] and describes the architecture of a smart parking system based on the IoT (Internet of Things) technology. In [1], sensors placed on the parking spaces detect the proximity of the vehicle and send these data to

cloud using mini PC (Raspberry Pi) deployed on the parking lot. The authors describe the use of Raspberry Pi GPIO pins, to which 26 sensors can be connected. The number can be increased even further using a suitable multiplexor. This solution utilizes an IBM MQTT server, to which the Raspberry sends data from the sensors via MQTT messages. The user communicates with the system via a mobile app written in Apache Cordova, which communicates with the server with messages in the JSON format. The user can use the mobile app to see the number of the free spots on the parking lot, make a reservation of a parking space, and pay the parking fee. In [2], besides the general architecture, the authors focus on using Elliptic Curve Cryptography (ECC) as an attractive alternative to the conventional public key cryptography such as RSA. Interesting algorithms employed in the contemporary smart parking systems are presented in [3]. Finally, an overall overview of the approaches and solutions is provided by [4].

Another topic covered by this article is the optimization of the logic and IoT approaches used to find and navigate to the parking space itself. Tsai et al.[5] describe the use of a mobile app and Internet of Things (IoT) technology in the parking system: the process of finding a parking lot, parking space reservation, and indoor navigation inside the parking lot. Another topic is finding the suitable algorithm to calculate the priority for recommending a specific parking lot based on the distance of the driver from the parking lot, number of free parking spaces, and the parking fee, with the preferences being set up by the user. After selecting the parking lot, the driver has an opportunity to book the parking space via the app. For the calculation of the distance from the parking lot and the navigation, the app uses GPS coordinates acquired from the Google API. Detection of the presence of the vehicle on the parking space is realized via an ultrasound sensor and transferred to the server via a WiFi module. The described solution is designated mainly for indoor parking lots, with the indoor navigation provided by iBeacon Bluetooth transmitters. The presence of the driver on the parking lot is detected by an RFID chip. The matter of parking specifically on the side of the roads is discussed in [6]. It describes the detection of the roadside parking spaces using sensors placed on the vehicle on the passenger's side. These sensors are placed on public transport vehicles, taxis, or sometimes on the vehicles of the volunteers who frequently pass through the marked measured areas.

The system uses an ultrasound detector to detect the parked vehicles and empty spots along the road in conjunction with the GPS system, and by using Map Matching (which compares the detected spots with the map), a map of available parking spaces is continuously generated and is distributed to the users via the mobile app or a website. According to the performed measuring, to map the same number of the parking spaces, mobile sensors are more efficient than sensors placed directly on the parking spaces. The success rate of the detection was between 76 and 94 percent, depending on the accuracy of the GPS system.

In the hereby presented complex solution, IQRF networks are utilized. Their communication system is discussed in [7] and the spread of the networks' usage is discussed in [8, 9].

3. The Proposed System Architecture

The aim of the proposed parking system is a complex and economic system for detection of free parking spaces on an outdoor parking lot, which is based on the mini PC platform. The system partially handles the problems with undisciplined drivers who would use the parking lot without the respective authorization. This is provided by using the automatic gate system at the parking lot entrance. Because of the budget requirements, the variant where the parked vehicles would be detected via ultrasound or infrasensors has been abandoned, and therefore, the proposed solution discusses the variant using magnetometric sensors connected to a mesh network. In regular intervals, the mini PC checks whether the individual parking spaces are occupied and stores the data in a real-time database. The driver is then informed via the mobile app for Android, and so they have current information about the numbers of available parking spaces on the selected parking lot. The availability can be checked either manually or automatically using the Geofencing service. The user is also able to book a parking space for 60 minutes. If they do not arrive at the parking lot during this time, the reservation is automatically canceled.

The proposed solution enables opening the parking lot gate directly from the mobile app by adding an entry into the real-time database. After the entry is added, the mini PC performs the corresponding steps. To check the presence of the driver (their smartphone, to be precise) on the parking lot, the mobile app compares the GPS coordinates before registering the vehicle as present on the parking lot. If the distance between the smartphone and the parking lot is higher than a given limit, the access is denied. Thanks to this solution, there is no necessity for use of an RFID chip, and to make use of the parking lot, a smartphone with the installed app and connection to the Internet should suffice. The identification of the smartphone is performed using a unique 64-bit number, ANDROID ID. The general model of the proposed solution is depicted in Figure 1.

3.1. Functional Diagram of the Proposed Solution. The mobile app monitors the entry in the database with the number of empty spots. On every change, the state of the notification icon on the app's main screen is changed. After tapping this icon, the current occupancy of the individual parking spaces is loaded from the database, and it is displayed on the smartphone screen. By tapping on the respective icon on the main screen, navigation to the parking lot is started. Using Google API, the application starts navigation on Google Maps. More detailed information is described in Section 5. To enable the automatic notifications about parking lots in proximity and the number of available parking spaces, the application uses Geofencing service, which is described in Section 5.2.

The following diagrams describe two main functions:

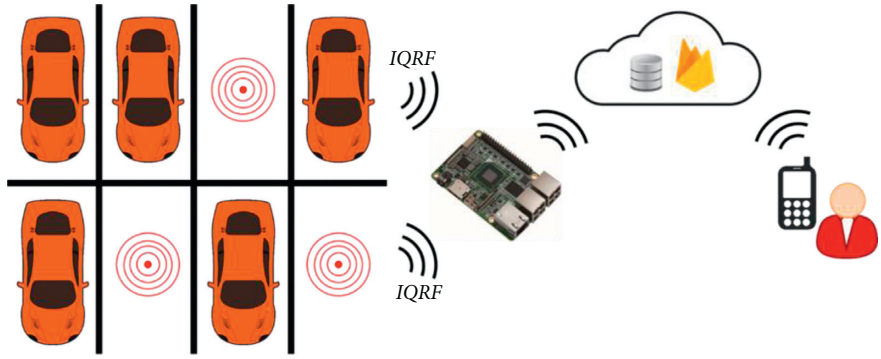

Figure 1: General system structure design.

Displaying the available spots at the parking lot and their current occupancy (Figure 2)

Turning on the automatic notification about available parking lot (Figure 3)

The following diagram (Figure 4) describes the logical steps required for opening the parking lot gate. First, the application checks the database of the parked vehicles to determine whether the vehicle is entering or leaving the parking lot.

When entering the parking lot, it is also checked whether there are available spots or if the driver who is sending a request to open the gate has a reservation saved in the database. If everything is in order, the distance from the parking lot is checked to prevent an unintentional or deliberate gate closure at higher distance from the parking lot. If any condition is not met, the driver receives information about the parking lot unavailability. Otherwise, the gate opens, and the arriving vehicle is added to the parking lot's database. At the same time, the driver's reservation validity is checked and deleted from the database if invalid. When leaving the parking lot, the system opens the exit gate and deletes the record of the vehicle from the parking lot's database.

The realization of the reservation process is depicted in Figure 5. During an attempt at making a reservation, the system first checks whether the vehicle is not currently located at the parking lot and that its ID does not have a reservation created already. Then, the available spots are checked. If all the conditions are met, a reservation entry is made in the parking lot's database. Otherwise, the app informs the driver that the reservation cannot be made. Duration of the reservation is limited to 60 minutes. For this reason, in 1-minute intervals, the mini PC on the parking lot checks the database if any reservation has exceeded this limit. If this occurs, the entry is deleted from the database.

On the general communication schema (Figure 6), you can see that the common storage for the whole system is a Firebase database. The communication between the database and the mini PC is procured via TCP/IP protocol. The communication between the database and the mobile app uses mobile network data transfers and the communication between the mini PC and the individual sensors is procured by the IQRF technology described in Chapter 3.

