

The WAVE System

Wireless Automatic Vendor E-Receipt System

Project Title	WAVE (Wireless Automated Vendor E-Receipt System)
Contributors	Daniel Dreise (7996630) Justin Turcotte (7860406) Brandon Harkness (8048050)
Academic Year	Winter 2021
Course	Capstone I – EECE74125
Program	Bachelor of Engineering – Electronic Systems
Institution	Conestoga College

Abstract

Environmental concerns continue to grow along with the reality of climate change. As such, paper receipts contribute to the waste generated by humans. This project aims to provide an alternative solution to receiving a paper receipt at checkout. Instead of receiving a paper receipt, the customer will have the option to receive a digital receipt to their mobile device. This is accomplished by system that delivers a digital receipt via Near Field Communication (NFC) directly to their phone without. The system has three main modules: Point-of-Sale interface switch (for directing the receipt to either be a paper or digital receipt), NFC device (for receiving the digital receipt from the POS and prepares to send it over NFC), and a mobile application (for receiving and storing the digital receipt on the mobile device).

Keywords: Near Field Communication, Digital Receipt, mobile application, Point-of-Sale

Table of Contents

Abstract	1
Introduction	4
System Description.....	4
Existing Solutions vs Proposed Solutions	4
Rationale	5
Environmental Impact & Safety	5
System Design	6
Overview	6
Components	7
UniCenta oPOS.....	8
CUPS.....	8
Raspberry Pi 4	8
Raspberry Pi Pico	8
NFC Controller + Antenna (OM5577).....	9
TEROW Thermal Printer	10
Android Device.....	10
Android Studio	10
POS Switch Interface	10
Preliminary Design	10
Detailed Design	13
Implementation	15
NFC Transmitter Interface	23
Preliminary Design	23
Detailed Design	35
Hardware	35
Implementation	57
Mobile Android Application.....	58
High Level Design	58
NFC.....	60
Detailed Level Design	68
Implementation.....	68
Testing	76
Unit Tests.....	76
POS Switch Interface.....	76
NFC Device	79
Mobile Application.....	83
System Tests	85
Conclusion.....	86

<i>Abbreviations</i>	87
<i>Appendices</i>	87
<i>Table of Figures</i>	89
<i>References</i>	90

Introduction

System Description

WAVE is a new method of receiving receipts. Through WAVE, the user will only need to tap their phone on a WAVE terminal to transfer their receipt from checkout directly to their phone. A new device, called the POS Switch Interface, would be connected to the POS terminals at various businesses. As customers checkout, a receipt would be passed from the POS into the POS Switch Interface. From there, a switch can be used by the cashier to decide whether the receipt is printed in paper form to a connected thermal printer or in digital form.

When the customer decides they want a digital receipt, the receipt is then sent to the NFC Transmitter Interface. The NFC Transmitter Interface will format an appropriate NFC message, attach the receipt, and prepare it for transfer via NFC. When the user is ready to receive their receipt, they may wave their phone over the NFC terminal and the receipt will be beamed to their device. The customer would also want to have the WAVE mobile application installed on their device to help them manage the receipts and view them.

The following link provides a quick look at the WAVE process, and the following figure shows the main components of WAVE: https://youtu.be/_rBCC7k05oE.

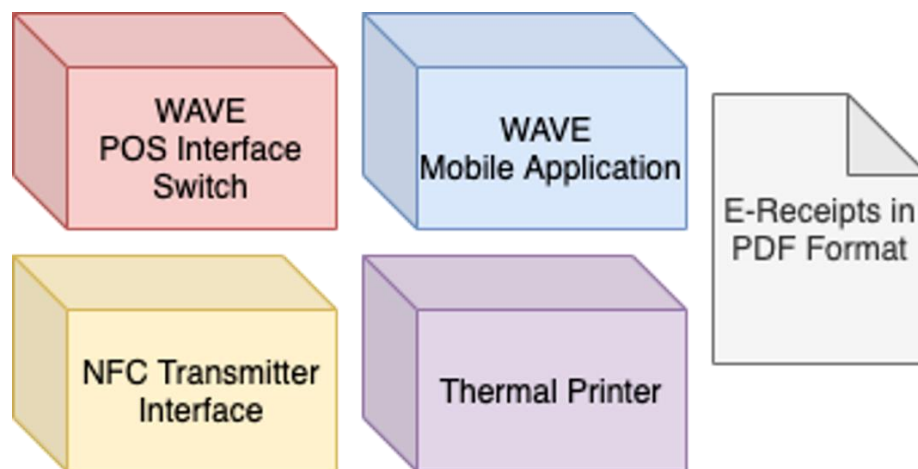


Figure 1: WAVE Components

Existing Solutions vs Proposed Solutions

Current attempts at solving paper receipts are varied. Some are looking to create an ID card that when tapped will reference an email or similar address in the POS and send the receipt directly to the customers inbox. Another possible solution being investigated is integrating directly with banks. Upon purchase, the data from your payment method is used to send a receipt directly to you or your bank.

These solutions have a few flaws. The main one being the loss of privacy. Customers do not want to give up their privacy and the method we mean to implement with WAVE will preserve privacy and keep receipts off the cloud unless the customer chooses to put them there.

WAVE will focus heavily on the NFC enabled device and its interaction with the customer. From this project, customers will have an innovative, standardized, and modern method for providing transaction receipts to their customers.

Rationale

At present, the two most common options available to customers are:

1. Paper receipts.
2. Digital receipts sent through email.

There are issues with both options. The paper copy is liable to decay and/or damage. Even though the email copy is digital and thus exempt from decay, it is cumbersome and awkward providing an email to a cashier being faced with a busy store. WAVE aims to solve these issues by providing a third option via a new device added to the cashier's tool set. Enter WAVE, NFC enabled to allow the receipt to be transferred directly to a customer's phone. A mobile application would be used to assist with receipts stored on customer devices.

Apart from the logistical issues of paper receipts, there are also numerous environmental and health issue associated with paper receipts [1]:

- The U.S. alone generates 686 million pounds of waste from receipts alone,
- Costing 10 million trees,
- 21 billion gallons of water and resulting in,
- 12 billion pounds of carbon dioxide,
- 93% of receipts are coated in a toxic chemical known as Bisphenol-A (BPA) or Bisphenol-S (BPS),
- Salesclerks have on average 30% more BPA or BPS in their systems from direct contact,
- Almost all receipts are not recyclable.

From this, an alternate, more modern, environmentally friendly, and health-conscious solution would be welcomed.

Environmental Impact & Safety

WAVE provides societal, economical, and industrial value. WAVE provides a modern way for customers to receive their receipts while preserving their privacy and remaining simple to use. WAVE also eliminates one point of contact from customer transactions by eliminating the need for a physical

receipt altogether and therefore no longer requiring contact from cashier to customer via the receipt. This is significant, especially during a pandemic, as it helps protect against transmissions.

Another point to consider is that receipts are easily lost and typically are not available when needed. WAVE solves this by providing digital copies that can be accessed simply through your mobile device. By utilizing the file system of the customers device, the customer is also able to backup and store their receipts to their medium of choice whether it be another device or a cloud provider.

Receipts are also a major source of waste. In the U.S. alone, 10 million trees and 21 billion gallons of water are used yearly for receipts. This results in 686 million pounds of waste and 12 billion pounds of carbon dioxide yearly. On top of that, receipt paper is typically coated in toxic chemicals which improve the print quality [2]. WAVE aims to reduce the amount of waste and danger generated from paper receipts. Not only would waste be reduced, but companies would be able to reduce their costs when it comes to obtaining receipt paper. They could also augment WAVE to automatically send employee receipts to a set server where refunds could be handled in a much more efficient and reliable manner.

System Design

Overview

WAVE is an innovative and modern method of receiving receipts. There are four main components: The POS (Point of Sales) system, the Splitter device, the Switch component, the Printer, the NFC device, and the Mobile device. Each of these components work together to get the receipt from the POS to the Mobile or Printer in a reliable and streamlined fashion. See the below figure.

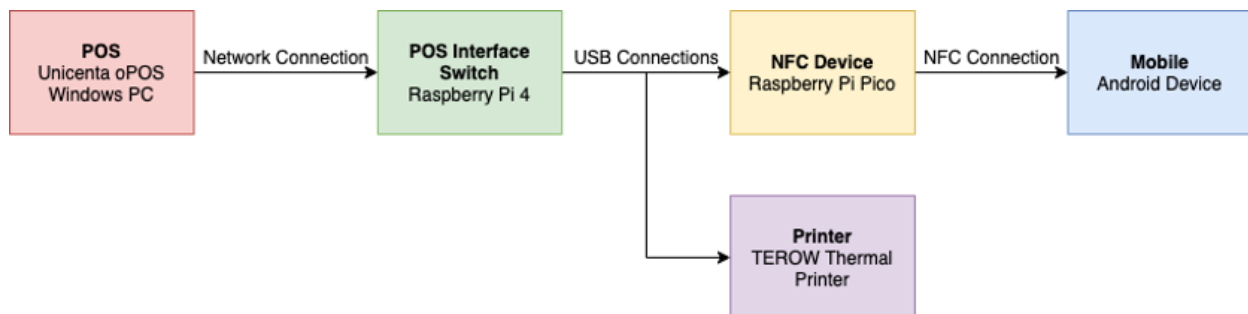


Figure 2: High level view of the WAVE System

The POS is meant to replicate the cash register systems currently in place to serve customers. Slight modifications to the POS are required to allow it to work well with the WAVE system. A new printer must be added to the POS environment with a PostScript compatible driver. The POS system must also be modified to send receipts to this new printer.

The POS Interface Switch device has a range of functions. It is responsible for taking the receipts sent by the POS, checking, based on a custom designed switch circuit, if the receipt will be printed or sent to the user via NFC, and then converting it to the appropriate format before sending (ESC/POS for the printer and .PDF for the NFC device).

The Printer is a standard USB enabled thermal printer that supports the ESC/POS protocol. Upon receiving the proper commands from the POS Switch Interface device, it will print out a receipt for the user.

The NFC Transmitter Interface device is responsible for taking the receipt and coordinating its transfer over NFC to the Mobile device. When the NFC Transmitter Interface device is ready to transfer the receipt, it will light up an LED indicator that will notify the user that it is ready for transfer. The user then *WAVEs* their device across the NFC terminal and the transfer begins.

The Mobile device is the end point for WAVE. A custom Android application will be responsible for ensuring the NFC transfer of the receipt is done intuitively for the user. Once the NFC transfer begins, an internal trigger in the Android environment will cause our WAVE application to launch and receive the receipt. Once the transfer is complete, the user will be able to navigate through the application to view their receipt. The receipt is only stored locally on their device as a PDF file. We do not restrict the user from deciding to upload it to cloud storage or move it to a different device.

Components

This section contains a table that lists the components being used with this project and whether they are 'off-the-shelf' or designed and developed internally.

Component	Origin
UniCenta oPOS	Off-the-shelf
CUPS (Virtual Printer Service)	Off-the-shelf
Raspberry Pi 4	Off-the-shelf
Splitter Software	Internal (Using Python Libraries)
Switch Circuit	Internal
Raspberry Pi Pico	Off-the-shelf
NFC Controller (PN7120)	Off-the-shelf
NFC to Raspberry Pi Pico Interface board	Internal
TEROW Thermal Printer	Off-the-shelf
Android Device	Off-the-shelf
Android Studio	Off-the-shelf

Although many components are 'off-the-shelf' it should be noted that they are being used to assist in the development of the software and infrastructure required for WAVE to function. For example, the Raspberry Pi 4, the POS Switch Interface device, is 'off-the-shelf'. Despite this, the interface between the Splitter and the various components is setup internally and the software responsible for handling this is created in Python utilizing various libraries.

UniCenta oPOS

UniCenta oPOS is an open-source point of sale system that can be installed on Windows, Mac, and Linux. It allows us to replicate the cashier experience in your typical store so that we can see exactly how it interacts with WAVE. It also saves us the cost of purchasing and assembling a complete POS system [3].

CUPS

CUPS is a network printer hub service for Unix-like operating systems. It utilizes the internet printing protocol (IPP) and allows us to create a virtual printer for the Splitter device that the POS system can print to. This is the first stage in re-routing the receipt from the POS [4].

Raspberry Pi 4

The Raspberry Pi 4 is a powerful miniature development computer that can run an operating system. With the Raspberry Pi 4 we can augment the benefits of an OS while maintaining a small footprint. This is the brains of the POS Switch Interface device and will handle things from receiving the POS receipt to deciding if it should be sent to the Printer or the NFC device. It also handles any required conversions [5].

