



ÖZYEGİN UNIVERSITY
FACULTY OF ENGINEERING

NFC CONTROL MECHANISM FOR ELEVATORS

AHMET GENCO GÜVEN
S029392

i. Problem Statement

The main problem addressed was the development of a reliable NFC control system for elevators to ensure that only authorized individuals could access and use them. This solution is particularly relevant in places such as hotels and dormitories, where secure access for guests and residents is required, and unauthorized use must be prevented. By setting up an NFC system, the ability to register and delete cards with NFC features was made possible. The system checks the card's authorization status when read, allowing only those with permission to operate the elevator, ensuring a high level of security and control over access. One of the most significant challenges in this project was the lack of experience with both hardware and software components. Prior to this, there was no experience working with SBCs or NFC modules, making the learning process more demanding as it required understanding how to connect these components and make them work together efficiently. Additionally, it was crucial to ensure that the NFC system operated effectively and stably, as security is a very sensitive issue and any mistakes could lead to unauthorized access.

Previously, Mik-El Elektronik, like many other companies around the world, utilized keycard or biometric systems to regulate elevator access. However, transitioning to NFC technology offers numerous advantages. One of the primary benefits is that it is contactless, which is particularly crucial in the context of hygiene, especially following the global pandemic when awareness of the need for contactless solutions increased. By reducing the necessity for physical contact, NFC systems help prevent the spread of bacteria, providing a cleaner and safer option for both customers and staff.

Another advantage is that NFC systems are significantly faster than keycard or biometric systems, allowing for quicker access to elevators without delays. They are also more cost-effective, making them an economical choice for many businesses. Finally, NFC technology is easier to integrate into existing systems, making it a versatile and practical solution for various applications.

ii. Tools and Techniques Used

The Raspberry Pi 4 was selected as the hardware platform due to its cost-effectiveness and powerful processing capabilities, which align with the project's technical requirements. The Raspberry Pi 4's capacity to facilitate hardware interaction through its extensive GPIO pins made it an optimal selection for the project, enabling seamless integration with a range of components such as the NFC reader.

In terms of hardware specifications, the Raspberry Pi 4 features two micro HDMI ports, one of which was utilized to connect to a monitor, and four USB ports, two of which were used for the mouse and keyboard. The device was powered by a Type-C charger and was equipped with Wi-Fi capabilities, which streamlined the overall project. The Raspberry Pi 4 was connected to

the PN532 module using jumper cables from the SDA, GND, and VCC pins. As the operating system, Raspberry OS was installed on the device, which is Raspberry's own operating system that provides a lightweight and efficient environment. The operating system also includes a bunch of pre-installed libraries and essential tools, which facilitated the setup process.

The PN532 NFC module was selected for this project for its reliability and robust support for NFC communication. This module has been designed for use in interacting with NFC cards and provides a wide range of functionalities that are essential for this task. It is well documented and widely used, making it a trusted choice for NFC applications. While there are other options, such as the RC522, which is cheaper but has fewer features, and the ACR122U, which has more features but is more expensive, the PN532 is optimal for this project.

I2C communication was chosen for this project due to the fact that it is efficient and simple in connecting multiple devices. I2C facilitates straightforward communication between the Raspberry Pi and the PN532 NFC module. In contrast to SPI, it employs just two wires (SCL and SDA) and unlike UART, the protocol is capable of supporting multiple slave devices, which can be incorporated into this project at a later stage. These attributes are especially advantageous for initiatives involving diverse peripherals, such as the NFC reader and additional sensors, enabling a compact and structured wiring configuration.

In terms of software, Python was selected as the main programming language due to its simplicity, flexibility, and the extensive range of libraries and frameworks it offers. The clear and concise syntax of Python makes it an ideal choice for integrating various hardware components, such as NFC readers, and controlling GPIO pins on the Raspberry Pi. Furthermore, the robust community support and comprehensive documentation available for Python proved invaluable in troubleshooting and skill acquisition, offering a distinct advantage. While Python's processing speed is slower than some languages like C++ and Java, it did not present any significant limitations for this phase of the project.

The Flask framework was selected for its lightweight, user-friendly design and ability to streamline development. Flask offered a flexible and straightforward way to build a web application interface for the NFC system, providing more control without the added complexity found in larger frameworks like Django. If the project continues to be developed in the future, it may be necessary to use large frameworks in the style of Django.

