ECE3375B Project Final Report

Xristopher Aliferis, 251269415

Yash Padhiar, 251297539

Carmel Kurland, 251235621

Submission Date: April 4th, 2025

Professors: Anestis Dounavis and Arash Reyhani-Masoleh

Problem Definition

Gaining secure access to a building or room should be straightforward and hassle-free, yet traditional methods often struggle. Many existing door lock systems rely on physical keys, keypads, or outdated electronic mechanisms that are prone to security breaches, complicated procedures or even something as simple as forgetting your keys. This not only affects the overall security of whatever the lock is protecting but it also adds an increased inconvenience on the individuals who don't have a lot of time and need quick access. Also, with increasing worries in homes, apartments, and hotels about unwanted entry and of key duplication, there is an increasing demand for a more modern and simple solution that can provide increased security and ease of use.

RFID technology provides increased convenience and easier access control; however, it also introduces unique security concerns that must be carefully addressed. RFID systems are widely used across a large amount of application, from public transit to retail asset management. A 2024 case study found that RFID-based smart door locks significantly improved service quality across multiple areas, specifically reliability, security, convenience, and responsiveness. The system allowed guests to quickly and easily access their rooms while also reducing the headache of a lost or malfunctioning key. Survey data from the study revealed that 80% of guests were "very satisfied" and "satisfied" with the ease of use, security, and privacy offered by the RFID system (Putra et al., 2024). It was also found in a 2024 study that RFID locks have extremely high recognition rates, with this article specifically finding 100% success up to 20mm (Kasim et al., 2024).

To address some security concerns, the following must be taken into the account of our design. First, encryption must be implemented on any data stored locally on the device while also being applied to any data transmitting across IoT networks. This prevents any sensitive information getting maliciously intercepted and used to gain unauthorized access. Second, the system must include continuous monitoring of all event logging, recording all access attempts as well as any additions or removals of authorized IDs. This is to provide a second layer of security for any potential spoofing or the use of stolen RFID tags.

One of the primary benefits of RFID door lock systems is their ability to integrate very easily with our current IoT systems. In a world where we find ourselves more connected to the internet every single day, these types of systems not only provide secure access but also real-time data logging, which can be extremely valuable for building managers and security personnel who would be working with them. This connectivity would also support quick security responses in case of any unusual access events. In addition, all data collected by these systems can be used to generate information on user patterns which could increase the optimization and overall efficiency of the areas they are implemented in.

Furthermore, RFID-based door lock systems are extremely scalable. Also, they can be added in many different areas, from residential buildings and office to higher-security settings such as research labs or even government buildings. Because RFID systems are somewhat modular, it allows for upgrades to be made easily and for increased features such as remote access via smartphones. These systems are also every cost effective compared to traditional lock systems may require more expensive key replacement and lock replacement while also being more reliable requiring less maintenance (Putra et al., 2024).

To give some specifications, our system should meet the following criteria in terms of read range, response time, reliability, security, and integration. Our system should accurately detect and read RFID tags that are within a minimum range of 20 mm while also having a response time of 1 second or less depending on the action being performed (unlocking, re-locking, tag addition, tag removal, etc.). It should be reliable meaning it achieves a success rate of 99%, more specifically this means it should perform the expected response 99% of time based on a certain interaction. Lastly our system should have proper anti-spoofing and encryption while also seamlessly being able to integrate with IoT systems for real-time logging to external devices.

In summary, RFID-based door lock systems provide a solution that addresses many of the issues involved with conventional door lock systems. By combining high accuracy reads, quicker access, easy integration with IoT and cost effectiveness, these systems provide a very compelling choice for current day access control systems. This proposed project aims to develop a secure, scalable, and user-friendly RFID door lock system that will increase security and contribute to safer and more efficient environments.

Functional Description

The RFID Door lock system is designed to provide secure and easy access control using RFID technology and automated door actuation. In this system, an RFID reader continuously polls for RFID tags or key fobs presented near the door. Typically polling could be seen as bad but because the system isn't doing anything else during the process of waiting for a fob, polling is completely normal to use in this scenario. When an RFID tag is detected, it will then transmit the unique ID (UID) to the microcontroller (hosted on our DE10 board) and will compare it to a list of authorized credentials. If the UID matches a known one, the system will proceed to unlock the door. If it is not recognized, it stays in a locked state and turns on incorrect UID LED signal.

The RFID reader functions as the main sensor in the system as shown in Figure 1: Physical Overview of RFID Door Lock System, it can detect both types of RFID tags (passive and active) with high accuracy from a short range. When the tag comes close to the reader, the electromagnetic field generated by the reader itself powers the tag (if the tag is passive) and then allows it to transmit its stored data. This transmission is very quick, which makes the authentication process quick as well.

Once the RFID reader captures the UID, the microcontroller then processes the information the reader has send to it instantaneously. It is programmed with the needed specifications to communicate with the internal memory with authorized ID's are stored. If a match is found, a command signal is sent to a motor connected to the locking mechanism. The motor then actuates to physically retract or rotate the locking bolt, which then would unlock the door for authorized entry. In the scenario where the ID's do not match, the microcontroller would identify this, and the door would not unlock, and it would turn on the LED to signal to the user it was a bad UID. If the door is currently unlocked, and the same known RFID is scanned, the door would then re-lock securely. This would also happen naturally after a certain period of the door being unlocked, this increases overall security as it is very common to forget to lock a door manually, so this helps protect against that.