4. IQRF Communication Platform

From the principle of the intelligent parking system proposal, it is necessary for the system itself to be able to detect whether the respective physical spot is occupied or not, and it must also be connected to the suitable network so it can communicate with the central mini PC. For this purpose, IQRF technology has been chosen [10].

IQRF is a platform suitable for wireless data transfer, which utilizes wireless data transfer, using the frequencies of 868 MHz and 433 MHz for the communication. The IQRF platform makes use of special transceivers that reciprocally interchange the data. The IQRF transceivers are known to have a very low consumption rate (12.3 mA during communication and 380 nA in sleep mode), and thanks to the supported MESH topology, communication at relatively long distance is possible. These parameters appear to be an ideal solution for the IoT technologies. The wireless IQRF network uses the IQMEST protocol [11, 12], which uses the principle that in any given area, and there are always at least two IQRF transceivers within the transmission reach. The maximum distance between two communicating transceivers is around 500 meters in space without obstacles. The transceivers transmit in synchronization, so they do not interrupt each other during the transmission. The communication is governed by a coordinator, which sends the data, transceivers in the reach receive the data, during their time-slot send the data further, and this way the data gradually spread through the whole network. Thanks to this principle, the whole network is very reliable and has a high success rate of the data transmission. Because of the duplicate paths between individual transceivers, the data reach their destination even if multiple communication paths are disrupted at the same time. The principle is depicted in Figure 7, where the coordinator C sends the data for the transceiver N2. The communication paths that are interrupted, e.g., by signal disruption, are represented by the red crosses. The data reach their destination, as the green arrows suggest, via the transceivers N1, N3, and N4.

4.1. DPA Protocol. Every transceiver has a hardware profile (HWP). Assigning a HWP to the transceivers enables their control via messages with DPA (direct peripheral access) protocol [13]. Because of this protocol, it is possible to build

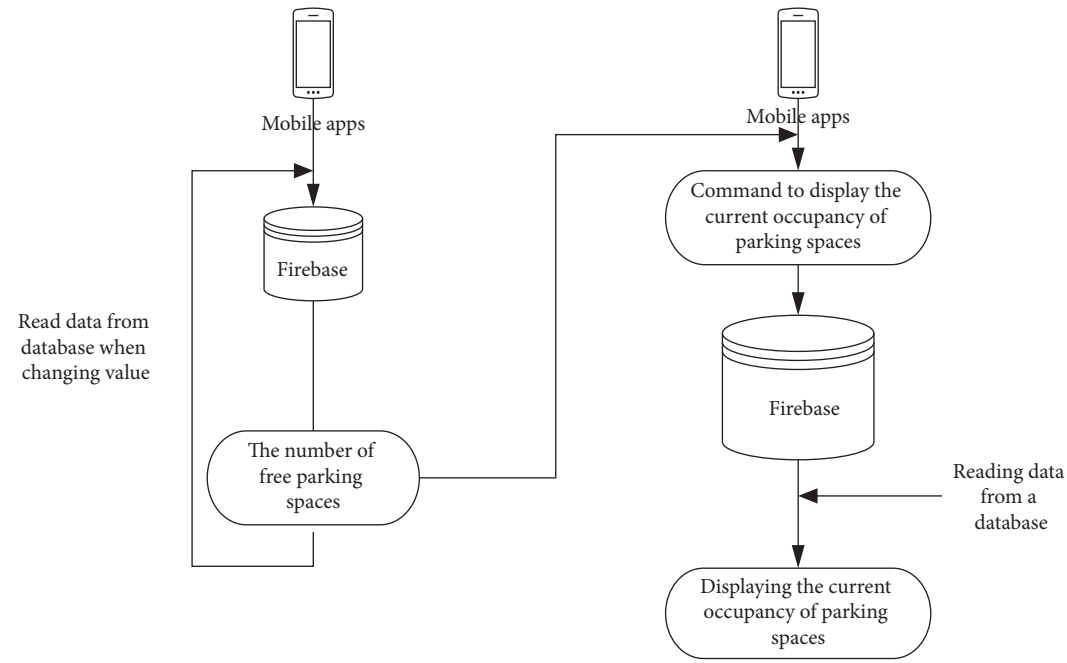

Figure 2: Functional diagram to show the number of available parking spaces and their current occupancy.

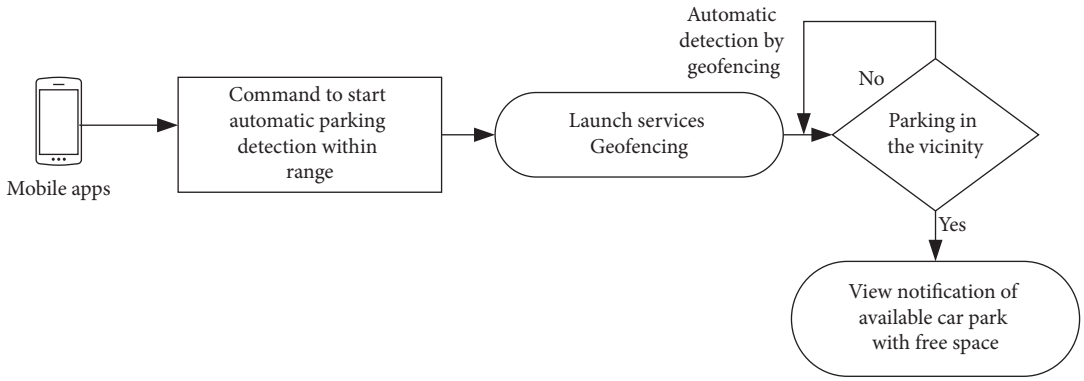

Figure 3: Functional diagram to trigger the automatic notification of available car parking.

a network comprised of up to 240 transceivers, and the transceivers can be controlled by sending the data in the specific format (Table 1).

NADR (node address) has the address 0x00 for the coordinator and 0x01–0xEF for the other transceivers and PNUM (peripheral number) 0x03 EEPROM, 0x08 SPI, 0x0C UART, etc. PCMD (peripheral command) is designated only by the type of the used periphery; HWPID (hardware profile ID) uniquely determines the functionality of the periphery device. If 0xFFFF number is used, the command is executed on any HW profile. PDATA (peripheral data) is an optional 56-byte field for additional command parameters.

4.2. Custom DPA Handler. For creating the own logic of the transceiver, custom DPA handler is used, i.e., it uses code written in C, which can be used to define custom user periphery and set up its behavior during received a DPA command with this periphery's ID and the ID of the respective command. Using the DPA handler, it is also possible to expand the set of the HW groups described above, and thus it enables the filtering of control message for the groups of peripheries of the respective type.