The Adafruit libraries were utilized to streamline communication between the Raspberry Pi and the NFC reader. They were selected for their reliability and robust support for the PN532 module, which enables seamless interaction with NFC cards.

The RPi.GPIO library was employed to oversee the Raspberry Pi's GPIO pins for the purpose of controlling access mechanisms. The library provided straightforward functions for managing the GPIO pins, which was critical for sending signals to the access control hardware.

JSON was selected as the data storage format because it allows for the efficient saving and retrieval of registered NFC card information. Its human-readable format and simplicity make it an ideal choice for storing small sets of data without the need for more complex database management systems. However, when the project is implemented in the field, a larger data

set and more structured queries will be required. Therefore, it will be more effective to use MySQL or a more robust database.

In addition to the previously mentioned tools, an oscilloscope was also frequently used in the project to identify and address any errors that may have occurred during the connection, coding or NFC reading phases.

iii. Detailed Explanation

The objective of the project should first be briefly recalled. By integrating the Raspberry Pi 4 with the PN532 NFC module, a system was established that allows access to the elevator buttons only through cards defined by the NFC module. This ensures that only the intended individuals (customers, employees, etc.) are able to utilize the elevator in certain businesses. The process is achieved by sending output signals from the Raspberry Pi 4 when an authorized card is read. The manual implementation of this system to the elevator cabin was not tasked.

To set up the project, an SBC, an NFC module, and jumpers are required for connection. A Raspberry Pi 4 was used as the SBC, as it is a cost-effective device with good integration capabilities with other components. The PN532 was selected as the NFC module due to its widespread use and robust functionality. To power the PN532, pin 2 (5V) of the Raspberry Pi was connected to the VCC pin of the PN532. Pin 6 (GND) of the Raspberry Pi was connected to the ground pin of the PN532. To establish the I2C connection, the Raspberry Pi's pin 3 (SDA) was connected to the PN532's SDA pin, and pin 5 (SCL) was connected to the PN532's SCL pin. In this step, the previously soldered headers were connected to each other using female-to-female jumpers.

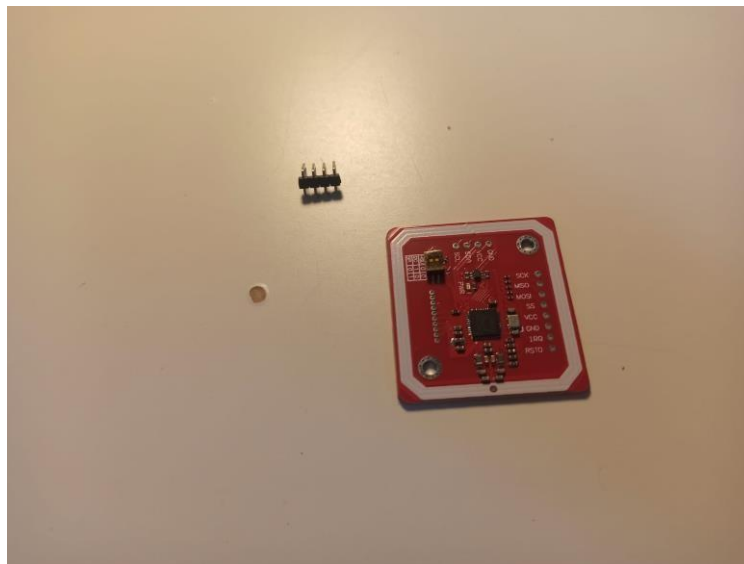


Figure 1: PN532 NFC Module and Header before soldering.

