RFID ID Reader & Recorder (RIDRR)

ELEX 7820 Real-Time Embedded Systems

Final Project Report

Last Modified Date: 12/1/2025

For:

Ed Casas, Ph.D., P.Eng.

By:

Ryan McKay, Yizuo Chen

1 Abstract

This report describes the design and implementation of the RFID ID Reader and Recorder (RIDRR) system developed for the ELEX 7820 Real-Time Embedded Systems. The device uses a PN532 NFC reader and a Raspberry Pi Pico 2W running FreeRTOS to record student attendance through NFC card scanning. When a student taps their ID card, the system reads the UID, provides LED and buzzer feedback, and stores the UID in on-board flash memory. At the end of the session, all stored UIDs can be transferred to a computer over USB. This report explains the hardware, software architecture, task structure, flash-writing methods, etc. The final RIDRR prototype performed reliably and met all project requirements.

Contents

2 Introduction

The goal of this project is to build a reliable and easy-to-use attendance system based on NFC card scanning. The system is designed to demonstrate real-time embedded programming concepts, including multitasking under FreeRTOS, interrupt handling, device drivers, and non-volatile data storage.

We chose this project because the idea of working with NFC readers seemed cool and our venerable instructor Lord Professor Eduardo F. Casas recommended the attendance system as a useful project to complete.

The project consists of three main parts:

- **NFC Reader Subsystem** – uses the PN532 module to read card UIDs.
- **FreeRTOS-Based Embedded Software** – manages scanning, feedback, timing, data handling, and flash writing using multiple tasks.
- **Flash Storage and USB Output** – stores attendance data and transfers it to a computer when needed.

This report provides a clear description of the hardware components, software structure, NFC subsystem, and flash-writing process. It also explains how each part works and how a future student can rebuild the entire system from the ground up.

3 System Overview

The system waits for a student card to be scanned, extracts the UID, gives feedback to the user, and stores the data. Figure 1 shows a simplified flow of how the device operates.