4.3. FRC. Fast response command (FRC) is a special coordinator DPA periphery, which enables sending a command that can be processed by all the transceivers in the network. The moment the transceiver processes the command, it stores the response on the specific position in the message and it passes through the whole network along with the data and collects the responses of individual transceivers. If we need to get the same information from all the transceivers, which is, in our case, the input from a detector about the presence of a vehicle on a parking space, the use of FRC is very practical as it is not necessary to send individually to every transceiver, but sending one command is enough. This positively affects not only the amount of the

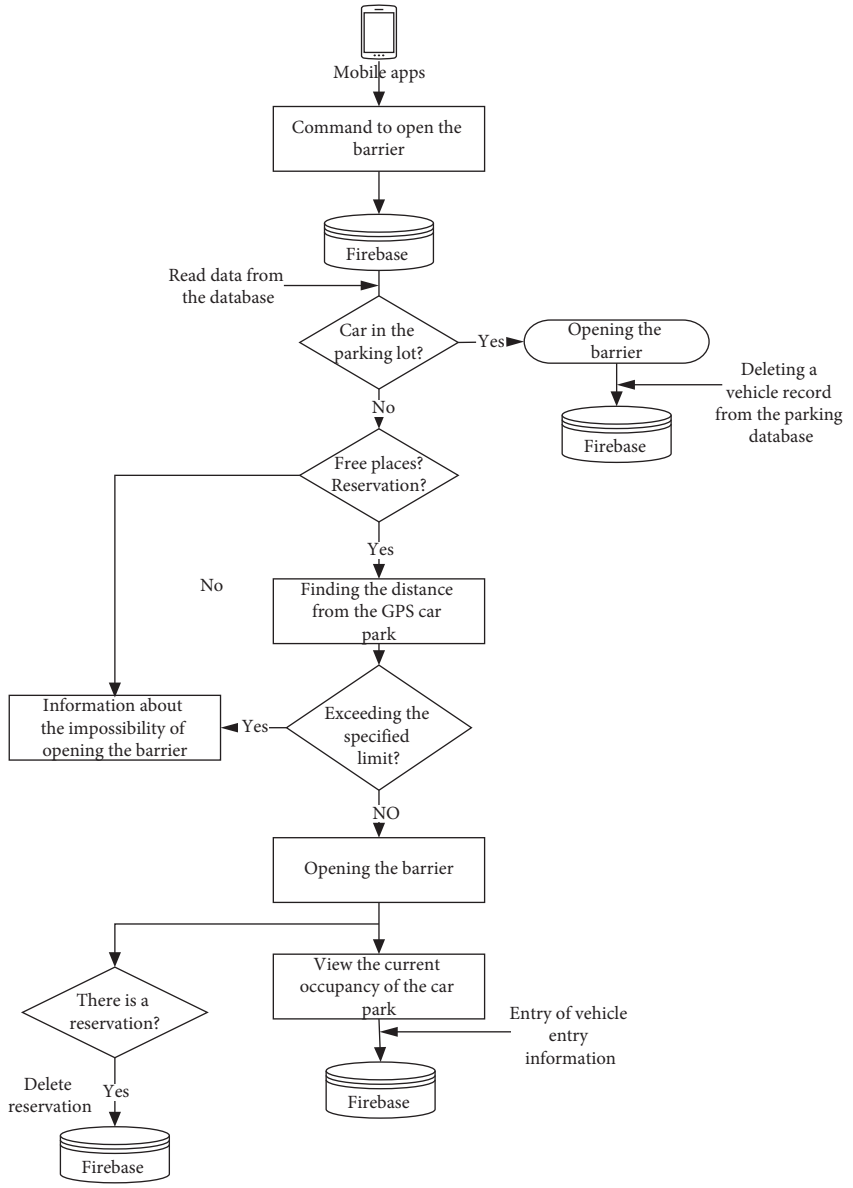

Figure 4: Functional diagram of the entry and exit gate control.

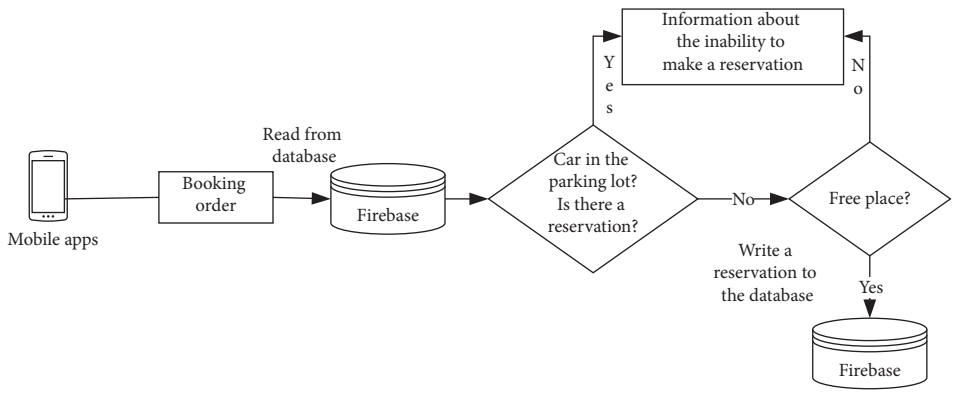

Figure 5: Functional diagram for making a reservation.

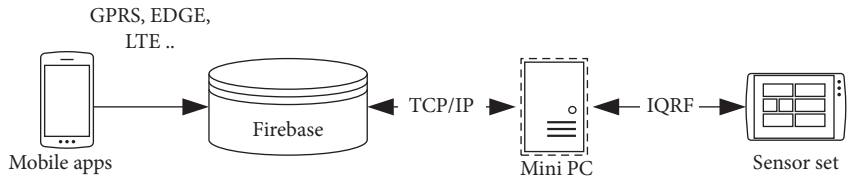

Figure 6: Functional diagram to communication.

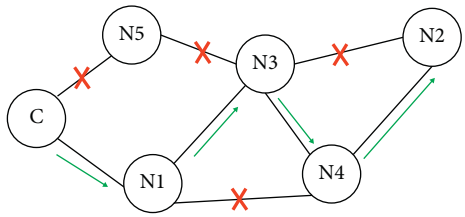

Figure 7: Principle of IQRF communication between coordinator C and N2 transceiver.

transferred data but also time needed to get a response. The FRC command is sent via the DPA protocol (Table 2).

User data item is not required by all the FRC commands, and if not used, it must be replaced by a 2-byte value of 00.00. SEND command can be replaced by the SEND selective (0x02) command, which makes it possible to send the command only to the chosen transceivers, which are defined in the data field between ID and user data entries.

The selected transceivers can be specified through 30-byte binary information. In IQRF network, there can be the maximum of 240 transceivers, which corresponds to 30 bytes, i.e., 240 bits. If the respective bit has a value of 1, the command will be sent to the corresponding transceiver. Otherwise (the value of 0), the transceiver will not be included into processing of the command.

For the initial configuration of the transceivers and for the creation of the IQRF network, it is necessary to use CK–USB–04K programmer and IQRF IDE development interface. It is also necessary to get hardware profiles for the coordinator and the individual nodes.

Into the newly created project, the author must add the HW profiles and the DPA custom handler for FRC command for detecting the state of the sensors on the parking spaces. The HW profiles belong to plug-ins section and the DPA custom handler to source section. Considering the fact that the inserted DPA handler is in the format of the source file written in C, this file must be compiled to * hex format so it can be uploaded to the transceiver. After inserting the first transceiver into the programmer and connecting it to a USB port of a PC, in the "Project" window in the "TR Configuration" section, the configuration settings for the inserted transceiver can be opened. For all the communicating transceivers, it is necessary to set the same communication channel (typically 52) in the "OS" tab. Next, in the "HWP" section, the option to process the FRC commands for the coordinator and use of custom DPA handler for every node must be enabled. In the "Security" tab, a password and communication encryption can be set up (Figure 8).

After programming all the transceivers and inserting them into the DK-EVAL-04A testing modules or alternatively by connecting custom modules that would need to be adjusted according to the transceiver connection schema (Figure 9), for the transceiver to be powered by the right voltage and have accessible RESET and bonding buttons, by connecting the coordinator into the CK–USB–04K programmer, an IQRF network can be created.