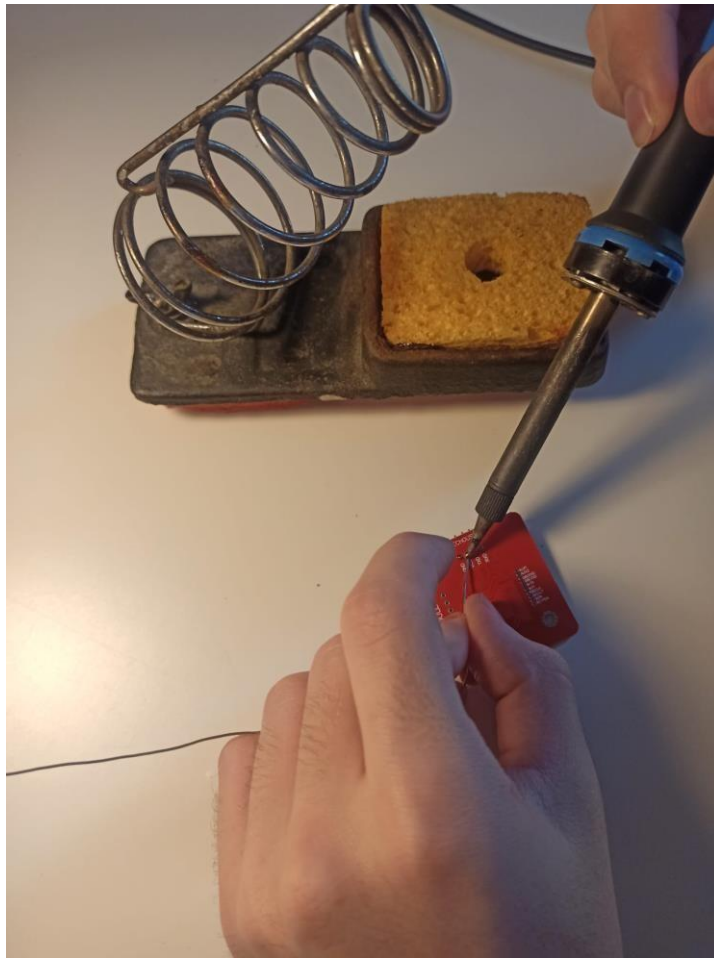


Figure 2: Soldering process of PN532 NFC Module and Header.

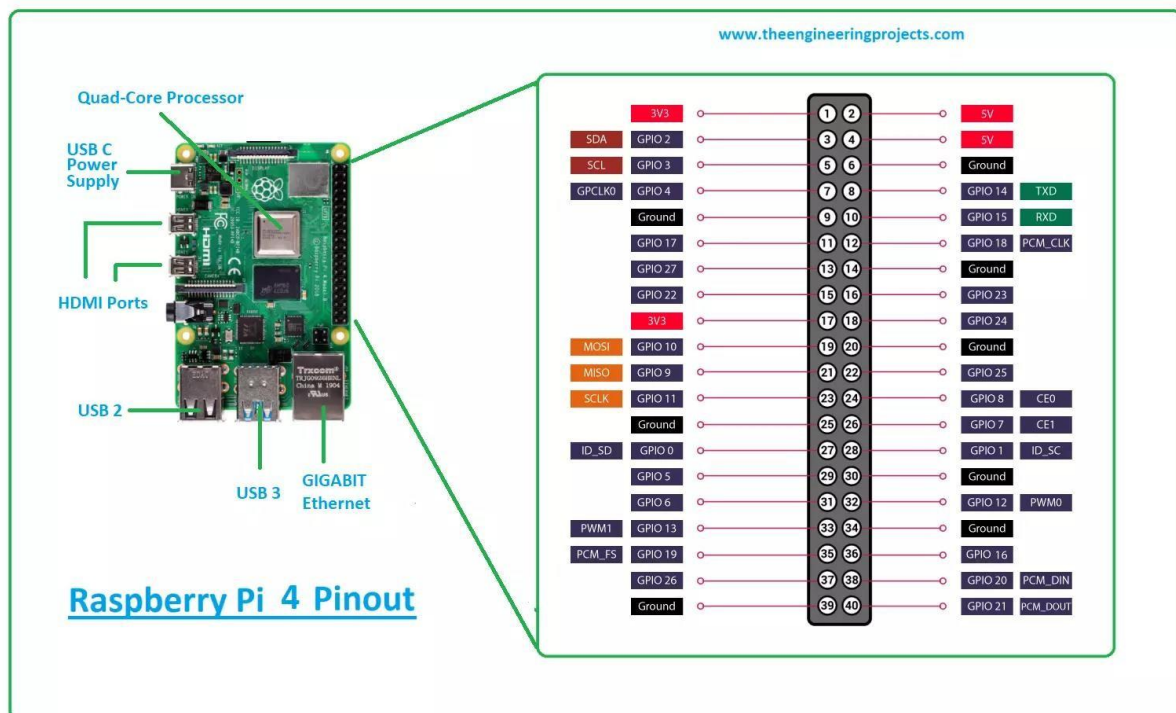


Figure 3: Raspberry Pi 4 Pinout.