Raspberry Pi Pico

The Raspberry Pi Pico is a microcontroller built and designed by Raspberry Pi based on the RP2040 SoC. Its technical specifications are as follows:

- 21 mm × 51 mm form factor
- RP2040 microcontroller chip designed by Raspberry Pi in the UK
- Dual-core Arm Cortex-M0+ processor, flexible clock running up to 133 MHz
- 264KB on-chip SRAM
- 2MB on-board QSPI Flash
- 26 multifunction GPIO pins, including 3 analogue inputs
- 2 × UART, 2 × SPI controllers, 2 × I2C controllers, 16 × PWM channels
- 1 × USB 1.1 controller and PHY, with host and device support
- 8 × Programmable I/O (PIO) state machines for custom peripheral support
- Supported input power 1.8–5.5V DC
- Operating temperature -20°C to +85°C
- Castellated module allows soldering direct to carrier boards
- Drag-and-drop programming using mass storage over USB
- Low-power sleep and dormant modes
- Accurate on-chip clock
- Temperature sensor
- Accelerated integer and floating-point libraries on-chip

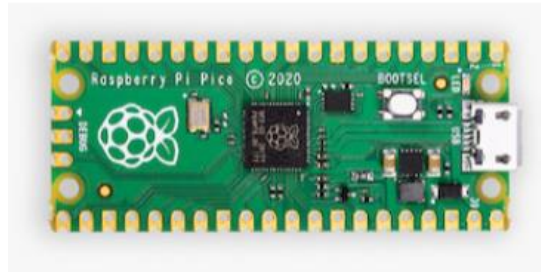


Figure 3: Image of Raspberry Pi Pico []

NFC Controller + Antenna (OM5577)

The OM5577 is a single-board-computer (SBC) which integrates the PN7120 NFC controller from NXP with a built-in antenna for easy integration into projects. It comes with a few development kits, such as the NXP-NCI and NDEF libraries. Some specifications are below:

- Integrated communication protocols to support Read/Write, Peer-to-Peer, and Card Emulation mode.
- Supports NFC Forum Type 1/2/3/4/5 Tags.
- Drivers for easy integration into Linux and Android based systems.
- NCI over I2C interface.
- Ready to use demo examples included for reading and writing tags as well as interfacing other NFC enabled devices via Peer-to-Peer.



Figure 3: Image of the OM5577 [],

TEROW Thermal Printer

The TEROW Thermal Printer is a 58mm wide thermal printer that allows us to emulate the exact type of hardware a typical sales system would have for printing customer receipts. It is also a USB enabled printer that can be connected directly to the Raspberry Pi 4 and interfaced with using the ESC/POS protocol which is a standard printing protocol for thermal printers [6].

Android Device

Testing will be done using the virtual device feature in Android Studios. The android device that will be used to initially test the mobile application will be the Samsung Galaxy S10, primarily due to already having this device available and it has NFC hardware.

Android Studio

Android Studio is the official integrated development environment for android applications. It is a free to download software.

POS Switch Interface

The POS Switch Interface serves as an intermediate system between the POS & the NFC Transmitter Interface device. The POS Switch Interface will receive the receipt from the POS system, and either transfer it to the NFC Transmitter Interface or the Thermal printer. A switch on the POS Switch Interface device allows the user to decide where the receipt is going.

Preliminary Design

Device Selection

During our research we compiled a list of requirements that the POS Switch Interface device would need to meet. These requirements included the ability to have three simultaneous USB connections, the ability to interface with external custom circuitry (via GPIO for example), and a small form factor. Keeping these requirements in mind, we compared three candidates for the Splitter: A Raspberry Pi, an Arduino, and a Micro Controller (such as a Nucleo board). The below table is a compilation of our comparison.

Advantages	Disadvantages
Raspberry Pi	
<ul style="list-style-type: none">• Put together small computer• Plenty of connection ports & GPIO's• Plenty of online resources• Variety of supported programming languages• Minimal initial setup• Least involved• In possession• Available OS	<ul style="list-style-type: none">• Potential for lots of bloat (unnecessary software such as an OS)• Lowest potential learning experience• Minimal customization options

<ul style="list-style-type: none"> • High computing power 	
Arduino	
<ul style="list-style-type: none"> • USB & GPIO compatible • High amount of available resources • FreeRTOS support • Mixed complexity → can use shields & circuit design. 	<ul style="list-style-type: none"> • Requires purchase of Arduino and necessary shields. • Most expensive option • Least familiar device • Can only find single USB shields → we need 3 USB slots.
Micro controller (Nucleo board)	
<ul style="list-style-type: none"> • RTOS support • Customizable • GPIO support • USB compatible with circuit design • Highest potential learning experience 	<ul style="list-style-type: none"> • Requires purchase • Most complex • Less available resources • High initial setup & design

After comparing the above options, we concluded that a Raspberry Pi device is our best option. It has four USB connection ports, it has a GPIO header that we can use to interface with the switch circuit, and it has a small form factor while still allowing us to augment the benefits of an operating system. We did consider a Raspberry Pi Pico however, it did not have the needed USB ports, and it is much more relatable to a micro controller. Therefore, we decided to move forward with a Raspberry Pi 4. An added benefit is that a Raspberry Pi 4 is already in our possession which is why we are not considering an earlier model.

Switch Comparison

The switch circuit which will allow the cashier to decide whether a receipt is received in paper or digital form. The basics of the circuit is to use a single toggle switch, connected between power and a GPIO pin of the Raspberry Pi 4. Which when in one state will read a high while the other state will read a low. The software will then discern between these values and decide what to do next. The following figure is an overview of the circuit.

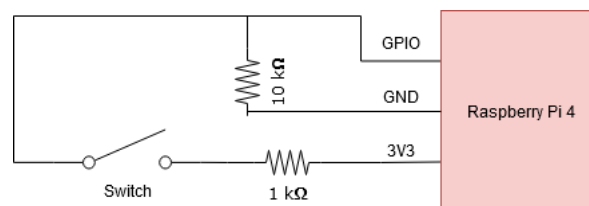


Figure 5: Overview of switch circuit

Essentially the power, 3V3 is fed into one side of the switch through a current limiting resistor. The other side is connected to a GPIO pin on the Raspberry Pi that will read the current voltage where $<0.8V$ is a low and $>2.0V$ is a high [7]. A pull-down resistor is used to ensure that when power is not connected, the GPIO pin will be pulled to ground.

Next, to complete this circuit, we had to research appropriate switch types that will work for our implementation. The main requirement for this is that the switch should be easy and intuitive for the cashier to manipulate. The following table is a comparison of a few types of switches and their features.

Name	Features
Rocker switch	<ul style="list-style-type: none"> one side for printer, other side for NFC device
Rotary switch	<ul style="list-style-type: none"> rotates, not suitable for this purpose because there's not a clear indication of where each output is & there are too many switch states that would over complicate things.
Slide switch	<ul style="list-style-type: none"> Two state slide switches as shown would be ok. clearly indicates output through slide to other side.
Toggle switch	<ul style="list-style-type: none"> 2-state toggle switch is also Switch outputs with a simple flick.

Based on this comparison and after thinking about what would be intuitive for a cashier, we decided a toggle switch would make the most sense. We have designed the circuit using a toggle switch (A101SDCQ04) [8].

Receipt Format

In determining the format for the digital receipt there were many considerations. The main two were a CSV format and a PDF format. The following table outlines the characteristics of each.

CSV File Format	PDF File Format
<ul style="list-style-type: none"> Simpler format. No image support low file size Has a python library .csv 	<ul style="list-style-type: none"> Larger file size Image support Higher complexity Common Has a python library PyPDF

After considering these characteristics we decided PDF would best fit our needs. Although CSV is tempting due to it being a smaller footprint and text based, it does not support images. Most receipts contain logos and other small images that we would also need to support. Therefore, although PDF is a binary format and a larger footprint, we decided it would be the better fit. Fortunately, many libraries also exist that allow us to work with PDF files.

Software Language Selection

Since the Splitter device required custom software, some effort needed to be used to consider which programming language was right for our implementation. We considered two main languages: C/C++ and Python. The following table compares them both.

C/C++	Python
<ul style="list-style-type: none"> Most familiar. More complex. Lower level, higher flexibility. 	<ul style="list-style-type: none"> Native to Raspberry Pi. High degree of resources & libraries available.

<ul style="list-style-type: none"> - Compiled to machine code → potential to be faster. 	<ul style="list-style-type: none"> - PyUSB - Scripting language → tends to be simpler. - Compiled at run time → potentially slower.
--	--

Although both languages have their advantages, and we are more familiar with C/C++. The ease of use and massive library support of Python, not to mention its native support with the Raspberry Pi pushed us to decide that Python was the right fit for our implementation. Although speed would normally be a concern, considering the strength of the Raspberry Pi 4 and that the bottleneck will most likely be the NFC transfer itself, we still believe Python will provide the most benefit.

Detailed Design

Hardware Overview

The hardware section of the POS & Splitter includes the various connections between the various components of the WAVE system. With this, there is a switch circuit that is designed and implemented internally that allows the cashier to manually choose where the receipts go. Whether that be the thermal printer or NFC device. The below figure shows an overview of the hardware connection in the POS & Splitter components.

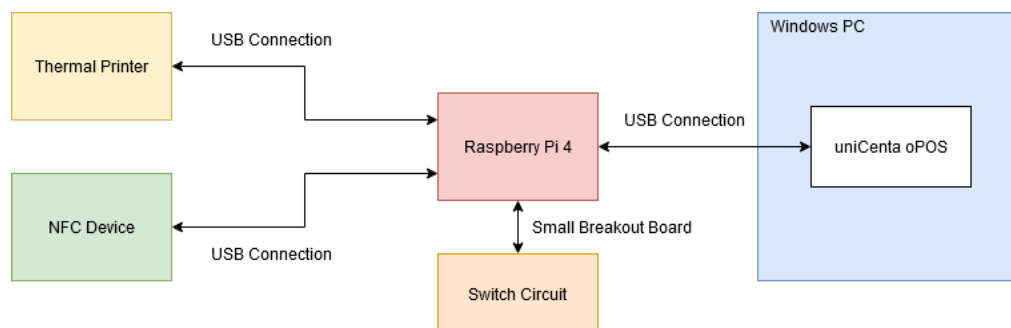


Figure 4: Overview of POS & Splitter design

As shown, the interface between the POS, Splitter (Raspberry Pi 4), Thermal Printer, and NFC device is a USB connection. Later, we will discuss the implementation and design of the Switch Circuit. The next section delves into our decision to use the Raspberry Pi 4 as the Splitter device.

Switch Circuit Design

Now we will verify the resistor values and ensure they will ensure the circuit operates correctly. With standards and most common values in mind, we initially chose the current limiting resistor to be 1k ohms and the pull-down resistor to be 10k ohms. The following calculations confirm these values in both switch states.

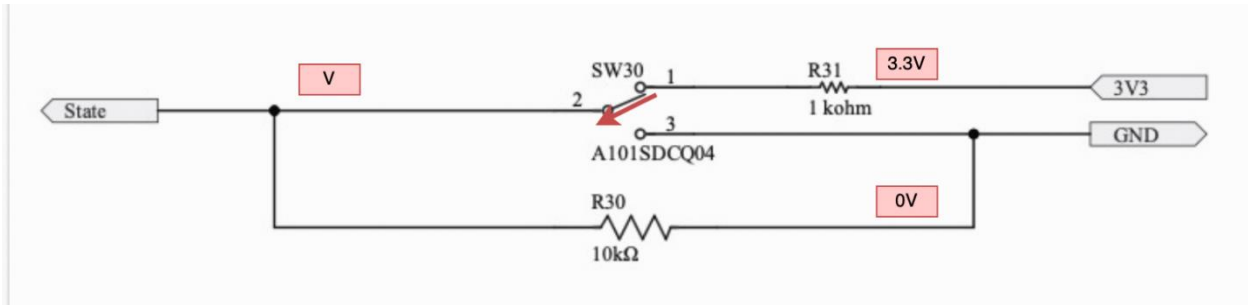


Figure 5: Switch circuit high state analysis

$$V = V_{R_{30}}$$

This is also essentially a voltage divider where:

$$V = 3.3 \left(\frac{R_{30}}{R_{31} + R_{30}} \right) = 3.3 \left(\frac{10k}{1k + 10k} \right) = 3V$$

Therefore we can see that with the current resistor setup, the voltage V is within the range of 1 to 3.3V.

Now we need to ensure the current does not reach above 16mA.

$$I = \frac{V}{R_{total}} = \frac{3.3}{10k + 1k} = 300 \mu A$$

Therefore we can also see that we are well below the maximum 16mA limit.

Figure 6: Switch circuit high state calculations

The Raspberry Pi 4 can handle a maximum of 16mA on each GPIO pin. Therefore, we had to ensure the current would be below this. From the above calculations, you can see that the value at the GPIO pin is 3V at 300uA. This is well within the range for a high value while well below the current limit.