If a red LED is blinking after connecting the power source, it means that no previous bonding is stored in the memory. Otherwise, it is necessary to manually unbond by pressing the reset and the user buttons on the testing module and then releasing the reset button. After a green LED blinks, the user button must be released at once. For erasing the data about bonding from the coordinator, IQMESH Network Manager, which is a part of the IQRF IDE program, must be used. To do so, press the "Clear All Bonds" button. Afterwards, by pressing the "Bond" button, the coordinator start searching for a new node, and in the frame of ten seconds, the bonding must be confirmed by pressing the user button on the respective testing module. By repeating this procedure, all the nodes must be bonded with the coordinator. At the time of the bonding, all the nodes must be in the communication reach of the coordinator. After the bonding is finished and the respective nodes are placed on their final positions, by clicking the "Discovery" button in IQMESH Network Manager, the IQMESH network topology is created. It can be viewed in the "MapView" tab (Figure 10).

4.4. IQRF Gateway. The created IQRF network requires a gateway for transmitting the data to the database/cloud via the Internet. For the purposes of the best compatibility and technical support, a one-board computer UpBoard has been used as it, in comparison to Raspberry Pi, offers a microprocessor with better performance in Intel Atom, as well as higher memory capacity, and does not require an OS to be installed on an external memory card, which can pose potential problems in conjunction with the mechanical connector. UpBoard is also used in Intel® RealSense™ Robotic Development Kit and has the following characteristics:

Intel® Atom™ x5-Z8350 SoC
Onboard DDR3L Memory up to 4 GB
Onboard eMMC Storage up to 64 GB
Gigabit LAN × 1, USB 2.0 × 4, USB 3.0 × 1, HDMI × 1
5V DC-in
40 pin GPIO × 1
DSI/eDP × 1
MIPI-CSI × 1

Table 1: DPA packet structure.

NADR (node address)	PNUM (peripheral number)	PCMD (peripheral command)	HWPID (hardware profile ID)	PDATA (peripheral data)
[2B]	[1B]	[1B]	[2B]	[0–56B]

Table 2: FRC (SEND) command structure.

NADR (node address)	PNUM (peripheral number)	PCMD (peripheral command)	HWPID (hardware profile ID)	PDATA (peripheral data)	
0x00	0x0D	0x00	0xFFFF	ID	User data
Coordinator	FRC	SEND	All groups	Command ID	Data for FRC processing

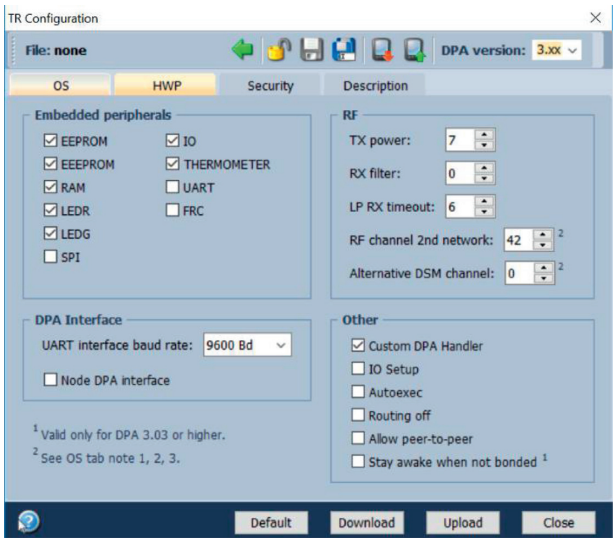

Figure 8: Example of HWP setting for "NOD" transceiver.

Considering the availability of the technical support, Ubilinux OS has been chosen for the smart parking system. The chosen mini PC serves for controlling the IQRF network and for conjoining the whole app via the Internet and the database. For those reasons, the IQRF coordinator had to be connected using an available reduction with UpBoard's GPIO pins, and corresponding utilities for the communication with the IQRF network and the database had to be installed as well. For interchanging the information between the UpBoard and the IQRF network, an MQTT Broker has been used. For the administration, we have chosen a web app for IQRF Daemon and NodeRED environment that will serve for programming the IQRF network maintenance. For the IQRF gateway we have used the following software setup. As an MQTT broker, we have decided to use the mosquito and mosquito-clients packages. Furthermore, the dirmngr server was used for certificate management. The IQRF Gateway Daemon package was used as an open-source IQRF gateway solution which is being widely used in this area, and it is supported also by Raspberry, Belagone, traditional PC and others. Node.js open-source server platform was used for code execution; for programming of IQRF network jobs, a database management, a NodeRED, was utilized.

4.5. Android Geofencing. The last component used in the presented solution is the Android Geofencing service, which serves to alert the driver about the availability of a parking lot within the mobile application. This service uses the capabilities of the mobile phone to determine its current location using GPS, the availability of known WIFI networks, and the distance from the mobile operator's BTS based on the signal strength of the mobile network. To use the service, firstly, it is needed to enter the latitude and longitude of a specific place (parking lot) and the radius of the circle centered in this coordinate. The circle created in this way is called geofence, and then it serves for detecting notifications or other actions such as turning on Bluetooth and turning off the phone's ringtone. The notification can be set for a situation when the phone enters a defined circle (we use this notification in the presented solution), or if it leaves the circle or stays in it for a certain period of time. Each Android user can have up to 100 geofences (circles) registered in all applications on their phone. If the phone is located at the intersection of several geofences, it can perform actions separately for each of them, or a notification of the availability of multiple geofences (parking lots) and the distance to the center of each can be sent and calculated from the difference of two GPS coordinates (phone and geofence).

5. Implementation of Key System Functions

In Section 5.1, the solutions described above, including the used IQRF network and NodeRED and the main component of the Android app are documented in detail.

5.1. Vehicle Detection. Vehicle detection is realized via MPU-9250 magnetometric sensor, which is connected via I2C bus with an Arduino Mini microprocessor unit (Figure 11).

For the communication with the sensor, MPU9250_asukiaaa.h has been used as it enables a simple maintenance of the magnetometric sensor. After the initial establishment, the measured values are stored for a reference and then

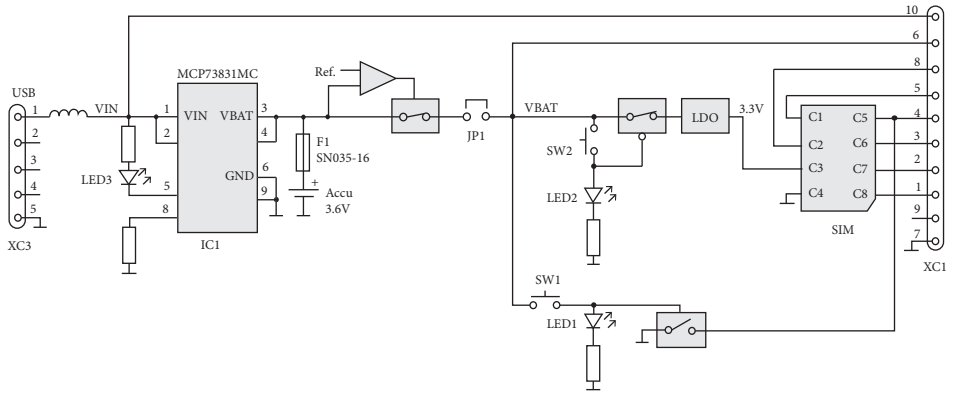

Figure 9: DK-EVAL-04A test module connection.