Another key feature of the system is a push button connected to the microcontroller can be used to enable a manual RFID registration and deregistration mode. When this button is pressed, the microcontroller then switches the state, and the RFID reader now will capture the first RFID tag it finds and send that back to the microcontroller. If this ID is not stored in memory yet, it gets added and the state automatically switches back to the regular lock and unlock mode. If the ID is stored in memory, it will get removed from the memory and just like the previous the device will switch back to the regular locking mode. This allows for administrators to manually change the ID's that have access to doors in emergency scenarios or in the case of a lock being used for different purposes.

To ensure continuous operation, the RFID door lock system continuously polls in a monitoring loop. This constant polling is very important because the lock needs to always be ready to respond to a scanned RFID because users don't want to have to wait for the system to turn on and read their UID, rather they want it to happen instantly.

In addition to this written explanation, a physical overview is provided to visually clarify the right and left side view of the RFID-enabled door lock system. Core user-engagement components are all listed and displayed in the diagram (RFID Scanner, LED Indicator, Pushbutton, Locking Mechanism). All components in this diagram are directly connected to the internal microcontroller which would perform the authorization checks as well as controlling the locking mechanism's movement.

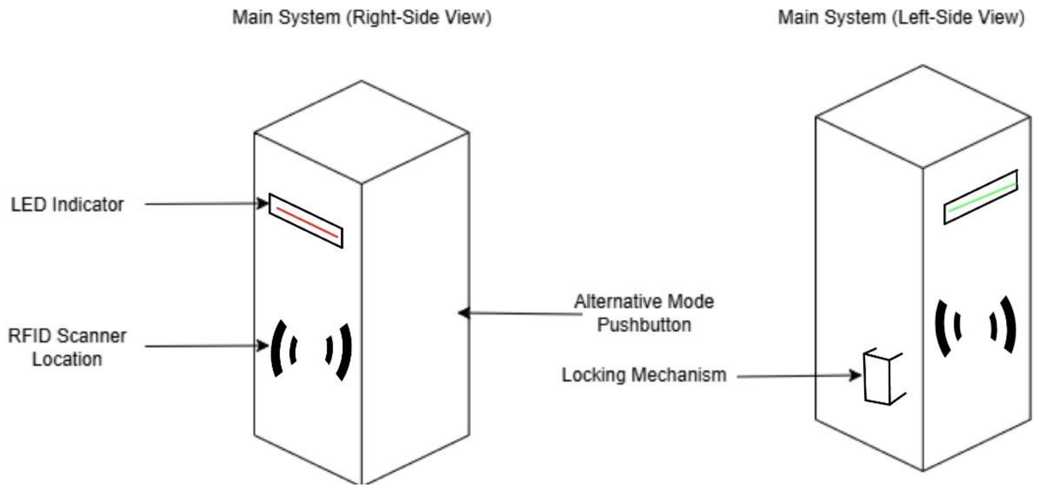
Main System (Right-Side View)
Main System (Left-Side View)
LED Indicator
RFID Scanner Location
Alternative Mode Pushbutton
Locking Mechanism

Figure 1: Physical Overview of RFID Door Lock System

Identify Input/Output Requirements

The RFID door lock system consists of multiple input and output components that work together to enable our secure and quick access control. The main input is the RFID reader, this detects and reads the UID of an RFID tag or fob that gets presented within a close range. Another important input is the manual mode push button, which when pressed, signals to the microcontroller to add or remove the next read RFID UID (depending on if it is stored or not). We will also be using the ARM A9 Private Timer to properly capture a certain timeframe so that a door only stays unlocked for a small period before relocking.

On the output side of our system, we will have a servo motor to unlock or lock our door based on the result of the read RFID UID. We will also have an LED indicator to display a bad read when an RFID is presented with an unauthenticated UID, this allows the user to know the door is working and they do not have proper access. Together these I/O components form our responsive system that can efficiently perform RFID access control.

To ensure robust and reliable communication, the RFID reader is connected to the microcontroller using a serial interface (such as UART or SPI) while also having level shifters as needed to make sure the signal levels match. The manual pushbutton is connected to a GPIO input pin, and we will use methods such as debouncing in our software to make sure we avoid any false triggers. The ARM A9 Private Timer is setup as an internal timer that will allow us to control the exact amount of time a door stays unlocked before it eventually re-locks again. The servo motor is controlled by a PWM signal from a dedicated digital output pin, and we will also have to make sure to select a microcontroller than can provide enough current to it. Our LED indicator is connected to another digital output pin, and this allows us to manually light up the LED when an unauthorized RFID UID is detected. Overall, we have planned all the connections carefully so that each component communicates smoothly with the microcontroller, allowing reliable operation of the entire system.

Initial Software Design

To begin we initialize all GPIO pins on the microcontroller, where the inputs pints are set for the RFID reader and a pushbutton while the output pins are properly configured for the servo motor that controls the locking mechanism as well as the LED indicators for the bad read signal. Once the hardware initialization is completed, the software enters the main loop which continuously polls the RFID read function for incoming tags and monitors the manual register/deregister button.