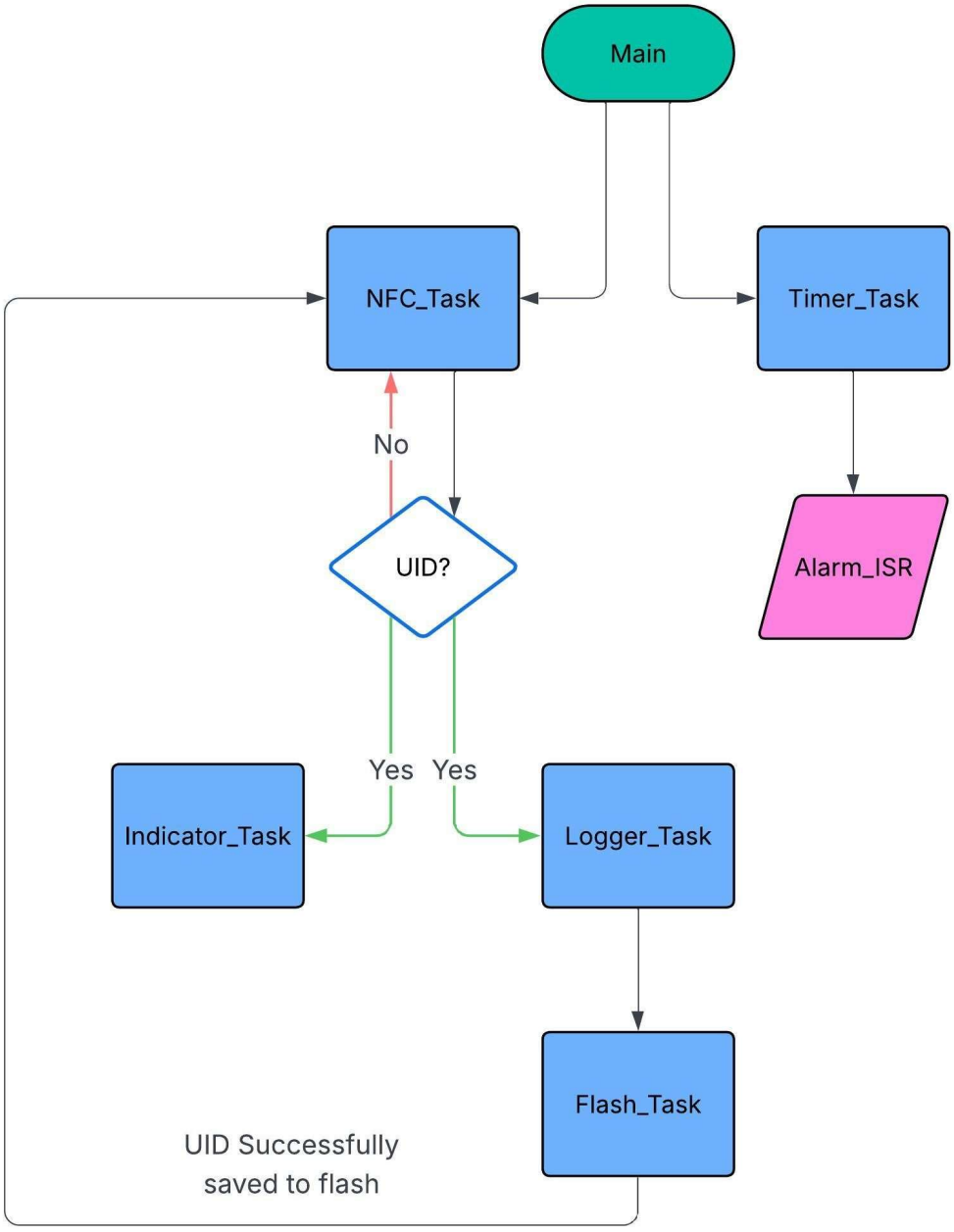

Figure 1: System Task Flow for NFC Attendance Recorder

3.1 Main Components

1. Raspberry Pi Pico 2W

Acts as the microcontroller running FreeRTOS. It handles NFC communication, LED/buzzer control, flash memory writes, and USB output.

2. PN532 NFC Reader

Reads ISO14443A cards (including BCIT student IDs) at 13.56MHz. Connected to the Pico using I^2C.

3. LEDs and Buzzer

Provide real-time feedback:

- Green LED/buzzer: successful scan
- Red LED: failed or invalid scan

4. On-Board Flash Memory

Stores all attendance UIDs until they are exported.

4 Software Architecture

The program runs on FreeRTOS and uses multiple tasks to keep the system responsive and organized.

4.1 Task List

1. **NFC Task** Continuously checks for cards. When a UID is detected, it sends it to other tasks.
2. **Indicator Task** Controls LEDs and the buzzer to confirm a successful read.
3. **Logger Task** Receives the UID and prepares it for writing.
4. **Flash Task** Stores the UID in non-volatile flash memory.
5. **Timer Task** Generates an ISR, periodic timing signals for real-time timing demonstrations. The ISR triggers every 100µs.

4.2 Supporting Software

In addition to the C code used to implement the RTOS system, we have provided 2 separate python scripts to control the basic functions of the system.

1. **Dump_to_csv.py** sends the “dump” command to the Pico over serial and records the UIDs along with the (currently just placeholders) time and dates. The python script creates a .csv file in the current folder called attendance with a unique date and time code appended to name of the file.

2. **clear_logs.py** sends the “clear” command over serial to remove all the saved information from flash memory.

5 System Operation

1. System powers on and initializes FreeRTOS tasks.
2. NFC task waits for a student card.
3. Student taps their ID card.
4. PN532 reads the UID and sends it to the Pico.
5. If valid:
    - LED + buzzer activate
    - UID is passed to the logger task
    - Flash task writes the UID to memory
6. At the end of the class, the user connects the device over USB.
7. The collected UIDs are transferred to the computer using the supplied Python code.