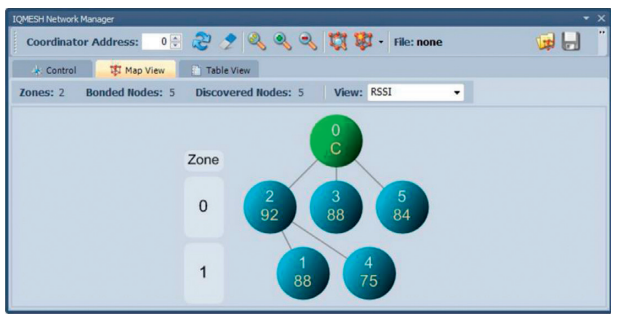

Figure 10: Example of network topology in IQMESH Network Manager.

compared with the currently measures values in the 500-millisecond intervals (during the initialization, the parking space must be empty). The measured values are transferred to the serial port for the purpose of checking the functionality of the sensor and checking the measured values. See the following code for the main program enabling the communication with the magnetometric sensor:

```
void loop() {
  if(start){delay(5000);}
  mySensor.magUpdate();
  mX = mySensor.magX();
  mY = mySensor.magY();
  mZ = mySensor.magZ();
  if(start){initSensor(); start = false;}
  Serial.println("magX: " + String(mX));
  Serial.println("maxY: " + String(mY));
  Serial.println("magZ: " + String(mZ));
  testSensor();
  Serial.println(""); // Add an empty line
  delay(500);
}
```

If a metal object (a vehicle) is detected in the proximity and the values in any of the axes exceed the given limit, the digital output signalizes the presence of a vehicle. After every

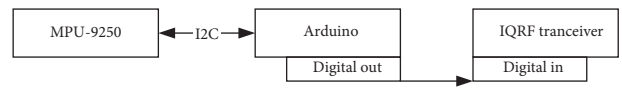

Figure 11: Block diagram of vehicle detector.

vehicle detection and its subsequent departure from the parking space, new rest values are saved as referential in order to reduce the amount of unprompted or erroneous detections caused by long-term changes of the magnetic field intensity. The implementation of the vehicle detection through the change of the magnetic field is described below.

```
void testSensor(){
  deltaMx = abs(normalMx-mX);
  deltaMy = abs(normalMy-mY);
  deltaMz = abs(normalMz-mZ);
  Serial.println("deltaMx: " + String(deltaMx));
  Serial.println("deltaMy: " + String(deltaMy));
  Serial.println("deltaMz: " + String(deltaMz));
  if(deltaMx > 7 || deltaMy > 7 || deltaMz > 7){
    Serial.println("Car is present");
    digitalWrite(13, HIGH);
    detect = true;
  }
  else
{Serial.println("Car is not present");
    digitalWrite(13, LOW);
    if(detect){start = true; detect = false;}
  }
}
```

The digital output of the Arduino module is connected to the IQRF transceiver, which provides the connection of the individual sensors into the IQMESH network.

5.2. Car Parking with NodeRED. As it has been already mentioned, the database communicates not only on the

mobile app but also the mini PC via NodeRED. For greater clarity, individual program flows are split into individual IQRF_Request tabs. The commands are sent to the IQRF network, IQRF_Response processes the responses from the IQRF network, Firebase communicates with the Firebase database, Gate operates the entrance and exit gate, and Reservations maintains the user reservations.

5.3. IQRF_Request. The program flow IQRF_Request (Figure 12) uses program nodes on the lies 2 through 5, which send the requests to open or close the entrance or exit gate. The requests come from other tabs via the connections (grey arrows at the beginning of the lines).

Individual requests are realized via custom functions written in JavaScript. That is the DPA command for the IQRF network (code: create a request for an IQRF network). Because of the physical absence of a gate, for the tuning purposes of the app, the commands were simulated by turn on or off a red LED on the IQRF coordinator (entry gate) and a green LED (exit gate). In practice, the LED would be replaced by a digital output that would send the command to the gate.

```
var data = {
    type: "raw",
    request: {
        nadr: "0x0000",
        pnum: "0x06",
        pcmd: "0x01",
        hwpid: "0xFFFF",
        pdata: "",
    },
    timeout: 1000
}
msg.payload = data;
return msg;
```

5.4. Gate. For the communication itself and its procurement, the gate is a key component of the whole architecture. The gate process flow is split into two parts. The first part controls the entry gate, and the other controls the exit gate (Figure 13).

The first flow begins with the Gate node, which is, in fact, a listener that watches the Gate item in the database. It follows with the switch on the value of 1 (opens the gate) and continues with the second path to the GateEntranceOpen node. The next node is a time delay for the vehicle to pass (sensors for the detection of a vehicle in the gate space are its part) and then the flow continues with the link to GateEntranceClose and sets the value of the gate item in the database to 0. In that case, the switch ensures that the gate does not open again after the value in the database is changed when the value of 0 is added, and so it takes the path 1, to which no other program node is connected. After opening the entry gate, the ID of the smartphone that asked for the gate to open is stored in reservationID item. This ID is then compared to the records in reservations, and if a valid reservation is found, it is deleted. After the time delay, the reservationID item is reset by ResetReservID.

The second flow starts with the GateID node, which continuously checks the database for changes in the gateID item. Into this item, the mobile app saves the ID of the smartphone registered on the parking lot (it has a record in carInPark) and sends a request to open the exit gate. The program continues with the link to IQRF_Request, where it opens the exit gate via the GateExitOpen node. After a time delay for the vehicle to pass through, the gate is closed again via the link to GateExitClose. The flow continues with loading the CarInPark items from the database and coverts them to the JSON format. These data are passed to the FirebaseConvert function (code: finding and passing the item ID for deletion), which, from the smartphone ID stored in GlobalContext, searches for a record in the database and deletes the record via the DeleteValue node. The last step of the program is the gateID item reset.

5.5. Reservations. In the case of this flow, a division into two parts has been done too (Figure 14). Every minute, the first part checks the reservation length, and in case of exceeding the time limit, the reservation is removed. The other part checks the reservations of the drivers entering the parking lot, and if such an entry exists, it is deleted.

Every minute, the inject node loads the reservations from the database and converts them to JSON. The data are passed by the FirebaseConvert (code: check out booking timeout) function, which checks the length of every reservation and in case of exceeding the time limit and passes the respective reservation for deletion. If a reservation that should be deleted is found, the switch continues with the second path. Otherwise, it continues with the first path, where the program ends.

```
var response = msg.payload;
var data = "";
var nic = 1;
var data1 = response.split("{").toString();
var l = data1.length;
for(i=0;i<l;i++){
    var s = data1.substring(i, i + 1);
    if(s == "\""){}
    else if(s == "{"){}
    else if(s == "}"){}
    else {data = data + s;}
}
var allmsg = data.split(",–");
l = allmsg.length;
for(i=1;i<l;i++){
var data1 = allmsg[i].split(":");
var fh = parseInt(data1[4]);
```

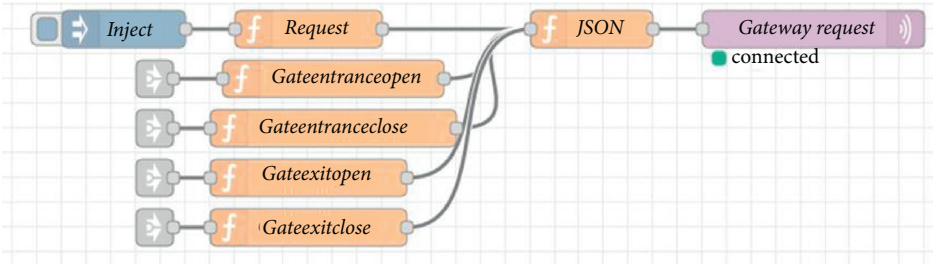

Figure 12: Program flow IQRF_Request.