During each step of the loop, the system checks if an RFID tag has been detected. If a tag has been detected by the reader, it will capture the UID, and the microcontroller will store that value in a register. It will take that value it read and compare it against the array of currently stored UIDs in memory. If a match is found, the software will execute an unlock subroutine that sends a signal to the servo motor to physically unlock the door. It will then launch a second subroutine that starts the Private A9 Timer and waits for a set amount of time before launching the lock subroutine. If an unrecognized UID is presented, it will launch the LED subroutine to turn on the LED to signal the user that their ID isn't valid. Also, if the pushbutton is pressed the system will enter a registration mode which still polls as normal but instead of unlocking the door it will read the presented RFID and compare it to the array of UIDs in memory. If it exists already, it gets removed from the array however if it doesn't exist then it gets added.

Below is pseudo-assembly code for this process:

```
Start:
  //Initialize GPIO Pins
  Set RFID Reader as INPUT
  Set Pushbutton as INPUT
  Set Servo Motor Control as OUTPUT
  Set LED as OUTPUT
  Clear Registers
  b Main

Main:
  b Check_Button
  b Check_RFID
  cmp read_RFID with 1 // if 1, it read a UID
  bne Main
  cmp Lock_State with 1 //if 1, its in registration mode
  beq register_deregister_device
  b unlock_lock

Check_Button:
  LDR Lock_State, [Button_Addr] //read the button state
  b Main

Check_RFID:
  LDR read_RFID, [RFID_Addr] // read the RFID state
  LDR read_RFID_value, [RFID_Addr, #4] // read the RFID UID
  b Main

Register_deregister_device:
  LDRB stored_UIDs, [UIDs_Addr] //get the UIDs
  Compare stored_UIDs, read_RFID_value
  If contained, remove from [UIDs_Addr] //remove if from UID array if exists
  Else, add to [UIDs_Addr] //add to UID array if doesn't

Unlock_lock:
  LDRB stored_UIDs, [UIDs Addr]
  Compare stored_UIDs, read_RFID_value //get the UIDs
  If contained, write to [servo_motor_addr] //unlock the lock
  If contained, start the timer // start timer
  If not contained, write to [LED_addr] //turn on bad read LED
  If timer is finished, write to [server_motor_addr] // lock the lock

```

Prototyping Plan

This project will be prototyped on the DE10 Development Board. The DE10 does not have an RFID Reader, or a servo motor, so it will instead be used to prototype the software of the controller. Additionally, a detailed test plan will be implemented to verify each simulation's behaviour against expected outcomes. To facilitate this, some aspects of the project will be changed to emulate the requisite hardware with what is available on the DE10.

- Cards and reader: The DE10 does not have an RFID reader as aforementioned, but it does have switches and buttons. To quickly prototype without needing reader hardware or several cards with unique IDs, the switches on the DE10 and a push button will be used to simulate card tapping behavior. Test cases for this will include verifying that each switch configuration correctly maps to a unique simulated RFID UID and that the push button reliably triggers the simulated "card tap" event. The switches will control the ID of the simulated card while the button will represent holding the card to the reader.
- Servo: Door locking sequences can range from simple to incredibly intricate but actually hooking up the DE10 to a door or even a servo is not important for this project, as it is interchangeable and just a prerecorded sequence. Instead, to represent the door being "unlocked" and "locked," an LED will be used. The prototype will include a timing analysis to ensure that the LED's on/off transitions mimic the actual servo behavior closely. The unlocking sequence will consist of turning the LED on, and the locking sequence will consist of turning it off. These sequences can then be swapped out whenever a lock is actually implemented.
- Persistent storage: To speed up prototyping, and because the DE10 does not have any persistent storage, the project will not retain saved keys past a power outage. A set of benchmark tests will be run to ensure that key registration and deregistration occur without error during continuous

operation. This would be redundant at this stage anyway, as the software itself is also stored in volatile storage. There is no way to simulate persistent storage in software, so instead the hardware solution to simulate it will be to simply never switch off the device.

Overall, the prototyping plan includes detailed success criteria and specific test cases for each of our simulated components. This will allow our team to ensure the software behaves as expected even without having every needed piece of hardware. This approach we have taken to testing will also provide confidence in our software design, making sure all possible test cases have been thought of and designed around prior to integration with the full hardware system.

Selecting a Microcontroller

	DE10 Standard (Cortex-A9)	MSP430 (TI)	STM32F4 (ARM Cortex-M4)	ESP32 (Xtensa LX6)
Cost	High (~$200)	Low (~$10)	Medium (~$20)	Low (~$8)
Power Consumption	High (dual-core, 1.5 GHz)	Very Low (25 MHz)	Medium (180 MHz)	Medium (240 MHz)
Peripherals Included	GPIO, SPI, I2C, UART, ADC, DDR3	GPIO, SPI, I2C, UART, ADC	GPIO, SPI, I2C, UART, ADC, DAC, USB	GPIO, SPI, I2C, UART, ADC, Wi-Fi, Bluetooth

Since this project involves using multiple input and output pins, connectivity, and real-time processing, the STM32F4 microcontroller is the most ideal choice. It offers a good balance between performance and power consumption, making it suitable for the RFID door lock system. The board provides a rich set of peripherals, including GPIO, UART, I2C, SPI, ADC, and DAC, which are essential for interfacing with RFID readers and actuating locking mechanisms. Additionally, the STM32F4 family is known for its efficient power consumption and high clock speed, making it ideal for real-time applications.