6 Code Module Hierarchy

```
PicoNFC System
|
+-- Application Layer
|     +-- main.c
|     +-- FreeRTOS Tasks
```

```
   |             +-- NFC_Task
   |                   +-- Indicator_Task
   |               +-- Logger_Task
   |               +-- Flash_Task
   |               +-- Timer_Task
   |             +-- USB_Task
   |
   |
  +-- Attendance Subsystem
|              +-- attendance_helpers.c / .h
|                  +-- UID formatting
|                  +-- buffer handling
|                   +-- flash record helpers
   |
   |
   |
  +-- NFC Core Layer
|              +-- piconfc.h / piconfc.c
|               | +-- piconfc_init()
|               | +-- piconfc_tagPresent()
|              | +-- piconfc_readNTAG()
|         |
|         |
|         |
|             +-- PN532 Driver (I2C)
|               | +-- piconfc_PN532.h / .c
|               | +-- firmwareVersion()
|               | +-- SAMConfiguration()
|               | +-- readPassiveTargetID()
|              | +-- inDataExchange()
|         |
|         |
|         |
|             +-- I2C Transport Layer
```

```
|                  | +-- piconfc_I2C.h / .c
|               | +-- sendcommand()
|                | +-- waitready()
|                | +-- readack()
|                | +-- parseresponse()
|          |
|          |
|          |
|          |
|          |
|          |
|           +-- NTAG Wrapper
|                 | +-- piconfc_NTAG.h / .c
|                | +-- page read/write
|          |
|          |
|          |
|            +-- NDEF Parser
|                      +-- piconfc_NDEF.h / .c
|                      +-- payload extraction
   |
   |
   |
   +-- Support / Config +--
   FreeRTOSConfig.h
   +-- lwipopts.h
   +-- CMakeLists + linker script
```

7 Hardware Details

7.1 Connections

Below is an example of the wiring between the PN532 module and the Raspberry Pi Pico 2W:

- MO/SDA/TX → GP4
- NSS/SCL/RX → GP5
- SVDD → 3.3V
- RST → 3.3V
- SIGIN → 5V
- 5V → 5V
- GND → GND

The LED and buzzer connect to GPIO pins 16 and 13 configured as digital outputs. GPIO pin 17 is used for measuring the time interrupts are down during flash operations (covered in the next section).

8 Flash Storage and USB Transfer

The Pico's internal flash memory is used to store UID values. A FreeRTOS mutex ensures safe access without data corruption. A simple USB serial interface is used to transfer the stored data to a laptop. The user can request a dump of all stored UIDs, which is printed over serial and transferred into a .csv file by the provided dump_to_csv.py script.

Flash Writing on RP2350: The RP2350 cannot execute code from flash while performing erase or write operations. For this reason, the Pico SDK temporarily disables all interrupts and the function of the second core during flash programming to prevent the CPU from jumping into an ISR located in flash memory. To reduce the impact on real-time performance, the system stores UIDs in RAM first and writes them to flash in short bursts. In the figures below we show the impact that these writes have on the performance of the system and its efficacy as a RTOS.

When writing to a page in flash the operation disables interrupts and the other core for a consistent time of 496µs this equates to roughly five of the 100µs interrupts. During this time the system is unavailable and thus fails as a reliable RTOS device. This issue could have been solved by moving the interrupt vector table and all functions into RAM before performing any flash writes; however, that information did not come to our attention until too late in the design process.

It should be noted that this downtime has no effect on the observable operation of the device. The flash writes happen at a speed that is not noticeable to users and will not interfere with card scanning behaviour.