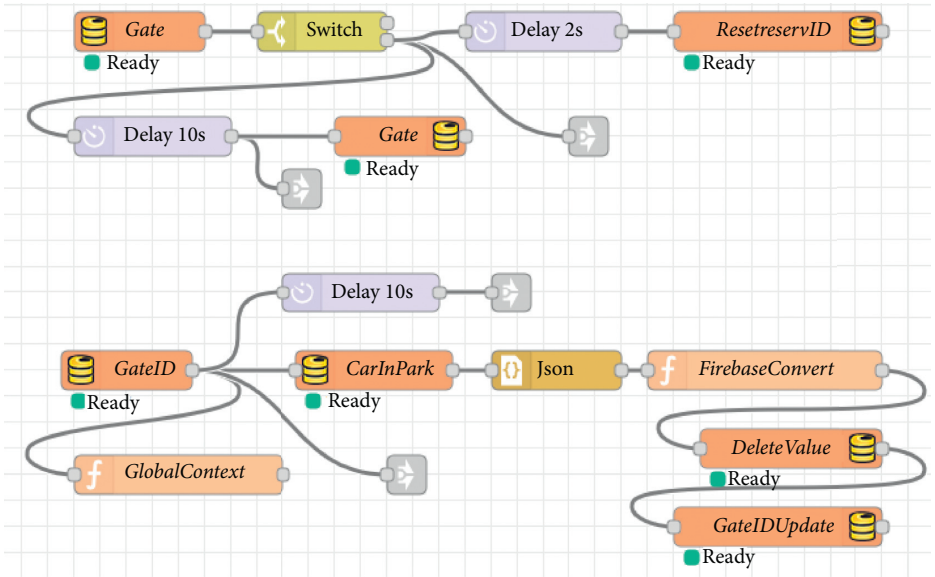

Figure 13: Program flow gate.

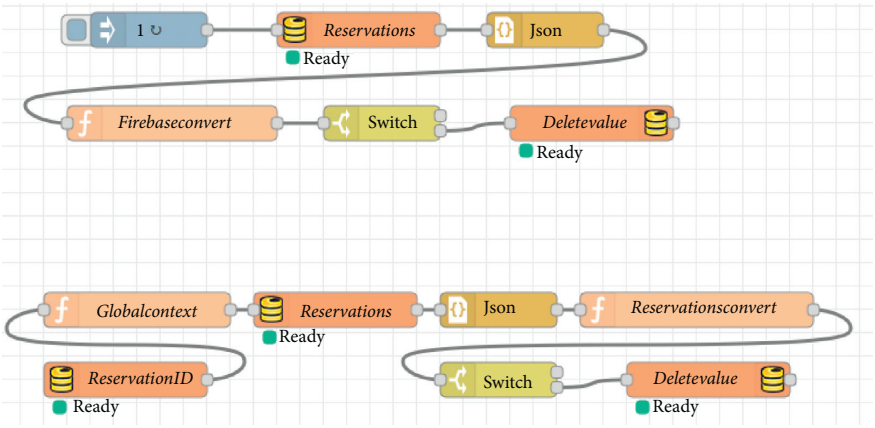

Figure 14: Parking reservation management.

```
var fm = parseInt(data1[5]);
var date = new Date();
var h = date.getHours()+1;
var m = date.getMinutes();
var delta = (h * 60 + m)−(fh*60+fm);
  if(delta > 60){
    var cesta = "parkings/parking1/reservations/
−" + data1[0];
    node.send({childpath:cesta});
  }
  else{
    node.send({payload:nic});
```

```
    }
}
return;
```

As mentioned previously, when the entrance barrier is opened, the phone ID is written to the reservationID entry. The change of its state is monitored by the ResourceID node, which begins the second part of this flow. The phone ID is stored in a global variable, and all reservations are read from the database, converted to JSON format and passed to the ReservationConvert (code: delete driver's reservation for parking) function, which compares the records in the database with the stored phone ID. When a match is found, the entry is deleted. The switch performs the same function as in the previous case.

```
var response = msg.payload;
var reservationID = global.get("reservationID");
var data = "";
var data1 = response.split("{").toString();
var l = data1.length;
for(i=0;i<l;i++){
  var s = data1.substring(i, i + 1);
  if(s == "\"){}
  else if(s == "{"){}
  else if(s == "{"){}
  else {data = data + s;}
}
var allmsg = data.split(",–");
var al = allmsg.length;
for(i=1;i<al;i++){
  var data3 = allmsg[i].split(",");
  var androidID = data3[1].substring(10, data3
[1].length);
  var path = "parkings/parking1/reservations/" +
  data3[2].substring(3, data3[2].length);
  if(androidID == reservationID){
  node.send({childpath:path});
  else {node.send({payload:1});}
}
return;
```

5.6. Smartphone Application. For the purposes of the testing and validating, an Android app has been developed. This app enables control of all the functions on the screen, so the controls are user-friendly and simple. The functions are started by tapping simple icons, so the driver is not forced to read through the context menus while driving. The overview of all the functions and meaning of the individual icons is represented by the image.

5.7. Automatic Detection of a Nearby Parking Lot. As described in chapter 4.5, the automatic detection of a nearby parking lot is procured via geofencing service For this purpose, the Constants class is used. It contains the parameters such as creating a geofence (GPS coordinates and radius). Using a hash map, several geofences can be created simply by adding a name and coordinates via another command.

```
LANDMARKS.put(. . .. . .)); (code: set geofence
parameters).
public class Constants {
  public static final float GEOFENCE_RADIUS_
IN_METERS = 1000;
  public static final HashMap < String,
LatLng > LANDMARKS = new HashMap < String,
LatLng > ();
  static {
    // Parking 1
    LANDMARKS.put("Parking 1", new LatLng
(50.420860, 16.185796));
  }
}
```

By calling the populateGeofenceList() method in MainActivity, custom geofence is created (code: creating geofence).

```
public void populateGeofenceList() {
  for (Map.Entry < String, LatLng > entry: Con-
stants.LANDMARKS.entrySet()) {
    mGeofenceList.add(new Geofence.Builder()
      setRequestId(entry.getKey())
      setCircularRegion(
        entry.getValue().latitude,
        entry.getValue().longitude,
        Constants.GEOFENCE_RADIUS_IN_
        METERS
      )
      setExpirationDuration(Geofence.
      NEVER_EXPIRE)
      setTransitionTypes(Geofence.GEOFENCE_
      TRANSITION_ENTER)
      build());
  }
}
```

The GeofenceTransitionsIntentService class subsequently creates a notification channel that it uses to inform the driver with a notification about available parking lot upon entering inside the created geofence (Figure 15).

5.8. Navigation to the Parking Lot. To run the navigation, the app uses Google API and Google Maps for Android, which enables running a map in the following modes:

Display the map on a given place with given zoom level

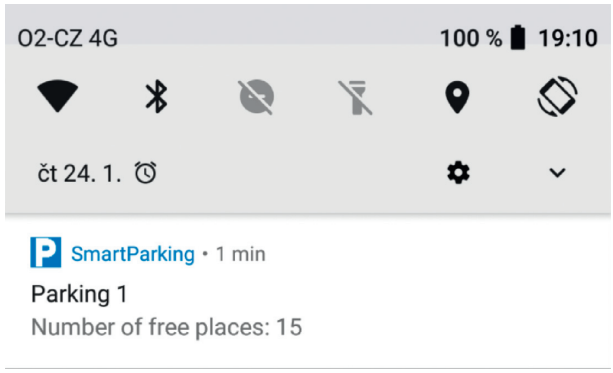

Figure 15: Notification after entering geofence.