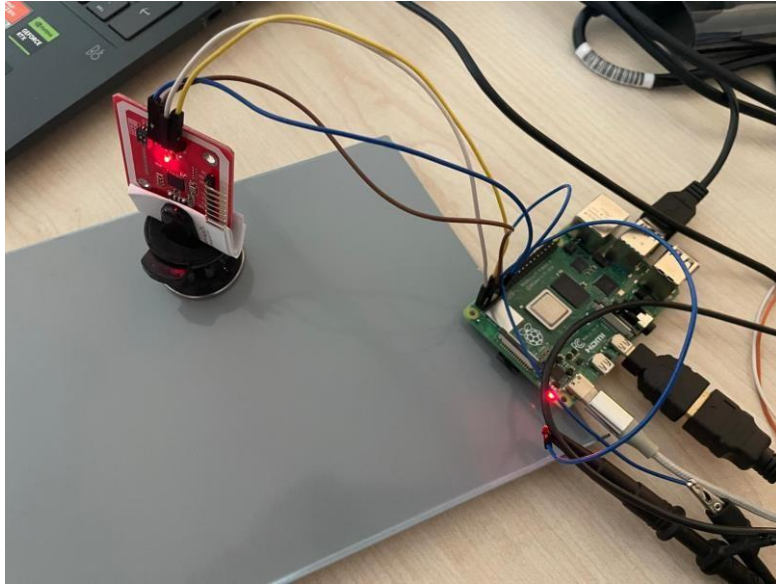


Figure 4: Raspberry Pi 4 and PN532 NFC Module connected and ready to read.

The software is divided into three sections. The first mode is designated as the "register mode," which is used to define or delete NFC cards. The second section is the "access control" module, which is used on a regular basis. This module enables the elevator buttons to be operated based on whether the card has been defined or not. Finally, the "server" module serves as the connecting element between these two sections.

Register mode branch starts with the following lines:

```
import board
import busio
from digitalio import DigitalInOut
from adafruit_pn532.i2c import PN532_I2C
import time
import json
import os
```

- `import board`: This module was used to interface with the hardware pins on the Raspberry Pi 4.
- `import busio`: Used to manage I2C communication between the Raspberry Pi 4 and the PN532 NFC module.
- `from digitalio import DigitalInOut`: Manages digital I/O operations, allowing control of the reset and request pins.
- `from adafruit_pn532.i2c import PN532_I2C`: Imports the `PN532_I2C` class to communicate with the PN532 NFC module over I2C.
- `import time`: Provides time-related functions, such as delays and timestamps.

- `import json`: Handles reading and writing data in JSON format, which is essential for data storage and retrieval.
- `import os`: Provides functionality to interact with the operating system, such as checking for file existence or manipulating files and directories.

```
i2c = busio.I2C(board.SCL, board.SDA)
```

The I2C object is initialized using `busio.I2C`. The clock and data lines needed for I2C communication are represented by `board.SCL` and `board.SDA`.

```
reset_pin = DigitalInOut(board.D6)  
req_pin = DigitalInOut(board.D12)
```

The reset (`board.D6`) and request (`board.D12`) pins are defined as `DigitalInOut` objects, allowing them to be controlled.

```
pn532 = PN532_I2C(i2c, debug=False, reset=reset_pin, req=req_pin)
```

The `PN532_I2C` instance is created, linking it to the I2C object and associating it with the specified reset and request pins.

```
pn532.SAM_configuration()
```

The PN532 module is configured to communicate with MiFare NFC cards by calling `SAM_configuration()`. This setup is necessary to start reading NFC cards.

```
file_path = "nfc_cards.json"
```

The pre-created path to the JSON file (`nfc_cards.json`) where registered NFC card UIDs will be stored is specified.

```
def read_list():  
    if  
os.path.exists(file_path):  
    with open(file_path, 'r') as f:  
        return json.load(f)  
else:  
    return []  
    def write_list(data):  
        with  
open(file_path, 'w') as f:  
            json.dump(data, f)
```

```
def uid_to_hex(uid):
    return [hex(i) for i in uid]
```