Moving on to the low state:

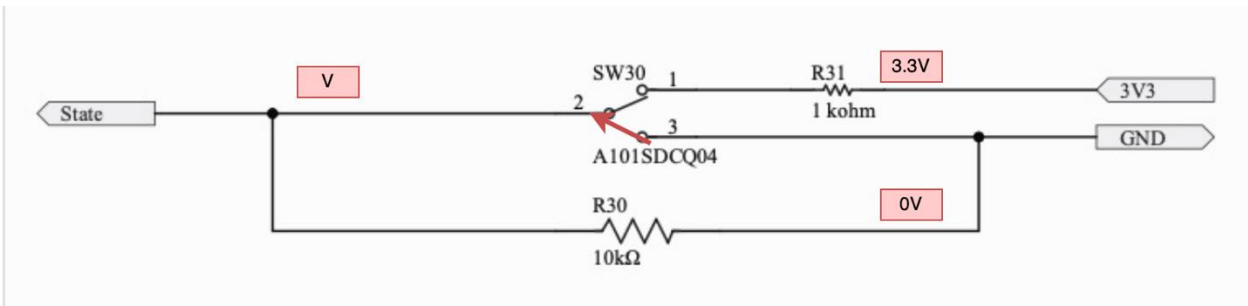


Figure 7: Switch circuit low state analysis

By inspection we can see that while in this state, the GPIO pin is connected directly to ground. Therefore, V is also 0V and no current is flowing. From this we can see that the switch circuit is able to provide the wanted states within acceptable parameters.

Schematic & PCB Design

Keeping the above requirements in mind, the final schematic and PCB design has been created. See the attached PDF.

Software

The software section of the POS & Splitter components resides almost entirely on the Splitter except for a bit of initial setup and configuration that is required to be done on the POS system. The following figure is a general outline of how the custom software implemented on the Splitter will interact with the rest of the components.

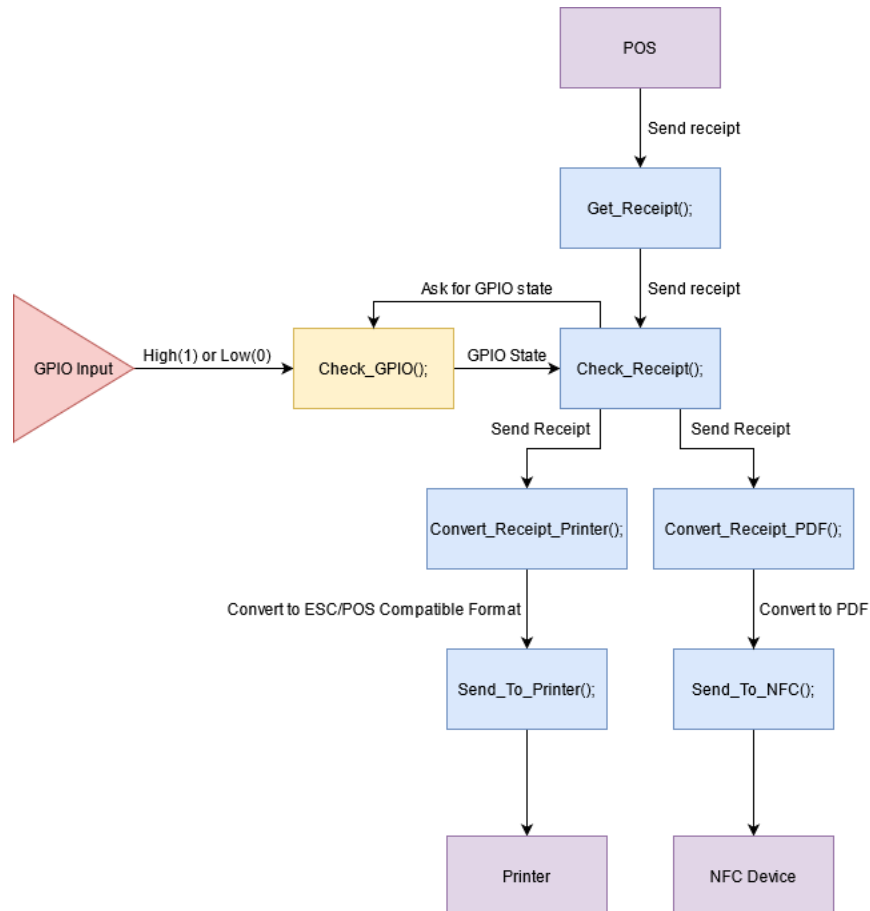


Figure 8: POS & Splitter software overview

Essentially, the receipt is sent by the POS and received by the Splitter at the “Get_Receipt()” section. After that the software checks the current GPIO state. If high, we send it towards the thermal printer. If low, we send it towards the NFC device. In either direction, the receipt is checked to make sure that it is in its proper format. For the thermal printer that’s ESC/POS print commands and for the NFC device it’s a PDF formatted receipt.

ESC/POS is a standard protocol used for printers commonly found in the sales environment. It is commonly used in the POS to print receipts. It’s commonly used due to its simplicity and there are already many libraries in python and various languages that implement it [9]. For these reasons we ensured we got a thermal printer with this compatibility.

Implementation

POS

Many POS systems found in commercial stores run on a Windows background with POS software on top. Fortunately for us, there are many open-source POS systems available for us to use that allow us to

replicate the sales environment during our development. Our main requirements for a POS system are flexibility in allowing us to define a printer, something resembling an existing POS system and low cost. Using these requirements, we found UniCenta oPOS. UniCenta oPOS is an open-source POS system that allows us to mimic the work of a cashier when developing and testing the WAVE systems. The following figures show the POS window a cashier might see as well as an example receipt that was printed to a PDF printer.

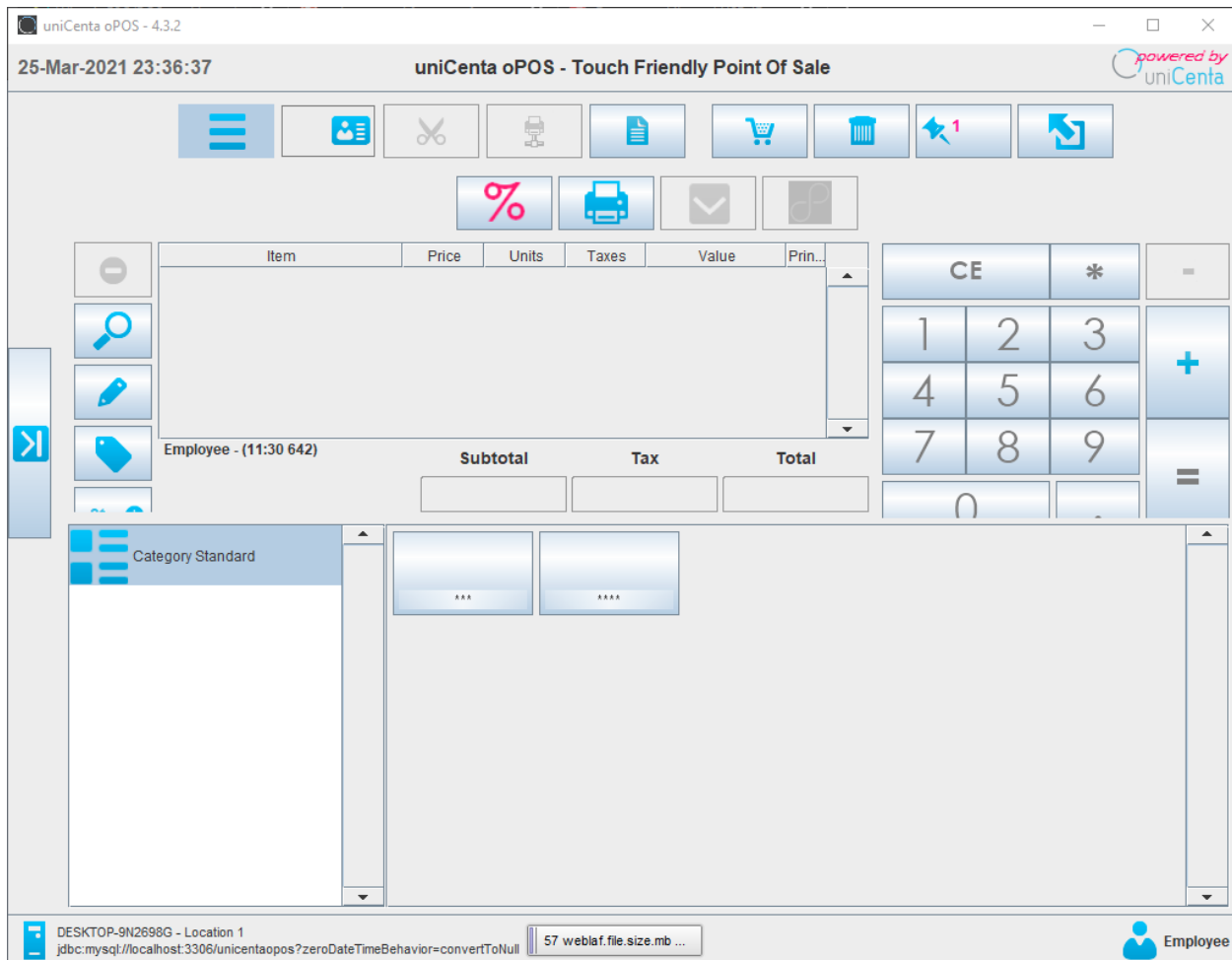


Figure 9: Unicenta oPOS cashier screen

Sample Restaurant		
Somewhere		
+0123456789		
*** PAYMENT RECEIPT ***		

RETAIL		
Terminal#: 270		
CHK#: 17		
Server: Admin System		
Printed: 2/18/21, 4:27 PM		

ITEM	QTY	SUB

KIDS BLUE T-SHIRT	1	10.00
DESIGNED T-SHIRT	1	12.00

Subtotal \$	22.00	
Tax \$	1.32	

Total \$	23.32	
Tendered \$	23.32	
Paid \$	23.32	
Due \$	0.00	

Figure 10: Unicenta oPOS sample receipt

The POS will connect to the Splitter device over the local network utilizing a service called CUPS. For the POS to be configured correctly, the CUPS service must already be setup on the Splitter device. Here we will explain that process.

The general process for installing CUPS is to first download it using the apt install manager on the Raspberry Pi 4, then modify the configuration file to configure CUPS to listen for all computers on port 631. Then you should be able to access the website, “localhost:631” and be shown the CUPS interface. From here you would go to the “Administration” page and add a printer. Before continuing however, you also must install cups-pdf with the apt install manager. This allows us to setup a virtual-pdf printer which will allow the POS to print to the Raspberry Pi and save the file locally on the Raspberry Pi as a PDF file [4].

After CUPS has been setup, we are able to return to the POS and finish its setup. In the windows environment on the POS a new printer needs to be added. It is very important that when configuring the new printer in the Windows environment that the associated driver is PostScript compatible. Without this, The WAVE system may work via NFC, but anything printed to the thermal printer will appear as jumbled symbols. Selecting the virtual printer setup with CUPS allows the POS system to then select that virtual printer as its destination and once that is complete all receipts will be printed to the Raspberry Pi 4.

POS Switch Interface

After the receipt is sent over to the POS system, a Python script is responsible for constantly checking the destination folder for new receipts. A tool called crontab can be used to set a python script to run on. The below figure is example code that performs this function:

```
1  import os
2  import time
3
4  # Paths to the POS Printer Destination & the Fake NFC Device
5  POS_PRINT_DEST = "/var/spool/cups-pdf/ANONYMOUS/"
6  RECEIPT_DIR = "/home/pi/Documents/Check_Receipt_Program/NFC_Device/"
7
8  # Do this forever so that all receipts are grabbed
9  while True:
10     pdfs = os.listdir(POS_PRINT_DEST) #get list of files in POS printer dest
11
12     # As long as there is a file, move it to the fake NFC device & delete it
13     for i in range(0,len(pdfs)):
14         # Open the original receipt and its new home
15         # Binary format because PDF files are binary
16         fr = open(POS_PRINT_DEST+pdfs[i], 'rb')
17         fw = open(RECEIPT_DIR+pdfs[i], 'wb')
18
19         fw.write(fr.read())    #copy the receipt to a new location
20
21         # Close the files
22         fr.close()
23         fw.close()
24
25         # Remove the oriinal file
26         os.remove(POS_PRINT_DEST+pdfs[i])
27
28         # Wait a moment before checking again
29         time.sleep(1)
```

Figure 11: GET_RECEIPT(); Example

This example program monitors the incoming receipts and automatically moves them to a new destination. In the below demonstration this destination is a fake NFC device that will be replaced by the thermal printer. This example program also utilizes the OS and Time libraries. The former allowing OS operations like the moving the file and the Time library allowing the wait command, so the python script doesn't get forcefully terminated.