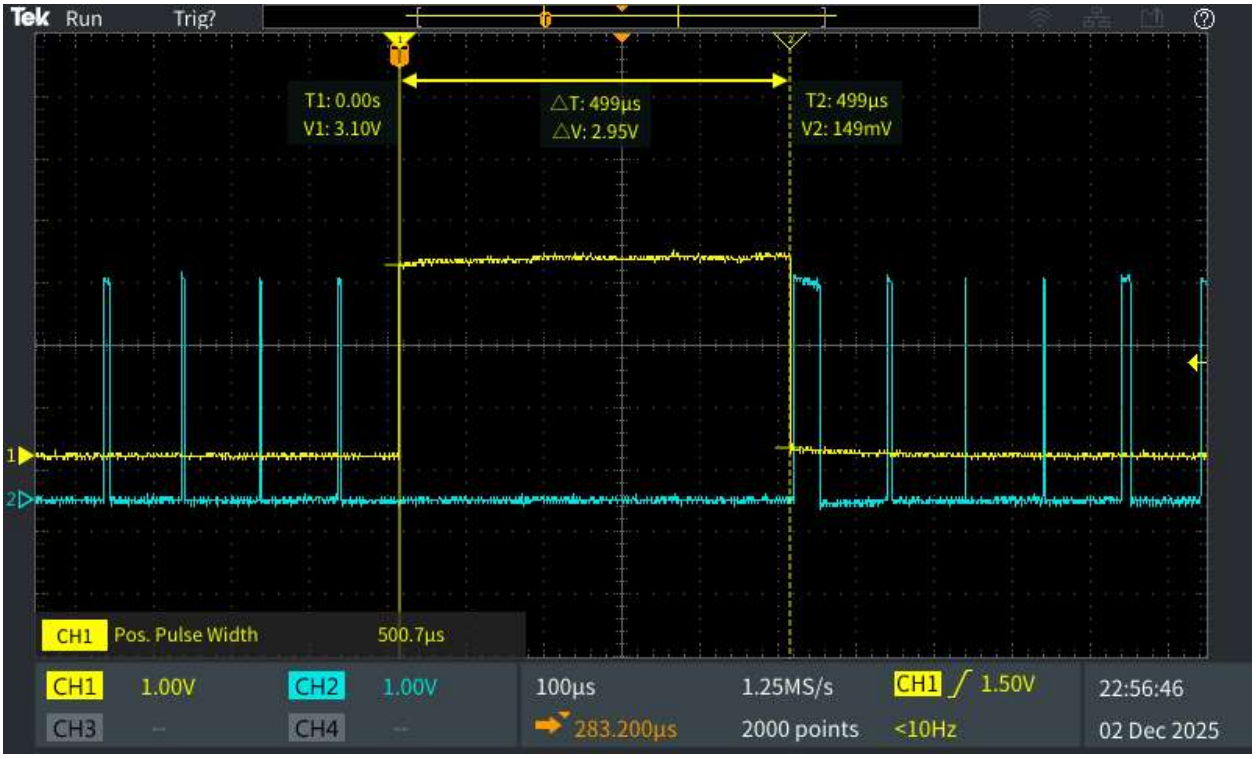

Figure 2: Interrupt downtime during flash write

When clearing the flash logs the operation stops the interrupts just the same as writing. This operation lasts nearly 20 times (19.72x) longer than the write operation. This is due to the write and erase operations using different minimum memory sizes; the writes use only a page of memory (256 bytes) while the erases use a sector (16 pages or 4096 bytes). This means the erase operation takes nearly 10ms, far too long for a real time system; however, this is not nearly as important as the writes since the erase operation is only ever called by a user and will therefore not be called when the system should be scanning in new IDs.