When comparing the STM32F4 with the ARM Cortex-A9 on the DE10-Standard board, the STM32F4 is a much more power-efficient device and requires significantly less computational power to function effectively. The DE10-Standard is powerful but overkill for a door lock system, while the STM32F4 balances processing power and efficiency.

The MSP430 is also a viable alternative for extremely low-power scenarios, but it lacks the real-time processing power and peripheral richness that the STM32F4 offers. The ESP32 was considered for its connectivity features (Wi-Fi/Bluetooth), but it is not necessary for a basic RFID door lock system.

The STM32F4 board is cost-effective, typically priced around $20, according to Digi Key. This makes it significantly cheaper compared to the DE10-Standard, which costs around $200. In addition, the STM32F4 has excellent support for embedded C programming and real-time control, making it the most suitable choice for this project.

Revised Software Design

As the project progressed and we began working more closely with the DE10-Standard board and ARM Cortex-A9 processor, we identified several areas where our original software design could be improved. Initially, the system was designed using a polling-based approach, where the microcontroller continuously checked the state of inputs like the RFID reader and registration button. While this was a simple and effective way to implement logic, it became clear that there were more efficient ways to handle these interactions, particularly as we began exploring the ARM's timer capabilities and learning about interrupt-driven programming.

One major change in our revised software design was the integration of the ARM A9 Private Timer to manage automatic door relocking after a valid RFID tag was detected. This feature was originally conceptual but was successfully implemented during software development using polling logic in assembly. The timer starts after a successful unlock and once expired, triggers a routine to relock the door. Although this was achieved using polling, the experience highlighted how interrupts could improve this feature's efficiency, especially if the system were scaled up or needed to perform other tasks simultaneously.

Another key update was refining how the system manages UID registration and validation. The initial design stored UIDs and compared them during each read, but the implementation process helped us simplify this logic. We used a clearer memory management structure to store UIDs, enabling efficient add/remove functionality for registration and deregistration. The incorrect UID feedback mechanism was also enhanced during implementation, ensuring that the HEX display indicator provides immediate and accurate feedback when an unauthorized tag is presented.

Working with the ARM architecture and the DE10 board introduced practical challenges, particularly around hardware emulation. Since the board lacks an actual RFID reader or servo motor, we used switches to simulate RFID inputs and HEX displays to represent door states. This forced us to abstract the hardware logic, focusing purely on software design. Through this abstraction, we realized that although polling works in a controlled prototype, it is inefficient for real-time embedded systems, where responsiveness and power efficiency are critical.

Interrupts were not included in the original design, but after learning more about their advantages, we identified several key areas where they could enhance our system. For example, the registration button could trigger a GPIO interrupt, eliminating the need for constant monitoring. Similarly, an interrupt could be used to detect when a UID is received from the RFID reader, allowing the system to respond immediately without continuously checking the input register. The timer, too, could use an interrupt to initiate the relock routine, avoiding delays caused by waiting for the timer to expire through polling.

In conclusion, the revised software design benefits from a clearer understanding of the hardware, a more modular and optimized code structure, and a recognition of how interrupt-driven logic can improve performance. While the prototype used polling due to simplicity and hardware constraints, a full deployment would benefit greatly from integrating interrupts to enhance efficiency, responsiveness, and scalability. This revised approach demonstrates a more mature and embedded-aware system design, suitable for a real-world RFID door lock solution.

Results from Prototyping

The prototyping phase on the DE10-Standard board confirmed that the core software components of our RFID-based door lock system functions exactly as intended. We were able to implement the polling-based input handling scheme in which the system continuously monitors for a pushdown button (button 1) for mode switching to registration mode and for a pushdown from button 0 to simulate the reading of an RFID. When it reads the simulated UID, we use the switches to create 10-bit UID that the system works with and performs the needed actions following the reading. This approach allowed us to accurately simulate what a user interaction would be like with both the unlocking mechanism and the registration mechanism. Additionally, we were able to integrate memory-based UID registration and validation using a fixed-size array in our data segment where we can add or remove UIDs based on the current system state. By simulating various switch configurations, we confirmed that the system was able to correctly identify valid UIDs, triggering an unlock sequence, and was able to appropriately flag any invalid ones with error feedback from the HEX Display.

Our development process followed an iterative process. The first iteration of our build involved creating the basic polling loop with a static message on the HEX display. Subsequent iterations introduced UID validation with single memory location storage, followed by integration of the memory-based array, then the timer countdown logic and dynamic HEX display updates. In the final iteration, we went through our entire source code and modularized it into subroutines, implemented automatic clearing of the display after interactions, and added enhanced error handling. This step-by-step iterative approach we took allowed us to rigorously test and refine individual components before finishing the complete system.

Several challenges showed up during the prototyping process. To start, without any RFID hardware available, we had to pivot using the switches as our temporary version of UIDs. Secondly, the lack of a real servo motor made us repurpose the HEX display for indication of the lock status and errors. Moreover, managing the UID storage array in ARM assembly was extremely challenging, we originally only stored individual words but that become too complex to manage when adding more than two. So, by pivoting to an array, it made the code itself less complex than using 11 unique word storages, however, to have the array itself work properly was an extremely challenging task and took as long as the rest of the code in the design to finish.