Switch Program

The switch program is utilizing the previously designed circuit to read in the state of the attached GPIO pin and decide where the receipt will be going. We chose GPIO25 partly at random since any of them will do. It aesthetically looked like a nice location in the schematic. We also created an example Python script to demonstrate the function that this functional unit will perform. We also utilize the GPIO Zero library to interact with the GPIO pin. See the below code statement and demonstration.

```
1 import RPi.GPIO as GPIO
2
3 GPIO_PIN = 25      #we're using GPIO "BCM 25"
4
5 GPIO.setmode(GPIO.BCM)      #use BCM numbering
6 # Initialize GPIO25 as LOW (also the default state of this pin)
7 GPIO.setup(GPIO_PIN, GPIO.IN)
8 state = GPIO.input(GPIO_PIN)      #get the state of the gpio pin
9
10 # Depending on the state of the GPIO, decide where the receipt goes
11 # 0 -> NFC Device
12 # 1 -> Printer Device
13 if state == 0:
14     print("Sending receipt to NFC device!")
15 elif state == 1:
16     print("Sending receipt to Printer device!")
```

Figure 12: Example getting the switch state

The above example works by configuring the GPIO pin to be an input and then simply calling the "GPIO.input" command to read the current state. If it is low, we send to the NFC device. While high means we send to the printer device.

Splitter to Thermal Printer

The splitter to printer interface mostly involves converting the PDF receipt into an ESC/POS compatible list of commands that can then be sent directly to the thermal printer via USB. Utilizing the Python libraries python-escpos and PDF to text we can develop an example program that will convert the PDF receipt into a set of ESC/POS commands and send it to the thermal printer for printing. See the below code statement and demonstration.

```

1  # Printer Information:
2  # Vendor ID:Product Code = 0416:5011
3  # Interface number = 4
4  # Output Endpoint Address = 3
5
6  # The python-escpos library is installed to a separate location so add
7  # that to the library path
8  import sys
9  escposModulePath = "/home/pi/.local/lib/python3.7/site-packages"
10 sys.path.append(escposModulePath)
11
12 # Import the necessary module from python-escpos
13 from escpos.printer import Usb
14 import pdftotext
15
16 p = Usb(0x0416, 0x5011, 0, 0x04, 0x03) #setup interface with printer
17
18 with open("test-receipt.pdf", "rb") as f: #open pdf file and
19     pdf = pdftotext.PDF(f)                #convert to text object
20
21 text = "\n\n".join(pdf)    #extract text and join it to a single string
22 p.text(text)               #print text to thermal printer
23 p.cut()

```

Figure 13: Example code which will print a pdf to the Thermal Printer

The USB specifications can be found with the “lsusb” command in a terminal. Essentially, the example program takes in the test receipt and extracts the PDF text and sends it to the printer. The printer then automatically reads the ESC/POS commands and prints out the resulting receipt.

The below figure is the receipt shown in the above demonstration.

Please Call Again
uniCenta oPOS
Touch Friendly Point Of
Sale
Copyright (c) 2009-2017 un
iCenta

Printed using WAVE!
Printer.Ticket

Receipt: 12
Date: 13-Apr-2021 4:27
:28 AM
Terminal: WAVER-1
Served by: Employee

Item	Price	Q
ty Value		

Black Tea	\$2.00	
x1 \$2.00		
Fruit Tea	\$5.00	
x1 \$5.00		
Green Tea	\$1.00	
x1 \$1.00		

Items count: 3		
Total	\$8.00	
Nett of Tax:	\$6.67	
Taxes:	\$1.33	
Tax Exempt	\$0.00	\$0.00
Tax Other	\$0.00	\$0.00
Tax Standard	\$1.33	\$6.67
Cash		
Tendered:	\$8.00	
Change:	\$0.00	
	WAVE	
=====		
=====		
Please Call Again		

Figure 14: Example paper receipt

Splitter to NFC Device

Fortunately, for the splitter to NFC device unit, the receipt is already in PDF format. Therefore, no conversion is necessary. Another thing to note is that the NFC device will be accessed via USB and can be interacted with like a flash storage device. Therefore, for this demonstration, a USB drive is used in place of the NFC device. Utilizing the previously mentioned OS and Time libraries, a similar script to the "Check_Receipt()" script is used. See the below code statement and demonstration.

```
1 import os
2 import time
3
4 NFC_DEVICE_PATH = "/media/pi/TESTUSB/"
5
6 # Binary format because PDF files are binary
7 fr = open("test-receipt.pdf", 'rb')
8 fw = open(NFC_DEVICE_PATH+"test-receipt.pdf", 'wb')
9
10 fw.write(fr.read())           #copy the receipt to the fake NFC device
11
12 # Close the files
13 fr.close()
14 fw.close()
15
16 # Remove the oriinal file
17 os.remove("test-receipt.pdf")
```

Figure 15: Example code to send the receipt to the NFC Transmitter Interface

The code fragment works by taking the path to the USB drive and opening the receipt in read binary format. A new file is created on the USB drive in write binary format. Then the contents of the original receipt are copied over to the new file. The original receipt is deleted in the end.

Demonstration

The following link leads to a demonstration of the POS Switch Interface going from the POS to the thermal printer & NFC Transmitter Interface devices: <https://youtu.be/qXKvb0ORoqs>.

NFC Transmitter Interface

The NFC device serves as the intermediate system between the POS and the customer's mobile device. The goal of this device is to transfer the receipt it received from the POS to the customer via an NFC connection. The NFC device will be responsible for formatting the message data with the proper headers and components required for an NFC transfer. This is different from the formatting done on the POS side as the POS is responsible for ensuring we are sending an appropriate file format. The NFC device will be powered by a microcontroller which will be capable of receiving the receipt file, formatting the message, signaling the customer it is ready to transfer the receipt, and finally performing the transfer. Upon completion of transfer, the NFC device will stop transmitting and wipe the receipt from memory. The customer will be signaled by a solid green LED which when lit, means the NFC device is ready for transfer.

Preliminary Design

The main purpose of the NFC device is to perform as a “black box” that receives a receipt file from the POS Switch Interface and send that file over NFC to a mobile device. This can be done in a number of ways, but there are a few main “ingredients” that are necessary to make it happen.

Necessary Components of NFC device

To accomplish this there are a few necessary components.

- Some method of communication with POS Switch Interface
- Something to process the information
- Something to handle and manage the NFC protocol
- And NFC antenna
- A power supply

For this project, we decided to use USB connection with the POS Switch Interface to act as our method of communication between the NFC device and the POS Switch Interface. The double benefit of using USB to communicate is that it can also be used as a power supply so we wouldn't have to have an extra power supply circuit.

For processing the information, we decided to use a micro controller. A microcontroller was selected because they are good at performing a few functions repeatedly, low power consumption, and are inexpensive. Since we didn't need a computing power of a micro processor or an actual computer, a microcontroller was our best choice.

To handle the NFC protocol, we would also need some form of NFC controller, which is essentially a microcontroller dedicated and made specifically for generating NFC compliant communication. NFC protocol, as defined by NXP, requires special data framing and headers, along with RF anti-collision measures. The NFC controller accomplishes these tasks independently of the host microcontroller.

Lastly, we need an antenna to generate the RF field that NFC communicates with. This can either be custom designed to meet certain constraints but can also be purchased as an off-the-shelf component.

Since there are a few different methods for implementing these things, to determine which method was best suited for this project a comparative analysis was done with some of the different top level design options.

Top Level Design

(NFC Antenna + NFC Controller) + (MCU + USB)

The second option separates Option 1's design into two main components. The first component consists of both the NFC controller and the NFC antenna, and the second component includes the microcontroller and the USB connection. The system design for this can be seen in Figure 16.

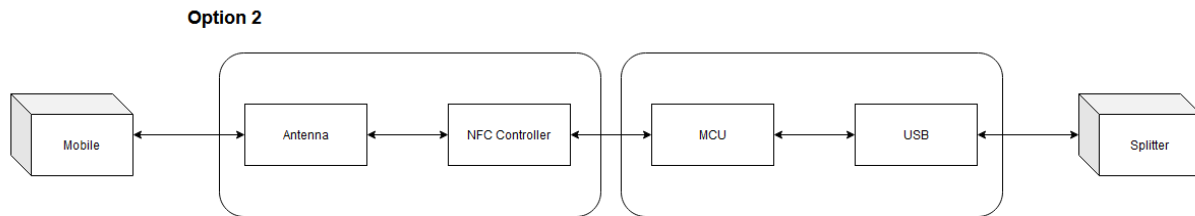


Figure 16: NFC device - device design option 2 - two separate modules.

This option uses a microcontroller (like an Arduino or Raspberry Pi Pico) with an NFC breakout/daughter board connected to it. Essentially separating the NFC portion from the MCU portion.

shows the research conclusions of the option 2 top level design for the NFC device.

Table 1: NFC device top level design option 2 research conclusions.

Conclusion	Possible Workaround	Pros	Cons
<p>So far, the only MCU's that have enough memory to hold the receipt file need to be powered by DC supply.</p> <p>Our goal is to be powered by USB.</p>	<p>Splitter device would store receipt instead and release to NFC when ready.</p> <ul style="list-style-type: none"> When NFC device detects mobile device, would send a "ready to receive" signal to Splitter. Splitter would release receipt and directly transmit via USB > NFC device > mobile. Data would need to be received in chunks from 	<p>Less time doing hardware design</p>	<p>More time doing software design (for splitting file into chunks)</p>

	Splitter to accommodate for small memory size.		
	Be powered by DC connector	Easy solution	Not ideal due to having another connection running to the NFC device (along with USB cable). Seems overkill.
No interface between the MCU and NFC controller	Design PCB interface board.	Simple, elegant.	A little bit of extra time and budget for designing and manufacturing.

Top Level Design Results

Based on the advantages and disadvantages of each system design, we decided to go with option 2, having two off-the-shelf components:

1. Microcontroller
2. NFC controller board development kit

It would look something like the high-level design shown in Figure 17.

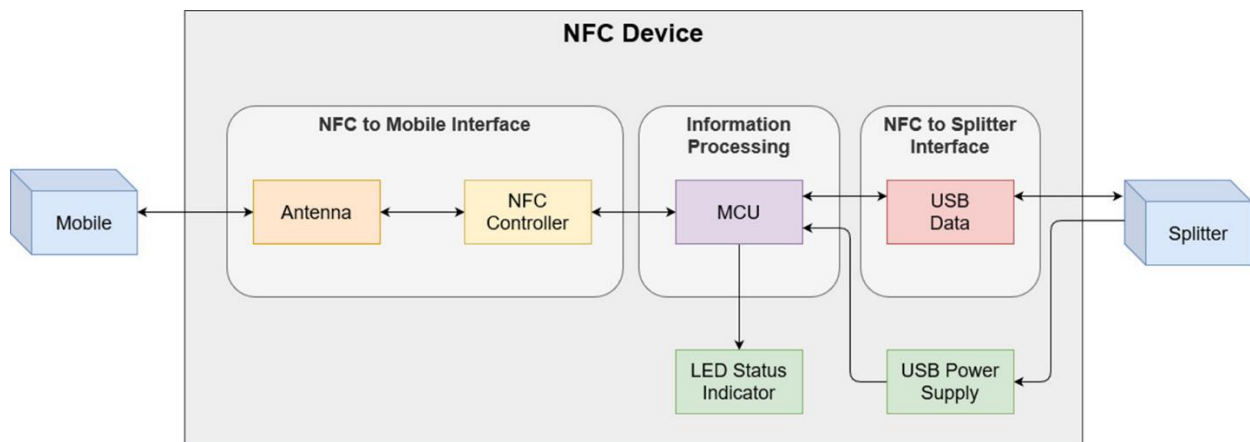


Figure 17: High-level design

Interface Board

Since the plan is to use an off-the-shelf component for both the NFC portion of the NFC device and the microcontroller portion of the NFC device, some form of physical interface needs to be designed between the two. As seen in **Error! Reference source not found.**, the idea is to design a Printed Circuit

Board (PCB) that you could mount the NFC Controller + Antenna, and the Microcontroller + USB to provide a solid interface between the two.

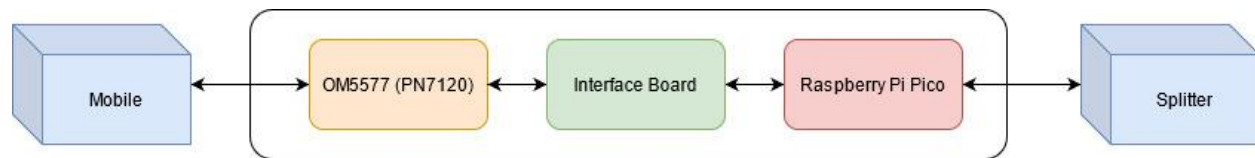


Figure 18: Diagram showing placement of interface board.

NFC Controller + Antenna

The first step of understanding how to design for the NFC portion, it is important to understand NFC itself, because one goal that we have is to get the speed of NFC transfer below a certain threshold. This section will look at investigating the NFC protocol stack to determine payload size and transmission speed.

NFC has three different modes:

1. Card Emulation
2. Read/Write
3. Peer-to-Peer

In each mode, the devices can be in either Active (generating the RF field) or Passive (listening for RF field) mode. Since both the mobile device and our NFC device are powered devices, we will be using Peer-to-Peer (P2P) mode which means that both devices will flip-flop between being Active and Passive. Therefore, we need a device that can operate in P2P mode.

NFC protocol is standardized. For our purposes, since we're using P2P mode, relevant information can be referenced in the ECMA-340 and the ISO/IEC-18092 documents.