- `read_list()`: Checks if the "nfc_cards.json" file exists. If it exists, it opens the file in read mode ('r') and loads its contents using "json.load()". If the file doesn't exist, an empty list is returned.
- `write_list(data)`: Opens "nfc_cards.json" in write mode and writes the given data (list of UIDs) to the file using "json.dump()".
- `uid_to_hex(uid)`: Converts each byte of the UID into its hexadecimal representation in order to facilitate the reading and handling process of UIDs.

```
print("Waiting for NFC card for registration...") while
True:
    uid = pn532.read_passive_target(timeout=0.5)
    if uid is not
None:
        uid_hex = uid_to_hex(uid)
    print(f"Detected UID: {uid_hex}")
    cards_list = read_list()
        if uid_hex in
cards_list:
            print("Card already registered.
Removing...")            cards_list.remove(uid_hex)
        else:
            print("Registering new card...")
    cards_list.append(uid_hex)

    write_list(cards_list)

    time.sleep(2)
```

- `print("Waiting for NFC card for registration...")`: Displays a message indicating that the system is ready to detect NFC cards.
- `while True`: An infinite loop that keeps the program running continuously.
- `uid = pn532.read_passive_target(timeout=0.5)`: Listens for an NFC card within a 0.5second timeout window using `read_passive_target`. If a card is detected, its UID is captured and stored in `uid`. If no card is detected within 0.5 seconds, the loop continues.

When a Card is Detected:

- `uid_hex = uid_to_hex(uid)`: Converts the UID to a list of hexadecimal values for easier handling.
- `print(f"Detected UID: {uid_hex}")`: Displays the detected UID.
- `cards_list = read_list()`: Reads the list of registered cards from `nfc_cards.json`.

Checking and Modifying the Card List:

- `if uid_hex in cards_list`: Checks if the card is already registered. If true, the card is removed from `cards_list`. Otherwise, it appends the card's UID to `cards_list`.
- `write_list(cards_list)`: Writes the updated list back to `nfc_cards.json`.
- `time.sleep(2)`: Introduces a 2-second delay before the loop runs again to avoid excessive reading.

Access control branch starts with the following lines:

```
import board
import busio
from digitalio import DigitalInOut
from adafruit_pn532.i2c import PN532_I2C
import time
import RPi.GPIO as GPIO
import json
import os
```

The imported libraries and modules are pretty much the same except for “`import RPi.GPIO as GPIO`” which imports the `RPi.GPIO` library used for controlling the GPIO pins. This is needed for the GPIO pins to send signals based on card scans.

```
OUTPUT_PIN = 17
```

- `OUTPUT_PIN = 17`: Defines the GPIO pin number to be used for sending output signals. Here, `GPIO17` is assigned to `OUTPUT_PIN` so when an authorized card is read, `GPIO17` will send 5V signal.

```
GPIO.setmode(GPIO.BCM)
GPIO.setup(OUTPUT_PIN, GPIO.OUT)
```

- `GPIO.setmode(GPIO.BCM)`: Configures the GPIO library to use the BCM (Broadcom) pin numbering scheme.
- `GPIO.setup(OUTPUT_PIN, GPIO.OUT)`: Sets the defined output pin as an output pin, allowing it to send signals.

```

i2c = busio.I2C(board.SCL, board.SDA)
reset_pin =
DigitalInOut(board.D6) req_pin =
DigitalInOut(board.D12)
pn532 = PN532_I2C(i2c, debug=False, reset=reset_pin,
req=req_pin)

pn532.SAM_configuration()
file_path =
"nfc_cards.json"
def read_list():    if
os.path.exists(file_path):
with open(file_path, 'r') as f:
return json.load(f)
else:
return []
def write_list(data):    with
open(file_path, 'w') as f:
json.dump(data, f)
def
uid_to_hex(uid):
return [hex(i) for i in uid]

```

These parts are the same as register_mode.py

```

print("Waiting for NFC card for access control...") while True:
uid = pn532.read_passive_target(timeout=0.5)

```

- print("Waiting for NFC card for access control..."): Displays a message indicating that the system is ready to detect NFC cards.
- while True: Starts an infinite loop to continuously check for NFC card scans.
- uid = pn532.read_passive_target(timeout=0.5): Listens for NFC cards with a timeout of 0.5 seconds. If a card is detected, its UID is stored in uid; otherwise, uid will be None.

```

if uid is not None:
uid_hex = uid_to_hex(uid)
cards_list = read_list()

```

Checks if a card has been detected. If so, it converts the UID to hexadecimal format and reads the list of registered cards from nfc_cards.json.

```

if uid_hex in cards_list:    print("Access
granted!")