Searching for a place and displaying it on the map

Navigation with the selected means of transport (car, bicycle, and walking)

Displaying the panorama view in Google Street View service

To run the map, it is first necessary to create an "Intent" object and specify the mode in which the map will be opened. Intent contains a special string (URI), which accurately specifies the requested action. After creating the Intent, the activity gets started by the startActivity() method. As you can see in the following code (code: create Intent and run Map Activity), Intent is created via URI, which defines the type of the action as navigation to a set GPS coordinate. Subsequently, the Intent is injected with a packet to secure that is processed by Google Maps.

```
public void startNavigationButtonHandler(View view)
{
  Uri gmmIntentUri = Uri.parse("google.navigation:
q = 50.475367, 16.179489");
  Intent mapIntent = new Intent(Intent.ACTION_-
VIEW, gmmIntentUri);
  mapIntent.setPack-
age("com.google.android.apps.maps") ;
  startActivity(mapIntent);
}
```

5.9. Information about the Parking Lot Occupancy. The information about number of available and occupied parking space is saved by NodeRED into the Firebase database, and in the Android app, the instance of the database is created: FirebaseDatabase database = FirebaseDatabase.getInstance(), along with the references for its individual items: DatabaseReference myRef. . . = database.getReference("parkings/parking1/. . .. ."). This is followed by the listener performing the corresponding actions every time a value changes in any of the values it is monitoring.

```
public void readFreePlaces(){
  myRefFplaces.addValueEventListener(new Val-
ueEventListener() {
    @Override
  public void onDataChange(@NonNull DataSnap-
shot dataSnapshot) {
      places = dataSnapshot.getValue().toString();
      int pl = Integer.parseInt(places);
      int cr = (int) (pl – countRes);
      Fplaces.setText(String.valueOf(cr));
  }
    @Override
  public void onCancelled(@NonNull DatabaseError
databaseError) {
      places = "Error";
  }
});
```

On every change of the available parking spaces, the method above (code: setting the listener to change the number of free parking spaces) sets the textView value on the main activity via the setText method, so the button always contains the current number of available spaces on the parking lot. When tapping this button, a new activity is run. It contains listView, which informs the driver about current occupancy of the individual parking spaces on the parking lot. The same activity is run also upon entering the parking lot after the request to open the entry gate, so the driver has an overview of the available spaces. For this purpose, the app uses, similarly to the previous case, the data stored in the database, whose changes are monitored by the listener. The data about the occupancy of the individual spaces are stored in the database via NodeRED in the form of a string "0000110100110010", which contains sixteen bits: "1" or "0". If the given position has the value of "1", it means that the space is occupied. Otherwise, the space is available. The getOccupancy() method (code: filling a text field based on database data) in this activity fills the OCCUPANCY[] array with the values "FREE" or "OCCUPIED".

```
public void getOccupancy(){
  for(int i=0;i < 8;i++){
    System.out.println(places.substring(i, i + 1));
    if(places.substring(i, i + 1).equals("0")){OCCU-
PANCY[i] = "FREE";}
    if(places.substring(i, i + 1).equals("1")){OCCU-
PANCY[i] = "OCCUPIED";}
    if(places.substring(i + 8, i + 9).equals("0")){OC-
CUPANCY[i + 8] = "FREE";}
    if(places.substring(i + 8, i + 9).equals("1")){OC-
CUPANCY[i + 8] = "OCCUPIED";}
    NUMBERS[i] = i + 1;
  }
}
```

Afterwards, these values are in the adapter using listView passed to individual textView (code: filling text boxes and setting font color based on content), as is shown in the image (Figure 16).

Figure 16: Parking occupancy is displayed.

```
public View getView(int i, View view, ViewGroup viewGroup) {
        view-
= getLayoutInflater().inflate(R.layout.customlayout, null);
        TextView
tvNumbers = view.findViewById(R.id.tvNumbers);
        TextView
tvOccupancy = view.findViewById(R.id.tvOccupancy);
        TextView tvOccu-
pancy1 = view.findViewById(R.id.tvOccupancy1) ;
        tvNumbers.setText(String.valueOf(NUMBERS[i]));
        tvOccupancy.setText(String.valueOf(OCCU-
PANCY[i]));
        tvOccupancy1.setText(String.valueOf(OCCU-
PANCY[i+8]));
        if(OCCUPANCY[i].equals("FREE")){tvOccu-
pancy.setTextColor(Color.parseColor("#00ff00"));}
if(OCCUPANCY[i].equals("OCCUPIED")){tvOccu-
pancy.setTextColor(Color.parseColor("#ff0000"));}
if(OCCUPANCY[i+8].equals("FREE")){tvOccu-
pancy1.setTextColor(Color.parseColor("#00ff00"));}
if(OCCUPANCY[i+8].equals("OCCUPIED")){tvOc-
cupancy1.setTextColor(Color.parseColor("#ff0000"));}
        return view;
    }
}
```

5.10. Creating a Parking Space Reservation. Every driver can make one short-term reservation of one parking space. For this purpose, the app uses the Reservation.java class, which has three attributes:

id–item ID generated by the Firebase database

androidID–unique 64-bit number generated by the Android OS

time–the time when the reservation is created

Before creating a reservation, based on the Android ID, the app checks whether the user is already saved in the database (has a valid reservation or is already on the parking lot) (Sample 14) and if there are any spaces available for reservation on the parking lot (number of free spaces—number of valid reservations). If a reservation is possible, it asks the database for a new item id: id = myRefRes.push().getKey(). It then creates a new instance of the Reservation class: myRefRes.child(id).setValue(reservations; and finally, this newly created instance is stored in the database (Figure 17): myRefRes.child(id).setValue(reservations) (code: determining the status of a user's reservation).

```
public void onDataChange(DataSnapshot snapshot) {
    countRes = snapshot.getChildrenCount();
    updateCountReservation();
    resIndicator = false;
    for (DataSnapshot postSnapshot: snapshot.getCh-
ildren()) {
        Reservation
post = postSnapshot.getValue(Reservation.class);
        String resCheck = post.getAndroidID();
        if(resCheck.equals(androidID)){resIn-
dicator = true;}
    }
}
```

The state of the reservations is regularly monitored also by the mini PC via NodeRED, and if the time limit for the reservation is exceeded, it is automatically removed from the database. The reservation is removed also if a driver with a valid reservation arrives on the parking lot and sends a request to open the entry gate. The procedure for removing the reservation from the database is described later.

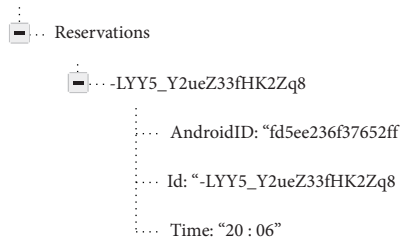

Figure 17: Example of saving the reservation to the database.

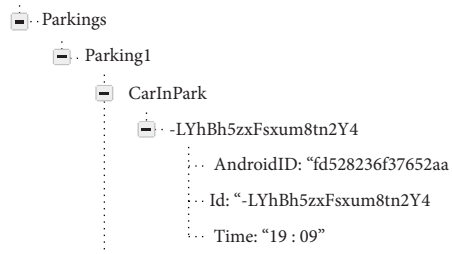

Figure 18: Record the vehicle present in the parking lot.

5.11. Opening the Entry/Exit Gate. To open the entry/exit gate, the app again uses the Firebase database, which contains two items:

gate: it can contain two values: 1 and 0 (entry gate open/closed)

gateID: in case the car is present on the parking lot, the app saves the androidID of the user who wants to open the exit gate

The app contains the CarInPark.java class, which has, similarly to the Reservation.java class, three attributes:

id–item ID generated by the Firebase database

androidID–unique 64-bit number generated by the Android OS

time–time of arrival on the parking lot

The same as with the requirement to create a reservation, during the requirement to open the gate, the app first checks whether the vehicle is currently present at the parking lot (has a record in the database) via androidID. Then, by comparing two GPC coordinates, it checks whether the maximum distance of the smartphone from the gate is not exceeded. If these conditions are met, by entering the value of "1" to the gate item of the Firebase database, NodeRED procures that the entry gate is open. Then, it follows with a request for a new id to save the item in the database: String id = myRefCar.push().getKey(); then, a new instance of the CarInPark class is created: carInPark = new CarInPark(id, androidID, time); and finally, the item is saved in the database: myRefCar.child(id).setValue(carInPark) (Figure 18). After closing the gate, using NodeRED, a potential reservation belonging to the driver entering the parking lot is deleted.