NFC stands for Near Field Communications Protocol. As the name suggests, it is designed for short range communications with a maximum distance of around 10cm. Connection time is relatively quick at 0.1 seconds. For reference, "tap" methods of payment already rely on NFC communications. Our implementation with WAVE extends it to the other side, allowing customers to receive their receipts, with the same technology allowing them to pay for their purchases, with their mobile devices. NFC can operate up to 424Kbps. Initial tests with custom .csv receipts brought the total file size well below that value and supports our expectation of fast transfer times. A typical NFC transmission is called a message. A message is made up of a header and payload, the payload being the data (file, URL, etc.) being transferred. The header is then made up of an identifier, length, and type which are used to describe the data and message being sent.

To understand NFC it's best to look at the protocol stack (Figure 19).

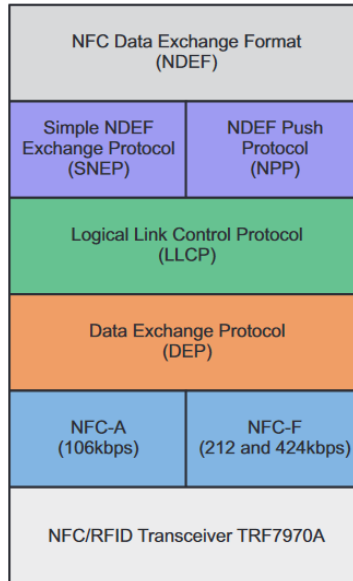


Figure 19: NFC Protocol stack

Application/Transport Layer (NDEF)

What gets sent over NFC is an NDEF Message, which essentially consists of some NDEF Records that have the header information and the payload (as seen in Figure 20).

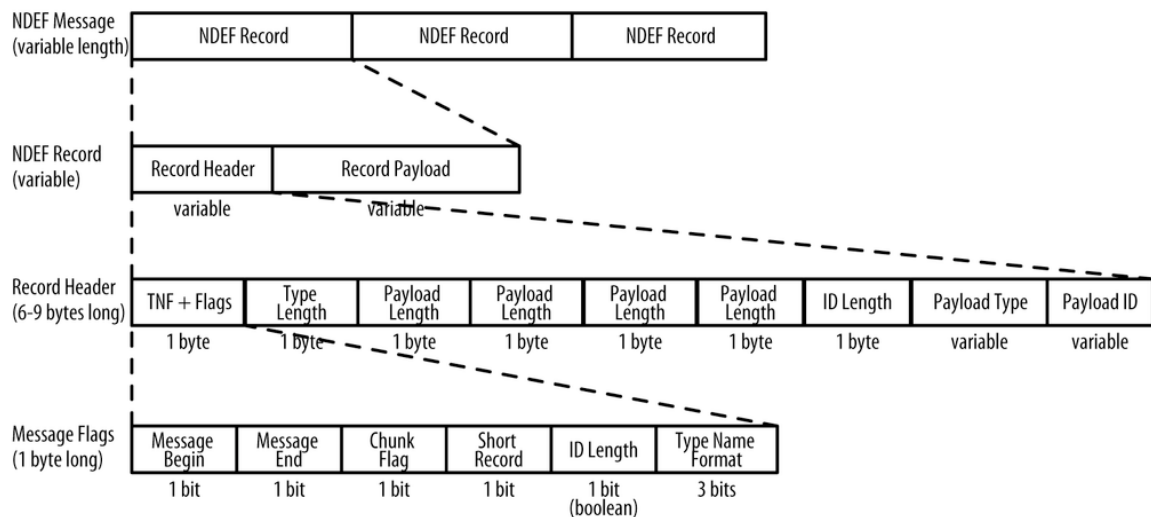


Figure 20: NDEF Message

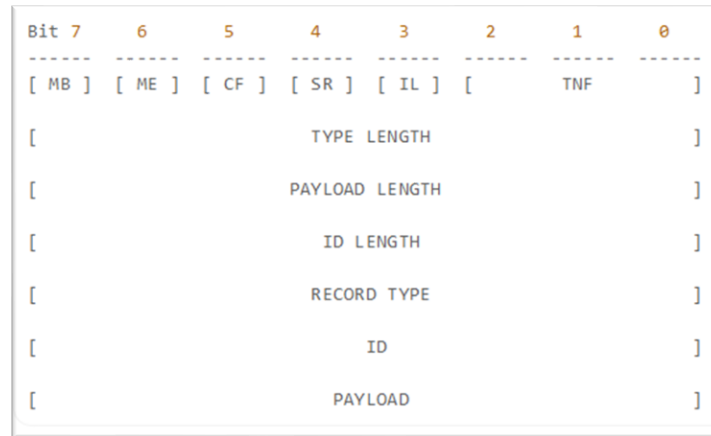


Figure 21: NDEF Record layout

The size of the payload is calculated below:

NDEF Record Payload Length = $2^{23} - 1 = 8,388,607$ bytes

NDEF Record Header Length = **9 bytes**

Assuming we're working with a typical receipt size of 100kB, let's say our payload is 100kB.

Payload = 100kB

Header = 9 bytes

Total size of NDEF packet = **109kB**

Logical Link Control Protocol (LLCP)

Defined in the IEEE 802.2 standard.

802.2 Frame Format (LLC)

6 bytes	6 bytes	2 bytes	1 bytes	1 bytes	1 bytes	Up to 1497 bytes
Destination Address	Source Address	Length	DSAP	SSAP	Control (0x3)	Network Packet

Header size = $6 + 6 + 2 + 1 + 1 + 1 = 17$ bytes

Payload size = < 1497 bytes

However, our current payload size (NDEF + NPP) = 109,006 bytes

Therefore, fragmentation needs to occur...

$$109,006 / 1497 = 72.82 = 73$$

Therefore, the LLC packet needs to be sent 73 times.

So the actual size of the header, multiplied by the number of packets...

$$17 \text{ bytes} \times 73 \text{ packets} = \mathbf{1241 \text{ bytes}}$$

$$\text{Total so far is } 109006 + 1241 = \mathbf{110,247 \text{ bytes}}$$

Data Exchange Protocol (DEP)



Figure 22: DEP frame format.

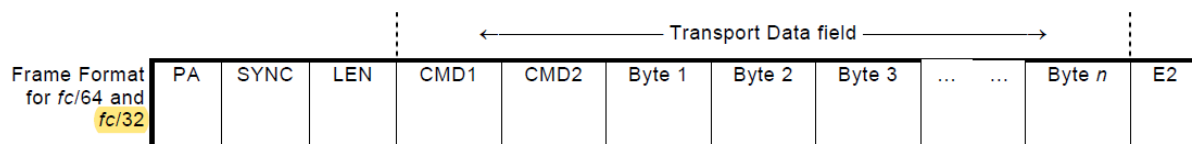


Figure 23: DEP transport data frame format.

Preamble → min 48 bits (all logic ZERO)

Sync → 2 bytes (B2 and 4D)

Length → 1 byte

Payload → 3 to 255 bytes

CRC → 7 bytes

$$6 + 2 + 1 + 2(\text{command bytes}) + 7 = \mathbf{18 \text{ bytes}}$$

The payload is only $255 - 3 = 252$ bytes

With our current payload of 110,247 bytes, fragmentation must occur:

$$110,247 / 252 = 437.49 = 438$$

Then, we must sent 438 packets.

The header size multiplied by the number of packets...

$$438 \times 18 = \mathbf{7884 \text{ bytes}}$$

Header size = **18 bytes**

Header size due to fragmentation = **7884 bytes**

Total size of payload so far = $7884 + 110,247 = \mathbf{118,131 \text{ bytes}}$

Analog / Digital Layer

This layer, which is specified under the ECMA-340 documentation (specifically NFCIP-1) is the actual radio frequency generation portion of NFC. Due to the use of P2P mode, there are certain collision measures that must take place, and therefore add more time to our transmission.

The general protocol for the RF portion is:

1. Mobile waits for initiator
2. NFC device switches to Initiator mode, selects Active or Passive, and transfer speed
3. NFC device tests for external field. If field present, it will not active RF field
4. If no field detected, NFC device will activate it's own field
5. Exchange commands and responses

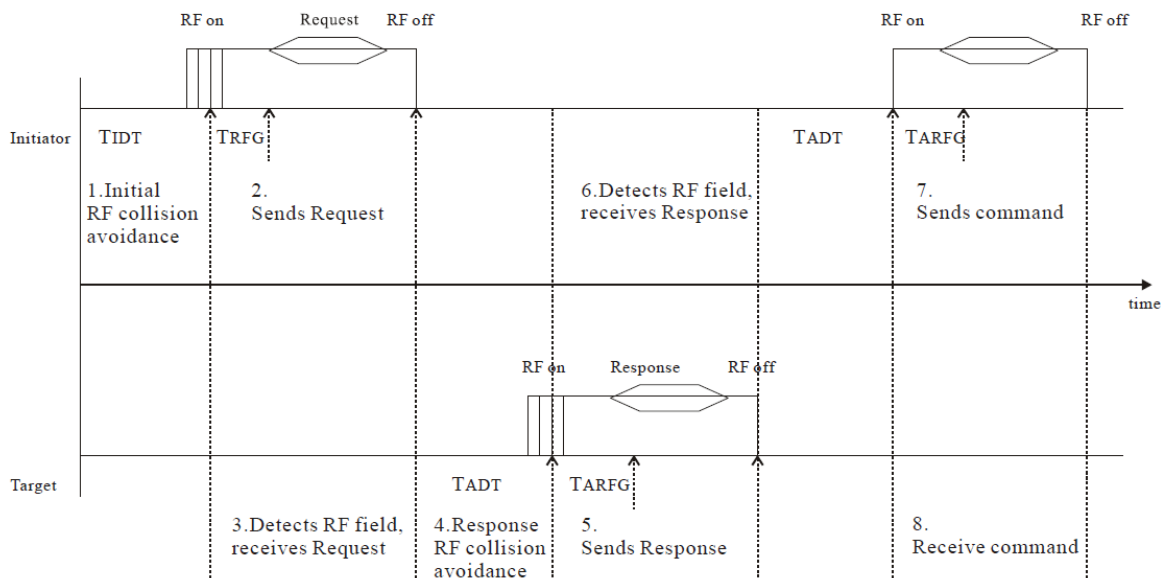


Figure 24: General protocol flow for initialization [10].

To calculate how long it will transfer, we needed to determine how much time it would take to deal with collision avoidance. Therefore each section of the points above were looked at.

Initial Collision Avoidance

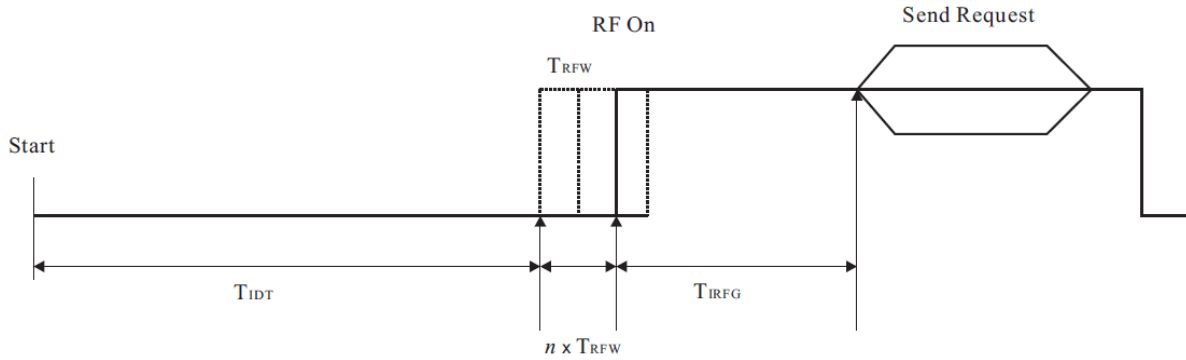


Figure 25: Initiator needs to confirm there is no other RF field [10].