```

```

        GPIO.output(OUTPUT_PIN, GPIO.HIGH)
time.sleep(2)
        GPIO.output(OUTPUT_PIN, GPIO.LOW)
else:
    print("Access denied!")

```

- if uid_hex in cards_list: Checks if the detected card's UID is in the list of registered cards. If it is, prints "Access granted!" and sends a HIGH signal (5V) to the output pin, allowing access. After that the script waits for 2 seconds while the signal is HIGH and then sets the pin LOW to stop the signal. If the UID is not found, it prints "Access denied!".

```
time.sleep(2)
```

Introduces a 2-second delay before the next iteration of the loop to avoid excessive card scanning.

Index.html branch starts with the following lines:

```

<!DOCTYPE html>
<html lang="en">

```

First line declares the document type and version of HTML. It tells the browser to render the page in standards mode.

The <html> tag is the root element of the HTML document and the lang attribute specifies the primary language of the document, which is English in this case.

```

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
initialscale=1.0">
  <title>NFC Control</title>
</head>

```

- The <head> section contains meta-information about the document.
- <meta charset="UTF-8">: Sets the character encoding for the document to UTF-8, allowing it to display a wide range of characters.
- <meta name="viewport" content="width=device-width, initial-scale=1.0">: Configures the viewport for responsive design. It ensures that the webpage scales correctly on different devices, particularly mobile devices.
- <title>NFC Control</title>: Sets the title of the webpage, which appears in the browser tab.

```

<body>
  <h1>NFC Control Panel</h1>
  <form action="/start/register" method="get">
    <button type="submit">Start Register Mode</button>
  </form>

```

The `<body>` section contains the content that will be visible on the webpage.

- `<h1>NFC Control Panel</h1>`: Displays a main heading for the control panel, giving users an overview of what the page is about.
- `<form action="/start/register" method="get">`: Defines a form that, when submitted, sends a GET request to the `/start/register` endpoint of the Flask application.
- `<button type="submit">Start Register Mode</button>`: Creates a button that the user can click to start the registration mode.

```
<form action="/stop/register" method="get">
    <button type="submit">Stop Register Mode</button>
</form>
```

This form submits a GET request to the `/stop/register` endpoint, allowing users to stop the registration mode.

```
<form action="/start/access" method="get">
    <button type="submit">Start Access Control Mode</button>
</form>
```

This form submits a GET request to the `/start/access` endpoint, enabling users to start the access control mode.

```
<form action="/stop/access" method="get">
    <button type="submit">Stop Access Control Mode</button>
</form>
```

This form submits a GET request to the `/stop/access` endpoint, allowing users to stop the access control mode.

Server branch starts with the following lines:

```
from flask import Flask, render_template, redirect,
url_for import subprocess
import os
```

- `from flask import Flask, render_template, redirect, url_for`: Imports the Flask library and some helper functions:
 - `Flask`: The main class for creating the Flask application.
 - `render_template`: Renders HTML templates using Jinja2.
 - `redirect`: Redirects the user to a different route after an action is performed.
 - `url_for`: Generates URLs for the specified functions.
- `import subprocess`: Allows the script to spawn new processes, connect to their input/output/error pipes, and obtain their return codes.
- `import os`: Provides a way to interact with the operating system, such as checking file existence or handling file paths.

```
app = Flask(__name__)
```

“app = Flask(__name__)” creates an instance of the Flask application. __name__ is passed to the Flask constructor to help locate resources such as templates and static files.

```
REGISTER_MODE_SCRIPT = '/home/genco/proje/register_mode.py'  
ACCESS_CONTROL_SCRIPT = '/home/genco/proje/access_control.py'
```

Script paths define the file paths to the scripts that handle registration and access control.

```
register_process = None  
access_process = None
```

register_process and access_process: These variables will hold references to the subprocesses started for registering and accessing NFC cards, respectively. They are initialized to None.

```
@app.route('/') def  
index():  
    return render_template('index.html')
```