If the vehicle is on the parking lot (user's androidID is in the database), androidID is saved in the gateID item, and based upon this action, NodeRED opens the exit gate and removes the corresponding user's record. The processes of opening and closing the parking lot gates using NodeRED have been described in Chapter 6.3.

6. Discussion

During the pilot operation, vehicle detection using a magnetometric sensor has been tested in two phases. In the first phase, unprompted detection testing has been performed. In the second phase, testing for detection of vehicles of various sizes was followed. For the first phase of the testing, an Arduino module program was modified to count the vehicle detections. The testing was carried out for the period of 24 hours, performing a check twice per second. The sensor was placed and fixed so its move in any direction was impossible. In 24 hours, 172,800 checks were performed, and only four detections were faulty, with the error rate of 2.3%.

In the second phase, correct detection of five vehicles of different sizes (Toyota Yaris, Škoda Roomster, Škoda Octavia combi, Citroën Jumpy, and Nissan Navara) was tested. Every vehicle was placed thirty times over the magnetometric sensor. The detection success rate was very high—there has been only one faulty detection, which happened with the smallest of the vehicles (Toyota Yaris).

The automatic nearby parking lot detection has been carried out over the course of two weeks. The first week, testing for choosing the appropriate radius of the geofence was carried out, so the driver would get an information about an available parking lot in time and would be able to respond to this information easily. After one-week testing, 1,000 meters was chosen as the most suitable radius. Because of the delay described earlier, the driver in town traffic would the information about 500 to 800 meters before the parking lot. The second week, after setting the suitable radius, testing on a 10-kilometer route was carried out. On this route, five geofences were set up. Ten drivers have driven in this route ten times in total, which means that 500 tests have been performed. In three cases, the notification was not received, and in four cases, the driver received the notification in distance under 200 meters from the parking lot. This adds up to the error rate of 1.4%.

One of the important factors for successful adoption of any car parking solution is the economic efficiency. The financial costs of traditional and complex solutions, which have been stated in related works, are certainly higher than the low-cost solutions based mini PC. The final costs are also affected by the selection of individual components, so the final cost of the solution was not high. The approximate retail prices, at which the individual components of the system were purchased, are summarized in the following list:

MPU–9250 (3 €)

Arduino Mini (2 €)

IQRF TR-72D (14 €)

Battery (19 €)

UbBoard-mini PC (90 €)

Entrance barrier (606 €)

7. Conclusion

The presented solution has been used for pilot deployment, and in case the solution was to be used commercially, some enhancements of its functionality would be needed. In the current state, since the time is saved in the database upon vehicle entering the parking lot, the app would be capable of calculating the parking fee if price tariffs were to be added. For the commercial use, it would be necessary to implement a way to pay the parking fee, which could be done either using a parking machine placed directly on the parking lot or implementing a connection to a payment gateway to the app. It would also be suitable to also develop an iOS app, so that the iPhone users could use the system as well. Another option would be to develop a web app for communication with the system. The app would run directly in a smartphone web browser, and therefore, the system would be cross-platform. The majority of the system parts and the mobile app have been designed in the way that would allow future development of more functionalities. The structure of the data stored in the database makes it possible to add additional parking lots, and the automatic parking lot availability detection enables to display notifications about more than one parking lot.

Therefore, after some modifications, the system could be used by, for example, owners of permanent parking lots, also during large occasional events. Considering its low expenses, the system could not only help the drivers solve their problems with searching for empty parking spaces and improve the traffic in the towns and cities, but it could also help to deal with the problems with the drivers who avoid paying the parking fees, especially on the open parking lots with the parking machines.

Data Availability

The measured data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by a Specific Research Project, Faculty of Informatics and Management, University of Hradec Kralove, Czech Republic. We would like to thank Mrs. H. Svecova, a doctoral student, and Mr. J. Dian, a graduate, of Faculty of Management and Informatics, University of Hradec Kralove, for the practical verification of the proposed solutions and close cooperation in the solution.

References

[1] A. Khanna and R. Anand, "IoT based smart parking system," in *Proceedings of International Conference on Internet of Things and Applications (IOTA)*, pp. 266–270, Pune, India, January 2016.

[2] I. Chatzigiannakis, A. Vitaletti, and A. Pyrgelis, "A privacy-preserving smart parking system using an IoT elliptic curve based security platform," *Computer Communications*, vol. 89–90, pp. 165–177, 2016.

[3] S. Kubler, J. Robert, A. Hefnawy, C. Cherifi, A. Bouras, and K. Främling, "IoT-based smart parking system for sporting event management," in *Proceedings of the 13th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services-MOBIQUITOUS 2016 [online]*, pp. 104–114, New York, USA, November 2016.

[4] H. Arasteh et al., "Iot-based smart cities: a survey," in *Proceedings of IEEE 16th International Conference on Environment and Electrical Engineering (EEEIC)*, pp. 1–6, Florence, Italy, June 2016.

[5] MF. Tsai, Y. C. Kiong, and A. Sinn, "Smart service relying on internet of things technology in parking systems," *The Journal of Supercomputing*, vol. 74, no. 9, pp. 4315–4338, 2018.

[6] C. Roman, R. Liao, P. Ball, S. Ou, and M. de Heaver, "Detecting on-street parking spaces in smart cities: performance evaluation of fixed and mobile sensing systems," *IEEE Transactions on Intelligent Transportation Systems*, vol. 19, no. 7, pp. 2234–2245, 2018.

[7] I. Calvo, J. M. Gil-García, I. Recio, A. López, and J. Quesada, "Building IoT applications with raspberry Pi and low power IQRF communication modules," *Electronics*, vol. 5, no. 4, p. 54, 2016.

[8] V. Jan, P. Martin, and R. Hajovsky, "Wireless measurement of carbon dioxide by use of IQRF technology," *IFAC-PapersOnLine*, vol. 51, no. 6, pp. 78–83, 2018.

[9] J. Skovranek, P. Martin, and R. Hajovsky, "Use of the IQRF and Node-RED technology for control and visualization in an IQMESH network," *IFAC-PapersOnLine*, vol. 51, no. 6, pp. 295–300, 2018.

[10] M. Pies and R. Hajovsky, "Using the IQRF technology for the internet of things: case studies," in *Mobile and Wireless Technologies 2017. ICMWT 2017. Lecture Notes in Electrical Engineering*, K. Kim and N. Joukov, Eds., vol. 425, Singapore, Springer, 2017.

[11] J. Kopják and G. Sebestyén, "Comparison of data collecting methods in wireless mesh sensor networks," in *Proceedings of IEEE 16th World Symposium on Applied Machine Intelligence and Informatics (Sami)*, pp. 000155–000160, Kosice, Slovakia, February 2018.

[12] O. Vondrouš, Z. Kocur, T. Hégr, and O. Slavíček, "Performance evaluation of IoT mesh networking technology in ISM frequency band," in *Proceedings of 17th International Conference on Mechatronics-Mechatronika (ME)*, pp. 1–8, Prague, Czech Republic, December 2016.

[13] IQRF Tech sro, *IQRF DPA framework Technical Guide*, Vol. 10, IQRF Tech, Jičín, Czech Republic, 2018.