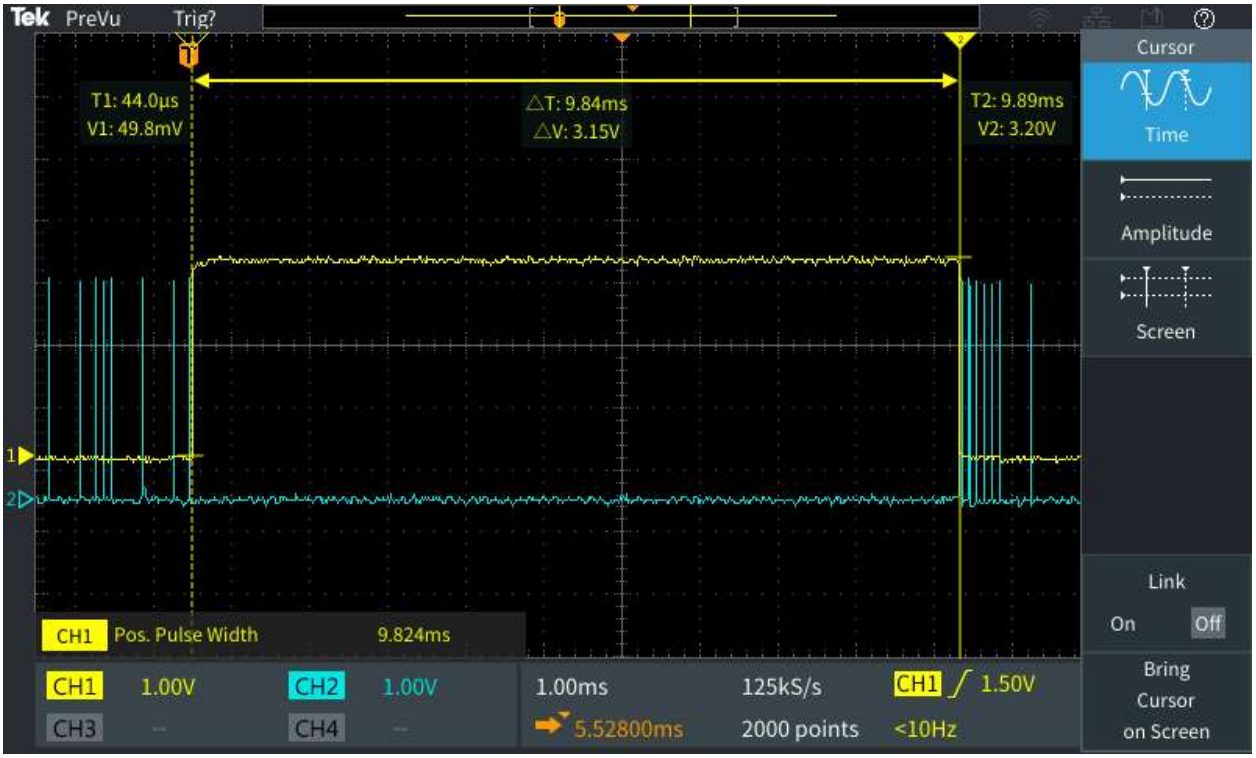

Figure 3: Interrupt downtime during flash erase

9 Design Improvements and Future Work

Although the current system meets the project requirements (as best as we could), several improvements could make the design more robust, more reliable, and easier to use in a real classroom environment.

1. Wear Levelling for Flash Storage

The RP2350's on-board flash has a limited number of erase cycles per sector. Currently, all UID records are written to a fixed flash region. Adding a simple wear-levelling scheme – such

as rotating through several sectors or using a log-structured write format – would reduce flash wear and extend the life of the device.

2. Physical Enclosure

The prototype is fully functional but exposed on a breadboard. A 3D-printed enclosure with proper mounting for the PN532 reader, LEDs, and buzzer would make the system durable and easier to handle in a classroom setting.

3. Move Critical Code to RAM for Flash-Safe Operation

Because the RP2350 executes code directly from flash (XIP), flash erase and write operations temporarily block instruction fetches. This can pause the system or prevent interrupt handlers from running.

A future improvement would be to move time-critical routines or the vector table to RAM using the `not in flash func()` attribute. This allows the system to continue servicing interrupts even during flash operations, making the design more compatible with real-time requirements.

4. Expand Flash Area, Fix RAM Storage Behaviour

Logged user IDs are currently capped at 1024, this was done mainly for testing purposes. Currently, on initialization, the program makes a copy of all UIDs in flash into RAM memory and keeps a copy of all newly scanned UIDs in RAM as well. This leads to a bloated cache of values that are already safely in RAM. Further iterations of the program will remove this otherwise useless behavior.