Calculate max delay during initialization:

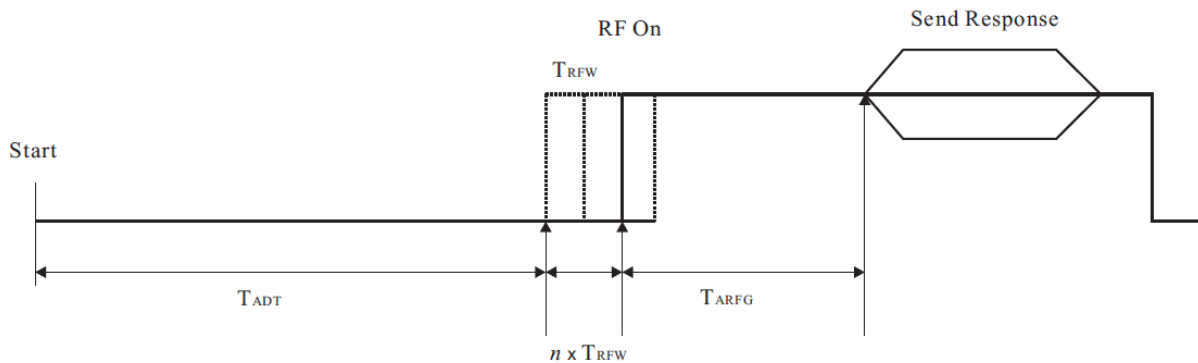
$$\begin{aligned}
 Delay_{max-init} &= (T_{IDT} + n + T_{RFW}) + T_{IRFG} \\
 &= \frac{4096}{13.54 * 10^6 \text{ Hz}} + 3 \left(\frac{512}{13.45 * 10^6 \text{ Hz}} \right) + 5 \text{ ms} \\
 &= 0.32 \text{ ms} + 0.11 \text{ ms} + 5 \text{ ms} \\
 &= 5.43 \text{ ms}
 \end{aligned}$$

Initiator Sends Request and Target Receives Request (Switches to Active Mode)

Calculate Command and Response Max Time:

$$Response_{max-time} = \frac{144 \text{ bits}}{424 \text{ kbps}} = 0.34 \text{ ms}$$

Response RF Collision Avoidance



$$\begin{aligned}
 \frac{768}{f_c} &< T_{ADT} < \frac{2559}{f_c} \\
 f_c &= 13.56 \text{ MHz}
 \end{aligned}$$

$$T_{ADT-MAX} = \frac{2559}{13.56 \text{ MHz}} = 188.72 \text{ us}$$

$$T_{RFW} = n * \frac{512}{f_c}$$

$$0 < n < 3$$

$$T_{RFW-max} = 3 * \frac{512}{13.56 \text{ MHz}} = 113.27 \text{ us}$$

$$T_{RFW} > \frac{1024}{f_c} = 75.52 \text{ us}$$

$$\begin{aligned} \text{Delay}_{max-response} &= 188.72 \text{ us} + 113.27 \text{ us} + 75.52 \text{ us} \\ &= 377.51 \text{ us} \\ &= 0.38 \text{ ms} \end{aligned}$$

In the end, the final total initialization time would be 6.87 ms, which is not significant.

Microcontroller + USB

Since we are going with Option 2 as defined in the Top-Level Design options document, we will need to investigate different micro controller units and determine if they meet all necessary requirements.

Module	Requirements
Memory	The average size of a receipt from our sample set was 77vkB. Therefore, we used an average PDF size with error margin as 100 kB. So, the MCU should have at least 100 kB of memory for receipt file + user/board code.
I/O	GPIO for LED I2C or SPI or UART for NFC controller communication UART communication with host computer
LED	Needs an LED for indicating status of device
Comms	Need some way of communicating with the host device. This was decided to be a USB.
Power	Planned on using USB so it must be below 5V/500mA

Component Selection

Microcontroller

The Raspberry Pi Pico with the RP2040 chip was chosen as the microcontroller

Name	Core	Mfg	Memory (needed 100k)	Supported NFC Controller Interfaces (I2C)	Serial port to Host	External IO Pins for LED	Voltage	3v3 Supply for NFCC?	Mounting Holes?	Price	Pass?
RASPBERRY PI PICO RP2040	ARM® Cortex®-M0+ Dual Core	Raspberry Pi	2M Flash 264 k SDRAM	I2C	Yes	Yes	5V	Yes	Yes	\$5.43	Yes
TRINKET M0 ATSAM D21E18 EVAL BRD	ARM® Cortex®-M0+	Adafruit	32kB RAM	I2C	?	5	5V	Yes	Yes	\$12.16	Maybe
TEENSY 3.5	ARM® Cortex®-M4	SparkFun	Flash 512k RAM 192k EEPROM 4k	3x I2C	Yes	Yes	5V	Yes	No	\$34.23	Yes

NFC Controller

The OM5577 board with an PN7120 NFC controller and antenna was chosen.

Name	Mfg	Integrated Antenna option	Speed at 424 kbps?	Supports NFCIP-1 Protocol?	Communication protocol with host MCU	Power rating	Helpful APIs?	Cost	Availability	Acceptable?

PN7150 on USB Dongle board link	MikroElektronika	Yes	Up to 424 kbps	NFCIP -1 (and many others)	I2C or NCI	3.3 V Or 5V from Vbat Max Current = 190 mA	Yes	\$39.40	Many	Maybe No accessible GPIO for LED Status Indicator
PN7120 on OM5577 board link	NXP	Yes	Up to 424 kbps	NFCIP -1 (and many others)	I2C or NCI	3.3 V or 5V from VBat Max Current = 180 mA	Yes	\$52.40	Many	Yes
PN532 on Adafruit breakout board	Adafruit	Yes	?	?	?	?	Yes	\$54.27	Slim	No No proper datasheet
ST25DV on STEVAL - SMART AG1 board link	STMicroelectronics	Yes	Less than 424 kbps	ISO/IEC 15693	I2C	3.3 V	No	\$25.43	Slim	No Not P2P ready
ST25R3911B on 113990	Seeed Technology Co., Ltd	Yes	424 kbps	INFCIP-1	I2C, SPI, USART	Power supply : 5V	No	\$29.87	Slim	Maybe No good backing

817 board link						Opera ting: 3.3V				commu nity
--	--	--	--	--	--	------------------------	--	--	--	---------------

Detailed Design

Tools Used

- For Hardware
 - Altium Designer
- For Software
 - FreeRTOS
 - TinyUSB
 - NFC Library (NXP-NCI & NDEF Libraries)
 - CMake (for building)
 - Raspberry Pi Pico Software Development Kit
 - OpenOCD (for debugging)
 - Minicom (for serial output)
 - Ubuntu 20.04 LTS

Hardware

This section describes the Interface Board that is necessary to connect the OM5577 and the Raspberry Pi Pico together.

Interface Board

The first step was to determine what the NFC Controller (OM5577) needed from the Raspberry Pico, which can be found in the “PN7120 NFC controller SBC kit user manual” [11].

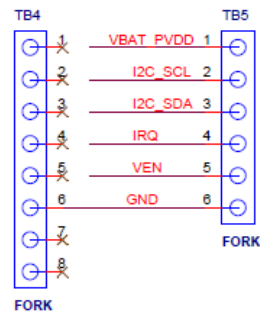


Figure 26: Schematic pinout of OM5577 (PN7120) [11]. This design will use the TB5 header, as the TB4 header is basically a mirror of TB5 but used for a BeagleBoard interface.

Internally, they are connected to the PN7120 as such:

PN7120

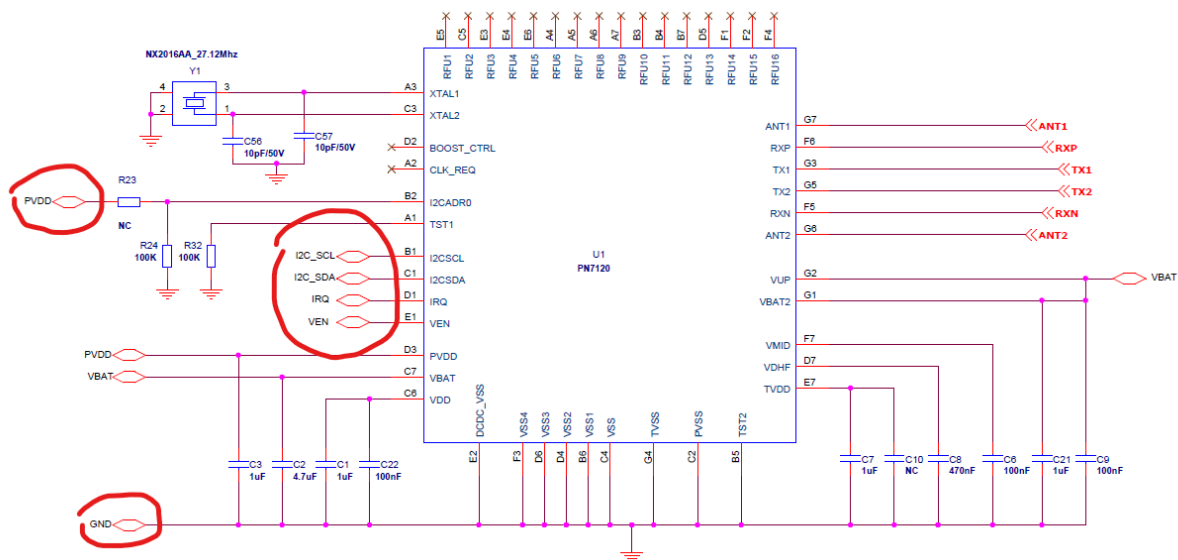


Figure 27: Internal wiring to the PN7120 [11]. Relevant pins are circled.

With a physical representation of the board shown in the following figures:

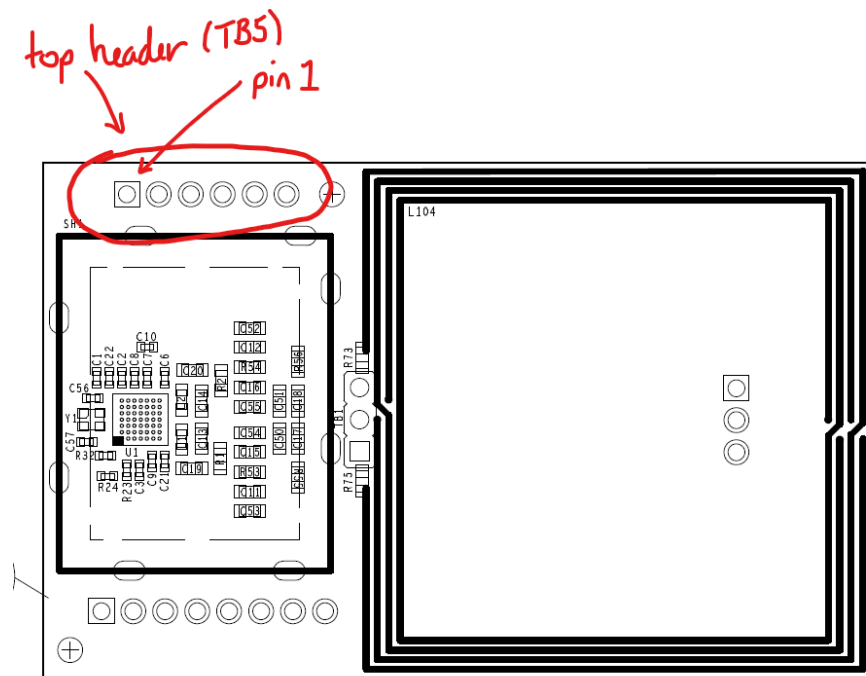


Figure 28: Physical pinout of OM5577 [11].

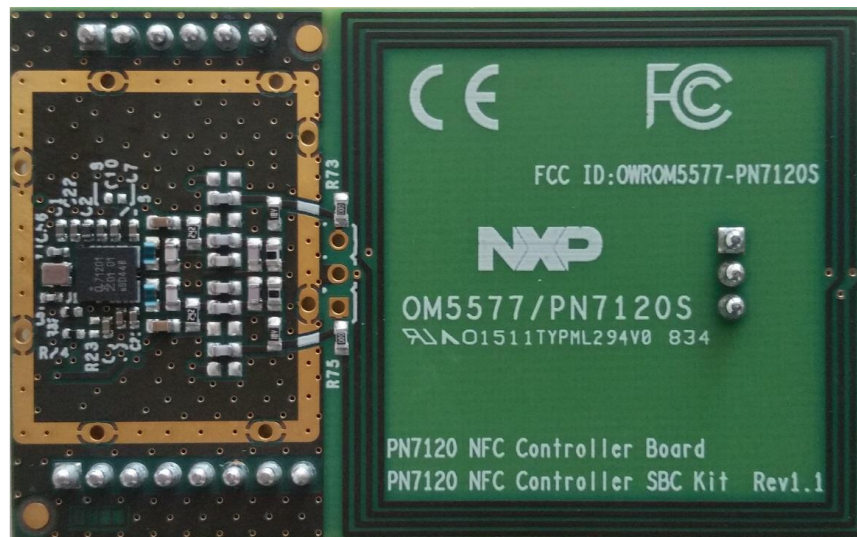


Figure 29: Actual view of the OM5577 (top view) [11].