- @app.route('/'): Defines the home route of the web application.
- def index(): This function is called when the home route is accessed.
- return render_template('index.html'): Renders the index.html template, which contains the control panel UI for starting and stopping the NFC functionalities.

```
@app.route('/start/register') def  
start_register():  
    global register_process    if register_process is None or  
register_process.poll() is not None:  
        register_process =  
subprocess.Popen(['python3', REGISTER_MODE_SCRIPT])  
return redirect(url_for('index'))
```

- @app.route('/start/register'): Defines the route to start the registration mode.
- def start_register(): This function is called when the “/start/register” route is accessed.
- global register_process: Allows modification of the register_process variable defined outside the function.
- if register_process is None or register_process.poll() is not None: Checks if the registration process is not running or has finished. If either condition is true, it starts a new subprocess to run the registration script.
- register_process = subprocess.Popen(['python3', REGISTER_MODE_SCRIPT]): Executes the registration script in a new process using Python 3.
- return redirect(url_for('index')): Redirects the user back to the home page after starting the process.

```
@app.route('/stop/register') def
stop_register():
    global register_process    if
register_process is not None:
register_process.terminate()
register_process = None    return
redirect(url_for('index'))
```

- @app.route('/stop/register'): Defines the route to stop the registration mode.
- def stop_register(): This function is called when the /stop/register route is accessed.
- global register_process: Allows access to the register_process variable.
- if register_process is not None: Checks if the registration process is currently running.
- register_process.terminate(): Terminates the running registration process.
- register_process = None: Resets the register_process variable to None.
- return redirect(url_for('index')): Redirects the user back to the home page after stopping the process.

```
@app.route('/start/access') def
start_access():
    global access_process    if access_process is None or
access_process.poll() is not None:
        access_process =
subprocess.Popen(['python3', ACCESS_CONTROL_SCRIPT])
return redirect(url_for('index'))
```

- @app.route('/start/access'): Defines the route to start the access control mode.
- def start_access(): This function is called when the /start/access route is accessed.
- global access_process: Allows modification of the access_process variable defined outside the function.
- if access_process is None or access_process.poll() is not None: Checks if the access control process is not running or has finished. If either condition is true, it starts a new subprocess to run the access control script.
- access_process = subprocess.Popen(['python3', ACCESS_CONTROL_SCRIPT]): Executes the access control script in a new process using Python 3.
- return redirect(url_for('index')): Redirects the user back to the home page after starting the process.

```
@app.route('/stop/access') def
stop_access():
    global access_process    if
access_process is not None:
access_process.terminate()
access_process = None    return
redirect(url_for('index'))
```

- @app.route('/stop/access'): Defines the route to stop the access control mode.
- def stop_access(): This function is called when the /stop/access route is accessed.
- global access_process: Allows access to the access_process variable.

- if access_process is not None: Checks if the access control process is currently running.
- access_process.terminate(): Terminates the running access control process.
- access_process = None: Resets the access_process variable to None.
- return redirect(url_for('index')): Redirects the user back to the home page after stopping the process.

```
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, debug=True)
```

- if __name__ == '__main__': Ensures that the Flask application runs only if the script is executed directly, not if it's imported as a module in another script.
- app.run(host='0.0.0.0', port=5000, debug=True): Starts the Flask application.
- host='0.0.0.0': Makes the server publicly accessible on all network interfaces.
- port=5000: Sets the port number for the server.
- debug=True: Enables debug mode, which provides detailed error messages and automatic reloads on code changes.