Our testing strategy was methodical and thorough as we were missing two major components, the RFID hardware, and the servo motor. Because of this, the software itself had to be tested deeply to make sure that when those get integrated, they would function without any problems. We developed a long list of test scenarios that simulated various switch configurations, verifying that the software responded correctly to both valid and invalid UID inputs. Each software iteration was tested after its development finished, making sure we only moved onto the next iteration when the previous was thoroughly tested.

Although our prototype currently operates in a mainly simulated environment, a detailed debug plan is ready for when the physical hardware can be implemented. Starting with integration tests, upon connecting the actual RFID hardware, we will perform tests to verify that the signal between our microprocessor and the hardware is working and has no issues. Secondly, we have to perform iterative debugging again, where each main subsystem (UID registration, countdown, display updates, etc) will each be isolated and tested independently following each integration of hardware integration. Lastly, we will have to perform an environmental stress test to the full completed lock system to ensure that the RFID based door lock can be exposed to various environmental conditions and ensure the overall system's functionality remains robust during those times.

Lastly, the software sections for our prototype can be credited as the following. Xristopher Aliferis implemented the countdown logic, complete unlock/relock control, and the Private A9 timer's full integration. Carmel Kurland developed the UID registration and deregistration routines while also creating the full message encoding scheme needed for the HEX display. Lastly, Yash Padhiar built the display subroutines, the main polling loops, and design all rigorous testing. However, despite us three splitting those parts separately, we worked strongly together to completely integrate the memory-based array and not one person did that alone.

Conclusion

Based on the results from our prototyping and testing on the DE10-Standard board, we believe our RFID door lock system is highly feasible to finalize in hardware using a more application-focused microcontroller like the STM32F4. The design itself is simple, modular, and realistic to deploy in real-world scenarios. Most of the essential features—such as UID verification, timed locking/unlocking, and manual registration/deregistration—were successfully prototyped in software, with simulated inputs and outputs replacing actual hardware components. The estimated cost of the system, as outlined in our bill of materials (BOM) below, ranges from $60–$100 CAD, which is a reasonable price for a secure access control solution, especially considering that bulk production and hardware optimization would reduce these costs further.

Component	Description	Estimated Cost (CAD)
STM32F4 Development Board	(e.g., STM32F407 Discovery)	$27 – $34
RFID Reader Module	MFRC522 (SPI-based)	$4 – $7
RFID Tags / Key Fobs	Typically used for access	$0.70 – $1.30 each
Servo Motor / Door Actuator	For locking/unlocking mechanism	$7 – $13
HEX Displays & Resistors	Status indicators	<$1
Push Button	For registration toggle	<$1
Power Supply	USB power adapter or battery pack	$7 – $14
Prototyping Materials	Breadboard, wires, connectors	$7 – $14
Enclosure (optional)	Protective casing	$7 – $20

From a sustainability standpoint, this system holds up well. By replacing mechanical keys and potentially reducing reliance on more complex or internet-connected smart locks, our solution cuts down on e-waste associated with digital displays, Wi-Fi modules, and batteries that wear out faster. Most components in our system—such as the RFID reader, STM32 board, HEX Displays, and servo—are relatively low-power and commonly used, meaning that they are mass-produced and can be sourced from manufacturers that follow RoHS (Restriction of Hazardous Substances) guidelines. In terms of energy efficiency, the system only actively consumes power when a card is being scanned or the door mechanism is actuated, which makes it practical for both battery-operated and wired installations.

Environmental concerns mainly revolve around electronic waste generated at the end of the product's lifecycle. However, many of the components are reusable or recyclable (such as the microcontroller and the RFID modules), and the absence of a display or continuous wireless connectivity keeps the energy consumption low during use. Moving forward, designing the casing and packaging with recyclable or biodegradable materials could further minimize environmental impact. Additionally, while the servo motor was not physically implemented during prototyping, the system was designed to support a generic servo motor rather than a proprietary actuator, which would make the final hardware easier to repair and maintain.

This project has been incredibly valuable in connecting the theoretical knowledge from the course with a real embedded systems application. It forced us to think critically about how input/output devices interact with software, how polling and interrupts differ in embedded control, and how to balance hardware constraints with user expectations. We learned how to simulate hardware like RFID readers and servo motors using available DE10-Standard components, and how to implement and structure the software logic in assembly using GPIO and timer functionality. Beyond that, we developed a much stronger understanding of how to optimize for simplicity and efficiency in embedded applications.

Perhaps the most important takeaway is that embedded systems don't need to be overly complex to be impactful. Through this project, we saw how even a simple and relatively inexpensive combination of sensors and logic can offer a highly effective solution to a common real-world problem. Additionally, we deepened our familiarity with ARM-based microcontrollers and recognized the value of choosing the right microcontroller for the job—not always the most powerful one, but the most efficient and appropriate. Overall, this project helped bridge the gap between classroom knowledge and practical embedded system development, preparing us to approach future projects with greater technical confidence and systems-level thinking.

Appendix

References

Kasim, Bukhari, et al. “Performance Analysis of RFID-Based Smart Door Lock Controlled by Arduino.” *Journal of Advanced Research in Applied Mechanics*, vol. 122, no. 1, 30 July 2024, pp. 163–174, https://doi.org/10.37934/aram.122.1.163174. Accessed 2 Sept. 2024.

Putra, Eka, et al. “Service Quality Analysis of RFID-Based Smart Door Lock in Front One Azana Style Hotel Area.” *Brilliance Research of Artificial Intelligence*, vol. 4, no. 1, 30 July 2024, pp. 372–381, https://doi.org/10.47709/brilliance.v4i1.4292.