Following is the pinout for the Raspberry Pi Pico:

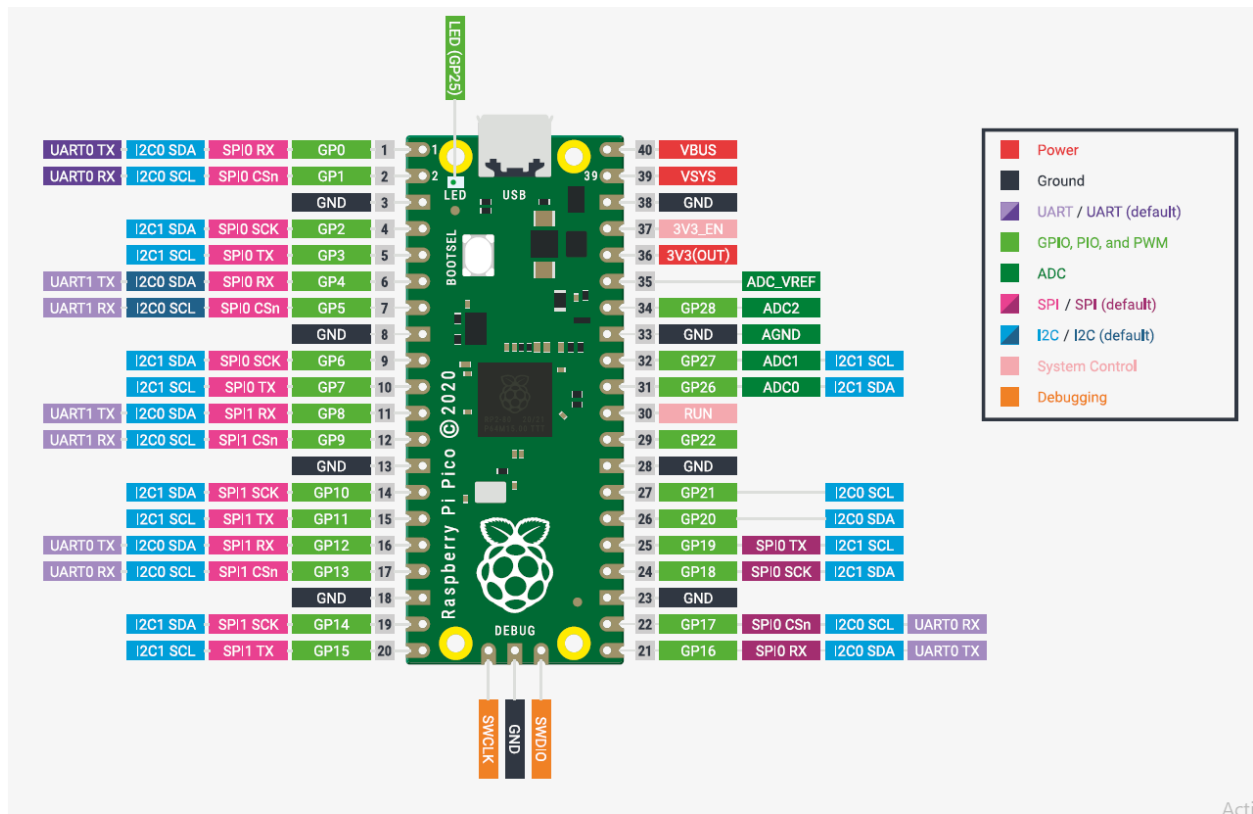


Figure 30: Pinout of the Raspberry Pi Pico [2].

As can be seen in Figure 26 and Figure 27, the OM5577 requires six connections to the Raspberry Pi Pico. We can determine the characteristics of each pin from the PN7120 datasheet [12].

Starting with Pin 1 (VBAT_PVDD), based on the table found below in the datasheet it is determined (since we will not be using a battery, therefore Vbat is not used) that pin 1 requires either a 1.65 to 1.95V supply, or a 3.0 to 3.6V supply. Since we are using the Raspberry Pi Pico that has a 3.3V supply (pin 36), we start with that.

Table 2: Shows the characteristics of VBAT_PVDD connected to the PN7120 [12].

Symbol	Parameter	Conditions	Min	Typ	Max	Unit
V _{BAT}	battery supply voltage	Card Emulation and Passive Target; V _{SS} = 0 V ^[1] ^[2]	2.3	-	5.5	V
		Reader, Active Initiator and Active Target; V _{SS} = 0 V ^[1] ^[2]	2.7	-	5.5	V
V _{DD}	supply voltage	internal supply voltage	1.65	1.8	1.95	V
V _{DD(PAD)}	V _{DD(PAD)} supply voltage	supply voltage for host interface				
		1.8 V host supply; V _{SS} = 0 V ^[1]	1.65	1.8	1.95	V
		3.3 V host supply; V _{SS} = 0 V ^[1]	3.0	-	3.6	V

To confirm this decision, it is important to look at the power capabilities of the Raspberry Pi Pico 3.3V pin to ensure that the PN7120 will not draw more power than can be provided.

The Raspberry Pi Pico data sheet states that “...it is recommended to keep the load on this pin less than 300mA” [13]. Since the PN7120 has a current limit threshold of 180mA [12], we can confirm that using the 3V3 supply from the Pico to power the PN7120 will suffice.

The next two pins are the I2C_SCL and I2C_SDA (I2C clock and data channel respectively). The Raspberry Pi Pico has two I2C blocks (I2C0 and I2C1). There is no difference between the two so we will use I2C0 and make our connections to the any of the I2C0 pin pairs on the Pico.

Pin 4 is an “interrupt request output” [12], meaning that if the mobile wants to send information to the NFC device, it needs to request it by sending a pulse to the interrupt request output pin. This pin will not be used, but we will connect it in the event that it is needed. It can be connected to any GPIO on the Raspberry Pi Pico that has internal pull-up resistors.

Pin 5 is used to reset the NFC controller. It can be mapped to any GPIO pin on the Raspberry Pi Pico.

Pin 6, ground, will be connected to one of the GND pins on the Raspberry Pi Pico.

Based on the above information, the pin map between the Pico and the NFC controller will look like this:

Table 3: Pin map between Raspberry Pi Pico and OM5577.

Raspberry Pi Pico		OM5577 (PN7120)	
Description	Pin	Pin	Description
3V3 (out)	36	1	VBAT/VDD(PAD): 3V3 supply voltage
I2C0 SCL	7	2	I2CSCL: I2C-bus serial clock input
I2C0 SDA	6	3	I2CSDA: I2C-bus serial data
GP6	9	4	IRQ: interrupt request output
GP7	10	5	VEN: reset pin
GND	33	6	GND: ground

An initial sketch of the interface board looks like this:

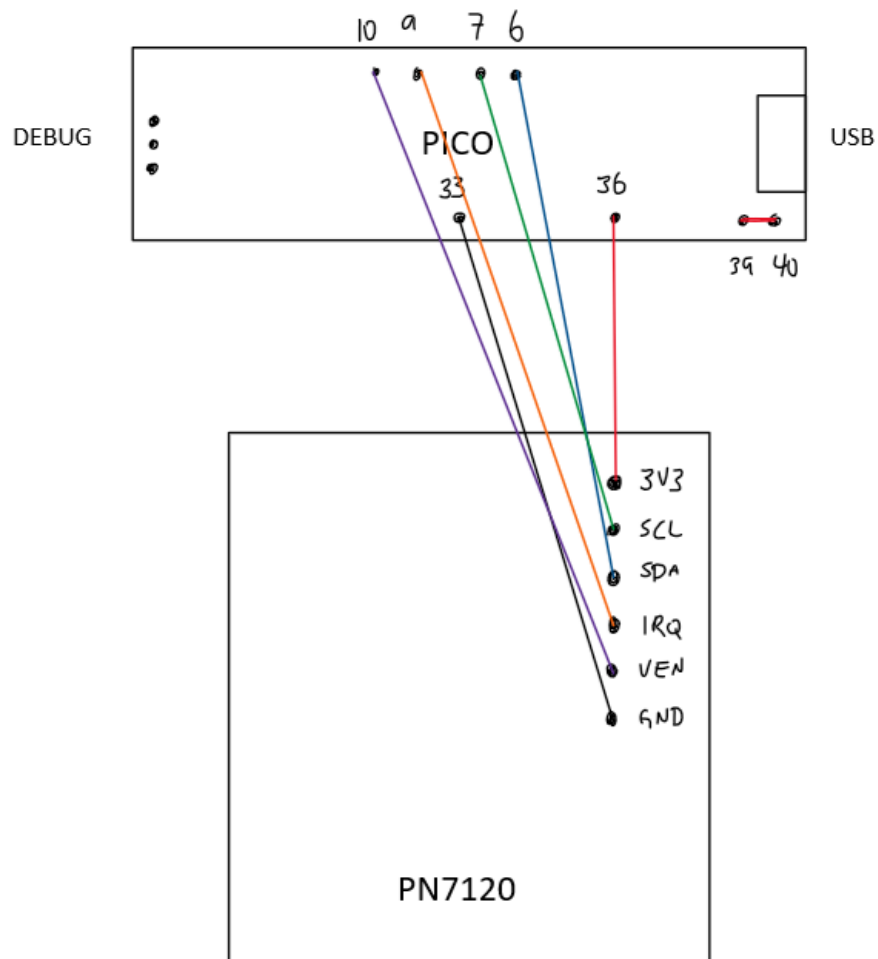


Figure 31: Initial sketch of the interface board. Note that pin 39 and pin 40 on the Raspberry Pi Pico can be shorted if not using a battery supply.

Schematic

The full schematic and PCB documentation can be found in the reference material. However, it suffices to show the diagrams here since they are not that large.

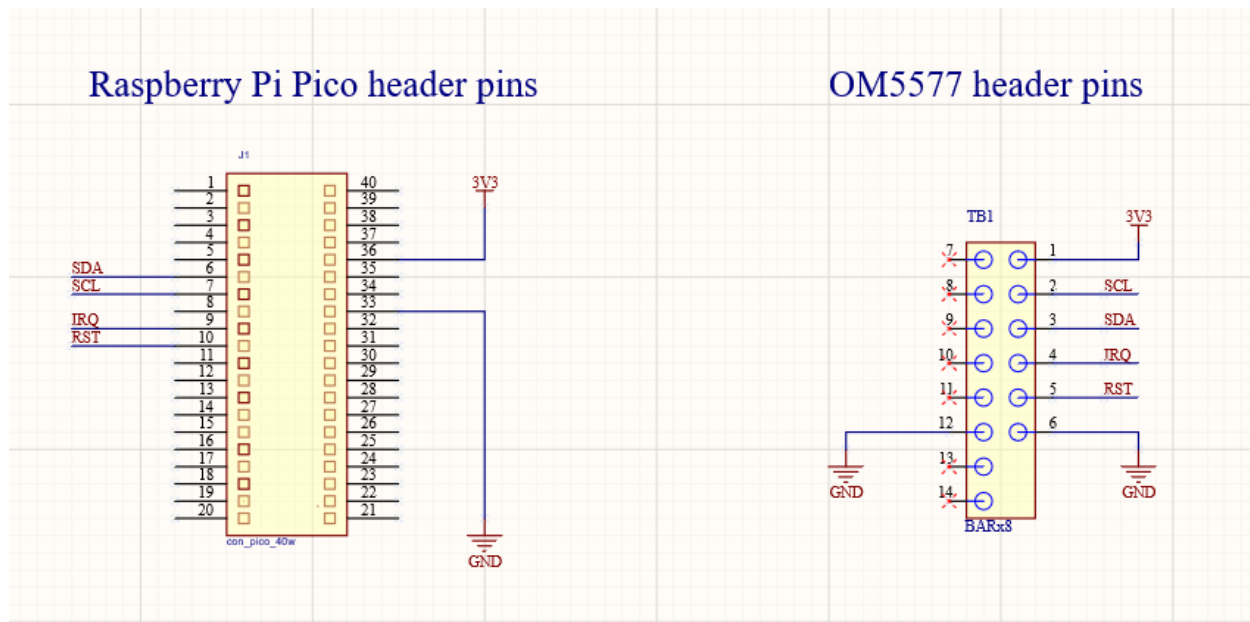


Figure 32: Schematic of the Interface Board between the Pico and the OM5577.

Thankfully, the schematic diagram for the Pico was provided by Raspberry Pi and can be found on their website under “Download design files” [14]. The OM5577 header pins need to be designed from scratch, since no schematic design file is provided. Even though pins 7 through 14 (except for GND) are not being used, I included them into the design for the sake of comprehensiveness and ease of understanding.

PCB Design

The PCB was a bit trickier to finalize than the schematic because there are no footprints available for either the Pico or the OM5577 board, therefore they must be designed from scratch.

The Pico footprint could be designed using the documentation provided by Raspberry Pi, which is shown in Figure 33. Based on this information it was relatively simple to design the header locations and mounting holes for the Raspberry Pi Pico and implement them into the Interface Board.