Additionally, looking at the memory map on initialization (in the Appendix) shows the large amount of flash that is neither being used by the program nor the storage of UIDs. Allotting the majority of this space to the storage of UIDs and removing the unnecessary storage of UIDs in RAM would allow us to increase the MAX_RECORDS from 1024 to a much, much larger value.

5. Additional Enhancements

Other possible improvements include:

- Adding CRC or checksum validation to stored UID data.
- Encrypting attendance records before saving them to flash.

- Implementing Wi-Fi or Bluetooth export instead of USB.
- Adding a small OLED screen to show attendance count in real time.

10 Conclusion

The RIDRR system successfully demonstrates NFC scanning, real-time multitasking, flash storage, and USB communication on the RP2350. The design follows clear hardware and software principles and can be reproduced by other students using the instructions in this report. The prototype meets all project requirements and provides a strong basis for future extension, including wireless uploading and improved durability.

Future improvements may include Wi-Fi uploading, a full web dashboard, or encrypted UID storage for privacy.

Appendix

Partitioning Flash Memory

Allegedly there are multiple ways to partition the flash memory and setup a safe zone for the new information. This is how we did it.

Following this link led to a helpful guide on reserving memory in the flash structure of the Pico.

Raspberry Pico C SDK: reserve a Flash memory block for persistent storage

By: Jan Cumps Date: 30 Aug 2023

https://community.element14.com/products/raspberry-pi/b/blog/posts/raspberry-pico-c-sdk-reserve-a-flash-memory-block-for-persistent-storage

Here is the quick rundown:

Find the memmap_default.ld file. Mine was in this location:

C:\Users\myName\.pico-sdk\sdk\2.2.0\src\rp2_common\pico_crt0\rp2350

Ensure you choose the file for the correct core (2350 vs 2040), then find this section:

```
23
24  MEMORY
25  {
26      INCLUDE "pico_flash_region.ld"
27      RAM(rwx) : ORIGIN =  0x20000000, LENGTH = 512k
28      SCRATCH_X(rwx) : ORIGIN = 0x20080000, LENGTH = 4k
29      SCRATCH_Y(rwx) : ORIGIN = 0x20081000, LENGTH = 4k
30  }
```

And change it to this: (Change the __PERSISTENT_STORAGE_LEN to whatever value you desire).

```
24  __PERSISTENT_STORAGE_LEN = 256k;
25
26  MEMORY
27  {
28      FLASH(rx)          : ORIGIN = 0x10000000,
29                           LENGTH = 2048k - __PERSISTENT_STORAGE_LEN
30      FLASH_PERSISTENT(rw) : ORIGIN = 0x10000000 + (2048k - __PERSISTENT_STORAGE_LEN),
31                             LENGTH = __PERSISTENT_STORAGE_LEN
32      RAM(rwx)           : ORIGIN = 0x20000000, LENGTH = 256k
33      SCRATCH_X(rwx)     : ORIGIN = 0x20040000, LENGTH = 4k
34      SCRATCH_Y(rwx)     : ORIGIN = 0x20041000, LENGTH = 4k
35  }
36
```

Save that in a custom .ld file (we named ours memmap_custom.ld) and point your makefile toward it like below. This will ensure your system is compiled with the top area of flash memory reserved for whatever you'd like to put there.

```
# ------------------------------------------------------------------
# Custom linker script to reserve flash for data storage
# ------------------------------------------------------------------

pico_set_linker_script(lab3 ${CMAKE_SOURCE_DIR}/memmap_custom.ld)
```

List of our custom files

These are the files where you can find the code we are responsible for.

1. lab3.c
2. attendance_helpers.h
3. attendance_helpers.c
4. memmap_custom.ld (very minimal tweaks)
5. CMakeLists.txt
6. dump_to_csv.py
7. clear_logs.py

NFC Library

The library we used to interface with the PN532 NFC reader is by MatthewIsHere posted on github. It was made specifically for the RP 2040, but we modified it to work with the RP 2350.

https://github.com/MatthewIsHere/piconfc

Last updated Nov. 11 2024.