“Soc Platform - Cyclone - DE10-Nano Kit.” *Terasic*, www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=1046. Accessed 27 Mar. 2025.

“MSP430 Microcontrollers.” *TI.Com*, www.ti.com/microcontrollers-mcus-processors/msp430-microcontrollers/overview.html. Accessed 27 Mar. 2025.

“STM32F4 Series.” *STMicroelectronics*, www.st.com/en/microcontrollers-microprocessors/stm32f4-series.html. Accessed 27 Mar. 2025.

“ESP32.” *ESP32 Wi-Fi & Bluetooth SoC | Espressif Systems*, www.espressif.com/en/products/socs/esp32. Accessed 27 Mar. 2025.

Source Code

```
.global _start

.data
@ 7-segment display codes for digits 0-9.
hex_number_array:
    .byte 0b00111111
    .byte 0b00000110
    .byte 0b01011011
    .byte 0b01001111
    .byte 0b01100110
    .byte 0b01101101
    .byte 0b01111101
    .byte 0b00000111
    .byte 0b01111111
    .byte 0b01101111
```

```

.align 4
@ Data array holding various display messages and characters.
hex_data_array:
    .word 0x3F737937      @ Encoded "OPEN"
    .word 0x383F6D79      @ Encoded "LOSE"
    .word 0x39            @ Encoded letter C
    .word 0x50793D        @ Encoded message part (reg)
    .word 0x5E5E795E      @ Encoded message part (ddEd)
    .word 0x77            @ Encoded letter A
    .word 0x38797750      @ Encoded message part (LEAr)
    .word 0x39            @ Encoded letter C
    .word 0x50505C50      @ Encoded message part (rror)
    .word 0x79            @ Encoded letter E
    .word 0x5079775E      @ Encoded message part (rEAD)

.align 4
@ Arrays and storage for UID management and timing.
uid_array:
    .word 0,0,0,0,0,0,0,0,0,0
uid_array_length:
    .word 0
time_storage:
    .word 0

@ Base addresses (pointers) for key data structures.
UID_BASE:              .word uid_array
UID_LENGTH_BASE:       .word uid_array_length
hex_data_array_addr:  .word hex_data_array
hex_number_array_addr: .word hex_number_array
TIME_BASE:             .word 0

.text
.align 4
@ Define hardware peripheral addresses.
.equ SW_BASE, 0xFF200040       @ Switches (simulate RFID input)
.equ LED_BASE, 0xFF200000      @ LED base address (unused in current code)
.equ HEX3_HEX0_BASE, 0xFF200020 @ Lower half of 7-segment displays
.equ HEX5_HEX4_BASE, 0xFF200030 @ Upper half of 7-segment displays
.equ BUTTON_BASE, 0xFF200050    @ Buttons input base address
.equ TIMER_BASE, 0xFFFEC600     @ Timer registers base address

_start:
    @ Initialize hardware pointers and load addresses of key data arrays.
    ldr   r9, =SW_BASE
    ldr   r8, =BUTTON_BASE
    ldr   r7, =hex_data_array_addr
    ldr   r7, [r7]          @ r7 now holds the address of hex_data_array
    ldr   r6, =TIMER_BASE
    ldr   r5, =hex_number_array_addr
    ldr   r5, [r5]          @ r5 now holds the address of hex_number_array

    @ Configure the timer to produce a 1ms tick (assuming a 200 MHz clock).
    ldr   r0, =200000       @ 200000 cycles per millisecond
    str   r0, [r6]          @ Load timer with tick count
    mov   r0, #1
    str   r0, [r6, #12]     @ Clear timer interrupt flag
    mov   r0, #0b011
    str   r0, [r6, #8]      @ Enable timer with auto-reload

    b _main_loop          @ Jump to the main polling loop

_main_loop:
    @ Poll the BUTTON_BASE to determine the operation mode.
    ldr   r4, =BUTTON_BASE
    ldr   r0, [r4]
    cmp   r0, #2          @ Check if button for registration mode is pressed
    beq _registration_mode_switch
    cmp   r0, #1          @ Check if button for UID read is pressed
    bne _main_loop      @ Continue polling if neither button is active

    @ Simulate a UID read by fetching the value from the switches.
    ldr   r4, =SW_BASE
    ldr   r0, [r4]
    ldr   r2, =0xFFFFFFFF  @ Default "not found" flag value for UID index
    bl _check_uid_exists   @ Look up UID in the memory array
    ldr   r2, =0xFFFFFFFF  @ Reset "not found" flag value
    cmp   r1, r2          @ Check if UID was not found
    beq _display_error    @ Display error message if UID is absent
    b _unlock_lock        @ Otherwise, proceed to unlock

_registration_mode_switch:
    @ In registration mode: display an alternate message and handle registration.
    bl _display_alt
    b _register_deregister_device

_display_alt:
    @ Display an alternate message on the HEX display for registration mode.
    push {r4, lr}
    ldr   r1, [r7, #12]   @ Retrieve the alternate message code from hex_data_array
    ldr   r4, =HEX3_HEX0_BASE
    str   r1, [r4]