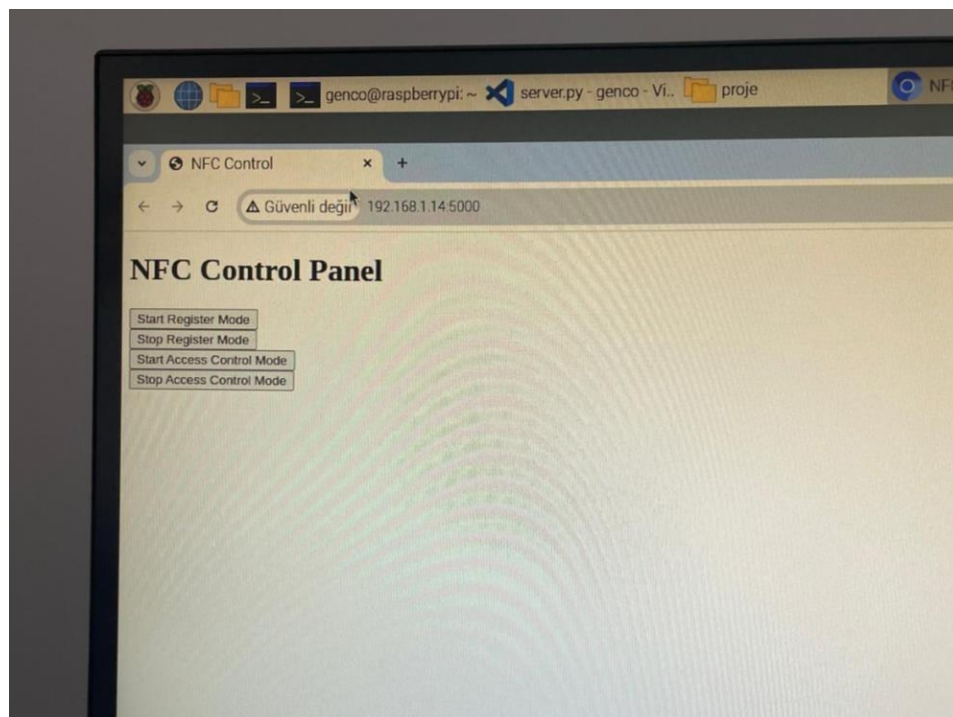


Figure 5: Interface of the NFC Control System.

iv. Results

The solution developed proved to be highly effective and can be considered a success. When the system is set to register mode, any card that is swiped gets registered as an authorized card, and if the same card is swiped again, it is removed from the system, making it easy to manage which cards are authorized. This feature is crucial for adjusting card registration as needed. In the access control mode, which is used daily, the system checks every card swiped. If the card is not authorized, the system immediately notifies the user, and the elevator will

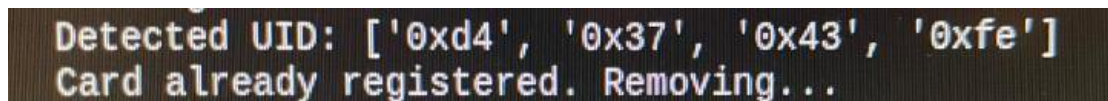
not operate. When an authorized card is used, the Raspberry Pi's output port sends a 5V signal, which should be connected to the elevator, allowing it to function.

Although direct involvement in installing the system into the elevator was not part of the experience, it is believed that once an authorized card is scanned, the system should grant a one-time elevator usage right. This can be managed by adjusting the embedded operating system of the specific elevator model to handle this functionality correctly.

A terminal window with a black background and green text. The text reads: "Detected UID: ['0xd4', '0x37', '0x43', '0xfe']" followed by "Registering new card..." on the next line.

```
Detected UID: ['0xd4', '0x37', '0x43', '0xfe']
Registering new card...
```

Figure 6: Registering a new card to the system in register mode.

A terminal window with a black background and green text. The text reads: "Detected UID: ['0xd4', '0x37', '0x43', '0xfe']" followed by "Card already registered. Removing..." on the next line.

```
Detected UID: ['0xd4', '0x37', '0x43', '0xfe']
Card already registered. Removing...
```

Figure 7: Removing a registered card from the system in register mode.

A terminal window with a black background and green text. The text reads: "Access granted!"

```
Access granted!
```

Figure 8: When a registered card is read in access control mode.

A terminal window with a black background and green text. The text reads: "Access denied!"

```
Access denied!
```

Figure 9: When an unknown card is read in access control mode.

As with any project, there are always opportunities for further improvement. For example, if the NFC control system were to be installed in a larger facility, such as a hotel or dormitory, using a more powerful web framework like Django instead of Flask would be advantageous for handling more complex operations. Additionally, the system could benefit from utilizing a more reliable database such as MySQL to store card IDs, offering better security and organization. Another enhancement would be to create a more user-friendly interface for the staff responsible for switching between the access control mode and register mode, making system management easier.

Moreover, incorporating machine learning algorithms could elevate this project to the next level. By analyzing access patterns, the system could detect unusual activities or attempts to use unauthorized cards, helping to prevent theft or unauthorized access. This would make the system not only more secure but also more intelligent, adapting to the facility's needs over time. Overall, these enhancements would make the NFC elevator control system more efficient, reliable, and suitable for larger-scale implementations. However, the internship period and level of expertise were not sufficient to implement these features.