Testing

Memory Locations

To ensure the logs were saving in the correct position we printed the log memory locations included with the “dump” command. This allowed us to confirm the information was saving in the correct position and not overwriting our programs.

```
Log[0] @ offset 0x003c0000 (XIP 0x103c0000): UID=04737E3A661990000000 TIME=00:00:00 DATE=YYYY-MM-DD
Log[1] @ offset 0x003c0020 (XIP 0x103c0020): UID=04737E3A661990000000 TIME=00:00:00 DATE=YYYY-MM-DD
Log[2] @ offset 0x003c0040 (XIP 0x103c0040): UID=04737E3A661990000000 TIME=00:00:00 DATE=YYYY-MM-DD
Log[3] @ offset 0x003c0060 (XIP 0x103c0060): UID=045714425B4080000000 TIME=00:00:00 DATE=YYYY-MM-DD
Log[4] @ offset 0x003c0080 (XIP 0x103c0080): UID=04737E3A661990000000 TIME=00:00:00 DATE=YYYY-MM-DD
Log[5] @ offset 0x003c00a0 (XIP 0x103c00a0): UID=045714425B4080000000 TIME=00:00:00 DATE=YYYY-MM-DD
Log[6] @ offset 0x003c00c0 (XIP 0x103c00c0): UID=04737E3A661990000000 TIME=00:00:00 DATE=YYYY-MM-DD
Log[7] @ offset 0x003c00e0 (XIP 0x103c00e0): UID=045714425B4080000000 TIME=00:00:00 DATE=YYYY-MM-DD
Log[8] @ offset 0x003c0100 (XIP 0x103c0100): UID=04737E3A661990000000 TIME=00:00:00 DATE=YYYY-MM-DD
End of log dump.
```

The XIP is the absolute location in memory, and the offset value is how far into the reserved memory the UID is saved. In our specific case with a static save location and no wear levelling implemented having both values is redundant. However, if the information were changing positions according to some algorithm, then both offset and XIP values would be useful.

Initialization

As the program first runs, if a user is connected over serial, they will see a full log showing all previously saved UIDs, memory diagnostics, and serial + NFC reader initialization.

```
***Persistent data address: 0x101C0000***
Log[0] @ offset 0x003c0000 (XIP 0x103c0000): UID=04737E3A661990000000 TIME=00:00:00 DATE=YYYY-MM-DD
Log[1] @ offset 0x003c0020 (XIP 0x103c0020): UID=04737E3A661990000000 TIME=00:00:00 DATE=YYYY-MM-DD
End of log dump.
XIP_BASE           = 0x10000000
PICO_FLASH_SIZE    = 4194304
BIN END OFFSET     = 0x0000A884
LOG FLASH OFFSET = 0x003C0000
LOG FLASH SIZE     = 0x00008000
FLASH: init found 2 existing record(s)
FLASH: log initialised with 2 existing record(s)
Logger ready: cap=1024 UID_LEN=7

Serial input ready.
  Commands: 't' toggle, '1' on, '0' off, '?' status
  Otherwise: any newline-terminated text is accepted as a string.
NFC task starting (I2C0 on GP4/GP5)...
[I2C SCAN] SDA=GP4  SCL=GP5
[I2C SCAN] Done
[NFC] Calling piconfc_init...
[NFC] piconfc_init returned: 1
[NFC] PN532 firmware: 1.60
[NFC] Ready. Will scan every 350 ms.
```

The main values to check are the BIN END OFFSET, showing the end of the program files in flash memory, and then the LOG FLASH OFFSET, showing the beginning of the UID persistent memory. Operators can use these diagnostics to ensure the UIDs and program files are not overlapping.

Invalid Inputs

After the NFC reader detects a card, we compare the received string to the expected string in both length and format. This means we will not be passing along invalid strings further into the logging process.

We also allow users to send UIDs over the serial communication. This input is heavily parsed to ensure it fits the expected format, including checks for the correct length, format, hex validity. The program also formats the strings into upper case to match the format expected from the NFC reader.