    ldr   r0, =1000      @ Delay for 1 second
    bl _delay
    bl _clear_hex_display
    pop {r4, lr}
    bx lr

_register_deregister_device:
    @ Wait for further button press during registration mode.
    ldr   r4, =BUTTON_BASE
    ldr   r0, [r4]
    cmp   r0, #2        @ Check if exit registration button is pressed
    beq _read_loop
    cmp   r0, #1        @ Check if UID input button is pressed
    bne _register_deregister_device

    @ Read UID from switches and check if it already exists.
    ldr   r4, =SW_BASE
    ldr   r0, [r4]
    bl _check_uid_exists
    ldr   r4, =0xFFFFFFFF
    cmp   r1, r4        @ Compare with "not found" flag
    beq _add_to_memory  @ Add UID if not present
    b _remove_from_memory  @ Otherwise, remove it

_read_loop:
    @ Display a message indicating a UID read operation.
    bl _display_read
    b _main_loop

_display_error:
    @ Display an error message by writing to both HEX display registers.
    ldr   r1, [r7, #32]   @ Load first part of error message
    ldr   r4, =HEX3_HEX0_BASE
    str   r1, [r4]
    ldr   r1, [r7, #36]   @ Load second part of error message
    ldr   r4, =HEX5_HEX4_BASE
    str   r1, [r4]
    ldr   r0, =2000      @ Keep error message on display for 2 seconds
    bl _delay
    bl _clear_hex_display
    b _main_loop

_display_added:
    @ Show a message indicating that a new UID was successfully added.
    push {r4}
    ldr   r1, [r7, #16]   @ Retrieve first part of the added message
    ldr   r4, =HEX3_HEX0_BASE
    str   r1, [r4]
    ldr   r1, [r7, #20]   @ Retrieve second part of the added message
    ldr   r4, =HEX5_HEX4_BASE
    str   r1, [r4]
    ldr   r0, =2000      @ Delay for 2 seconds
    pop {r4}
    bl _delay
    bl _clear_hex_display
    b _main_loop

_display_read:
    @ Show a message to indicate a UID read event.
    push {r4}
    ldr   r1, [r7, #40]   @ Get read operation message code
    ldr   r4, =HEX3_HEX0_BASE
    str   r1, [r4]
    ldr   r0, =1000      @ Delay for 1 second
    pop {r4}
    bl _delay
    bl _clear_hex_display
    b _main_loop

_display_clear:
    @ Display a clear message on the HEX displays.
    push {r4}
    ldr   r1, [r7, #24]   @ Load first part of clear message
    ldr   r4, =HEX3_HEX0_BASE
    str   r1, [r4]
    ldr   r1, [r7, #28]   @ Load second part of clear message
    ldr   r4, =HEX5_HEX4_BASE
    str   r1, [r4]
    ldr   r0, =2000      @ Delay for 2 seconds
    pop {r4}
    bl _delay
    bl _clear_hex_display
    b _main_loop

_check_uid_exists:
    @ Look up a UID in the stored uid_array.
    @ Returns the index in r1 if found, or 0xFFFFFFFF if not found.
    push {r4, r5, r6, r7, r8, r9}
    mov   r2, #0          @ Start index at 0
check_loop:
    ldr   r4, =UID_LENGTH_BASE
    ldr   r4, [r4]
    ldr   r1, [r4]        @ Get the current UID count
    cmp   r2, r1
    beq   not_found       @ If reached UID count, UID is not present
    ldr   r4, =UID_BASE
    ldr   r4, [r4]
    ldr   r1, [r4, r2, LSL #2] @ Fetch UID at current index
    cmp   r0, r1          @ Compare with input UID
    beq   found           @ If equal, UID exists
    add   r2, r2, #1      @ Increment index
    b check_loop
found:
    mov   r1, r2          @ Return index of found UID
    pop {r4, r5, r6, r7, r8, r9}
    bx lr
not_found:
    mov   r1, #0xFFFFFFFF @ Return flag indicating UID not found
    pop {r4, r5, r6, r7, r8, r9}
    bx lr

_add_to_memory:
    @ Add a new UID to the uid_array if there is available space.
    push {r4}
    ldr   r4, =UID_LENGTH_BASE
    ldr   r4, [r4]
    ldr   r1, [r4]
    cmp   r1, #10        @ Check if maximum number of UIDs (10) is reached
    bleq _display_error @ If full, display an error message

    @ Store the new UID at the current end of the array.
    ldr   r4, =UID_BASE
    ldr   r4, [r4]
    str   r0, [r4, r1, LSL #2]
    add   r1, r1, #1     @ Increment the UID count
    ldr   r4, =UID_LENGTH_BASE
    ldr   r4, [r4]
    str   r1, [r4]       @ Update the stored UID count

    bl _display_added   @ Show confirmation that UID was added
    ldr   r0, =2000     @ Delay for 2 seconds before clearing display
    bl _delay
    bl _clear_hex_display

    pop {r4}
    b _main_loop

_remove_from_memory:
    @ Remove a UID from the uid_array by shifting the remaining entries.
    push {r4}
    ldr   r4, =UID_LENGTH_BASE
    ldr   r4, [r4]
    ldr   r1, [r4]
    ldr   r4, =UID_BASE
    ldr   r4, [r4]
    ldr   r3, [r4, r2, LSL #2] @ Retrieve the UID to be removed

    cmp   r0, r3
    mov   r0, r2
    bleq _shift_uid_array @ If found, begin shifting the array
    add   r2, r2, #1
    cmp   r2, r1
    beq _main_loop      @ If UID not found in remaining entries, return to main loop

    cmp   r0, r3
    bne _remove_from_memory
    pop {r4}
    b _main_loop

_unlock_lock:
    @ Unlock the system by displaying an "OPEN" message followed by a countdown.
    ldr   r2, [r7]      @ Get the "OPEN" message code from hex_data_array
    ldr   r4, =HEX3_HEX0_BASE
    str   r2, [r4]
    ldr   r0, =2000     @ Display the open message for 2 seconds
    bl _delay

    @ Initialize a 5-second countdown (5000 milliseconds).
    ldr   r8, =5000     @ r8 holds the remaining countdown time in ms

countdown_loop:
    @ Divide the remaining milliseconds by 1000 to obtain the full seconds.
    mov   r0, r8
    ldr   r1, =1000
    bl unsigned_div   @ r0 will contain the quotient (seconds)
    mov   r9, r0      @ r9 now holds the seconds to display

    @ Get the address of the 7-segment display codes.
    ldr   r10, =hex_number_array_addr
    ldr   r10, [r10]

    cmp   r9, #10     @ Check if the seconds equal 10
    beq load_10
    @ For seconds less than 10, load the corresponding 7-seg code.
    mov   r0, r9
    ldrb  r0, [r10, r0]
    b display_digit

load_10:
    @ For 10 seconds, combine two 7-seg codes to display "10".
    ldrb  r0, [r10, #1]
    lsl   r0, r0, #8
    ldrb  r1, [r10, #0]
    add   r0, r0, r1

display_digit:
    @ Display the computed 7-seg code.
    ldr   r4, =HEX3_HEX0_BASE
    str   r0, [r4]
    mov   r0, #1000   @ Wait for 1 second
    bl _delay

    @ Decrease the countdown by 1 second (1000 ms) and loop if time remains.
    sub   r8, r8, #1000
    cmp   r8, #0
    bgt countdown_loop

    @ After the countdown, display the closing "LOSE" message.
    ldr   r2, [r7, #4]   @ Get first part of "LOSE"
    ldr   r4, =HEX3_HEX0_BASE
    str   r2, [r4]
    ldr   r2, [r7, #8]   @ Get second part of "LOSE"
    ldr   r4, =HEX5_HEX4_BASE
    str   r2, [r4]
    ldr   r0, =2000     @ Delay for 2 seconds
    bl _delay
    bl _clear_hex_display

    b _main_loop

_delay:
    @ Delay subroutine: pauses execution for a specified number of milliseconds.
    @ Expects the desired delay (in ms) in r0.
    push {r4, r5, r6, lr}
    mov   r4, r0         @ Save the target delay in r4

    @ Reset the millisecond counter.
    ldr   r5, =time_storage
    mov   r0, #0
    str   r0, [r5]

    @ Configure the timer for 1ms ticks.
    ldr   r6, =TIMER_BASE
    ldr   r0, =200000
    str   r0, [r6]
    mov   r0, #1
    str   r0, [r6, #12]
    mov   r0, #0b011
    str   r0, [r6, #8]

_delay_loop:
    ldr   r0, [r6, #12] @ Poll the timer's interrupt flag.
    cmp   r0, #1
    bne _no_update
    @ On timer tick, clear the flag and increment our counter.
    mov   r0, #1
    str   r0, [r6, #12]
    ldr   r1, [r5]
    add   r1, r1, #1
    str   r1, [r5]
_no_update:
    @ Continue looping until the target delay is reached.
    ldr   r0, [r5]
    cmp   r0, r4
    blt _delay_loop

    pop {r4, r5, r6, lr}
    bx lr

_clear_hex_display:
    @ Clear both HEX display registers.
    push {r4}
    mov   r1, #0
    ldr   r4, =HEX3_HEX0_BASE
    str   r1, [r4]
    ldr   r4, =HEX5_HEX4_BASE
    str   r1, [r4]
    pop {r4}
    bx lr

_shift_uid_array:
    @ Remove a UID from the uid_array by shifting subsequent entries left.
    push {r1, r2, r3, r4, lr}
    ldr   r4, =UID_LENGTH_BASE
    ldr   r4, [r4]
    ldr   r3, [r4]
    subs  r3, r3, #1     @ Decrement UID count
    str   r3, [r4]
    mov   r1, r0        @ r1 holds the index to remove
shift_loop:
    cmp   r1, r3
    bge shift_done
    add   r2, r1, #1
    ldr   r4, =uid_array
    ldr   r0, [r4, r2, LSL #2]
    str   r0, [r4, r1, LSL #2]
    add   r1, r1, #1
    b shift_loop
shift_done:
    bl _display_clear
    pop {r1, r2, r3, r4, lr}
    bx lr

unsigned_div:
    @ Perform an unsigned division of r0 by r1.
    @ Returns the quotient in r0 (remainder in r1 is ignored).
    push {r2}
    mov   r2, #0         @ Initialize quotient counter
    cmp   r1, #0
    beq div_end        @ Avoid division by zero
div_loop:
    cmp   r0, r1
    blt div_end
    sub   r0, r0, r1
    add   r2, r2, #1
    b div_loop
div_end:
    mov   r0, r2        @ Return the quotient in r0
    pop {r2}
    bx lr

STOP:
    b STOP


```