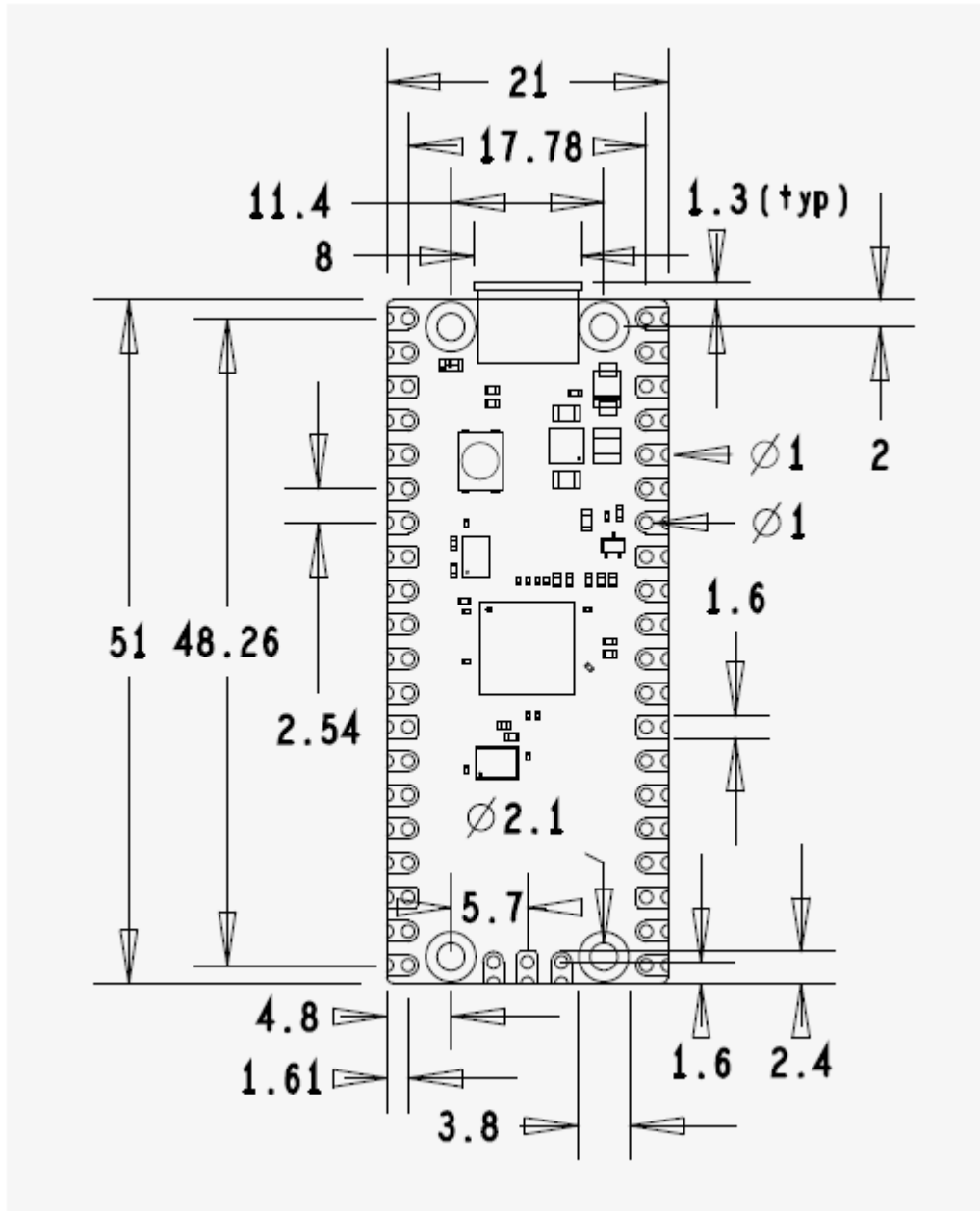


Figure 33: Mechanical design of Pico [13].

Since there are no mechanical drawings for the OM5577 board, it is impossible to determine the location of the header pins without purchasing the component and measuring directly. Therefore, the components was purchased, measured, and implemented into the interface board. The result of the PCB design is shown in Figure 34.

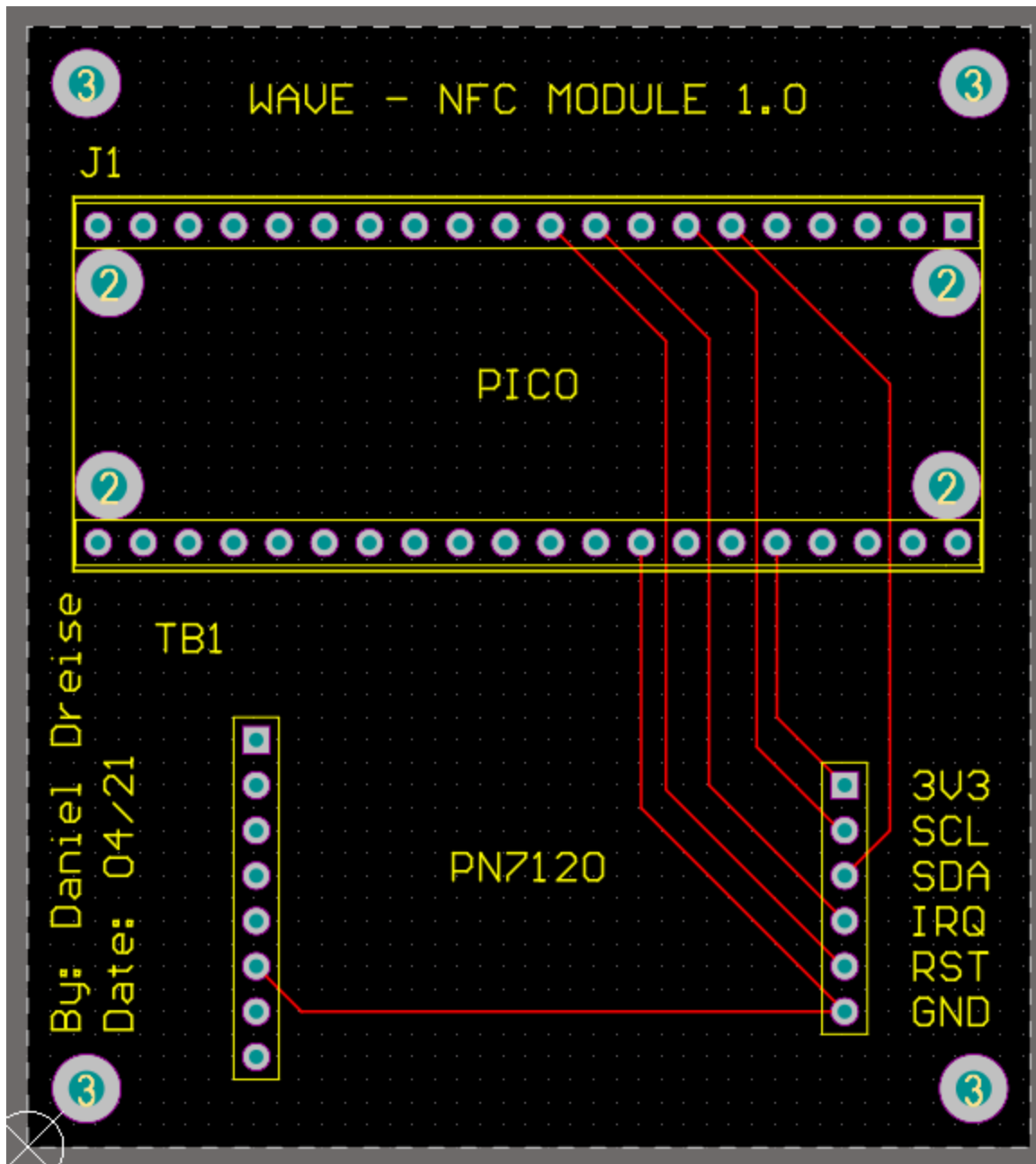


Figure 34: Final PCB design of the Interface Board.

This following table goes into detail regarding the components in the PCB design:

Component	Description
Vias (3)	Mounting holes to mount to a case
Vias (2)	Mounting holes for the Raspberry Pi Pico
J1	Raspberry Pi Pico header pinout footprint
TB1	OM5577 (PN7120) header pinout footprint
Traces	10 mil width (min width is ~5 mil for 3V3 trace)

This PCB design is based on manufacturer limitations from PCBWay (<https://www.pcbway.com/>). It can be designed for other manufacturers without much issue. It is a single layer board because there is no need for layers on the bottom or middle.

The PN7120 board will stick out from this platform (in the negative “y” direction). This is intentional because we do not want to have copper ground plan underneath the antenna, which would hinder the RF field.

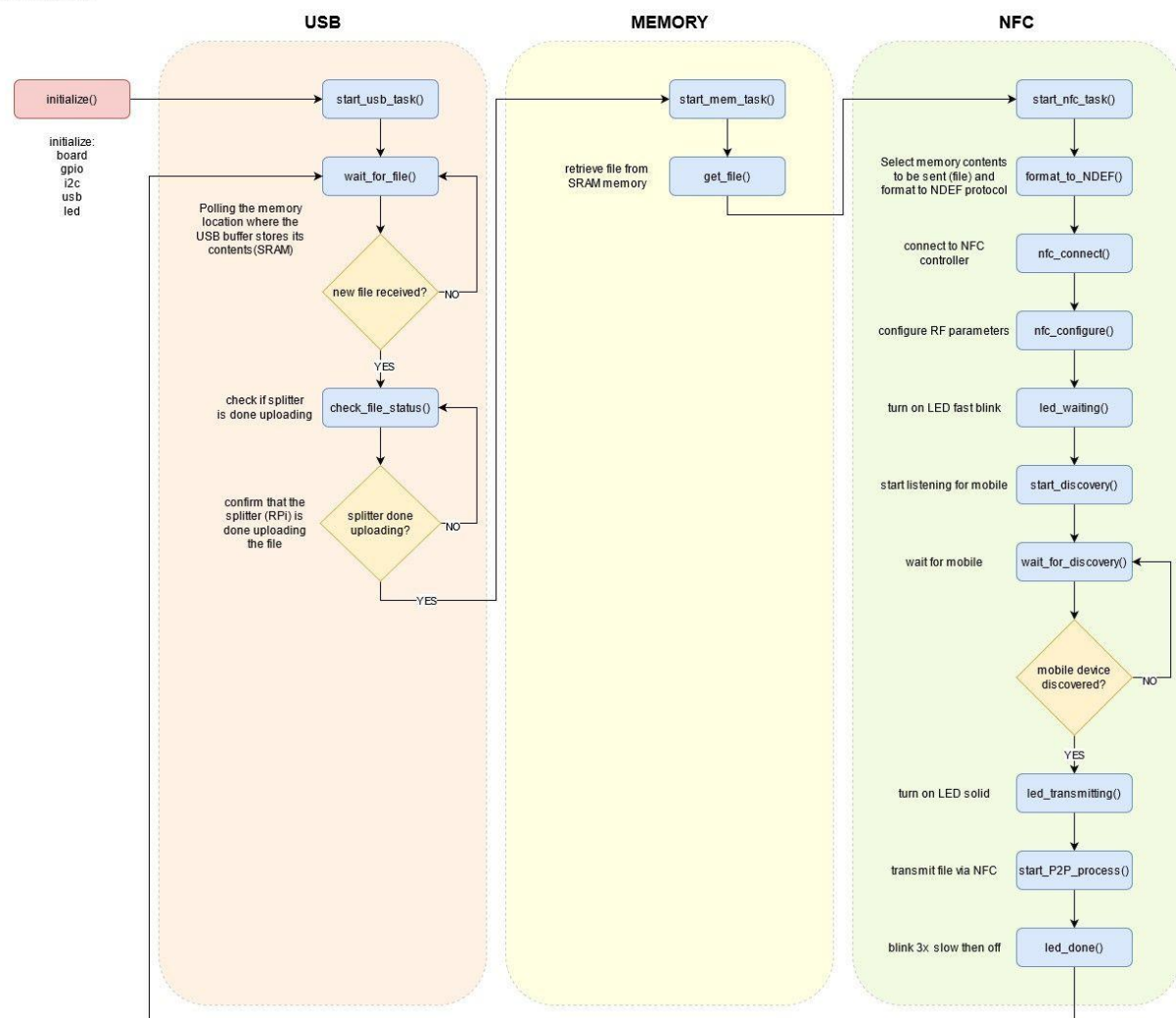
Also, it was intended that the PN7120 board would overlap the Pico to make the overall footprint smaller, however this would obstruct the LED on top of the Pico which is needed for the LED status indicator functionality. Therefore a “side-by-side” approach was implemented because it still fits within the 15cm x 15cm limitations.

Software

NFC Logic Diagram

By: Daniel Dreise

Date: April 14, 2021



The top-level view of the software logic diagram can be seen in the reference documentation (“NFC Logic Diagram”), which gives a good understanding of how all the different functional blocks work together. In it there are three different blocks which represent the three functional blocks:

1. USB (NFC to Splitter Interface)
2. Memory (Information Processing)
3. NFC (NFC to Mobile interface)

Before the functional blocks are discussed, it is important to note the major software development kits (SDK) and libraries used in this project. The following is the list with the best place to start for implementing them:

Library	Best place to start
Raspberry Pi Pico SDK	Raspberry Pi Pico C/C++ SDK: Libraries and tools for C/C++ development on RP2040 microcontrollers [15] [15]
NXP-NCI library	AN1190: NXP-NCI MCUXpresso example [16]
TinyUSB library	TinyUSB [17]

NFC to Mobile Interface

This functional block’s purpose is to handle all physical aspects of NFC including RF generation & reception, mobile RF field sensing, collision control, NFC protocol formatting, and reception of data from MCU for transmitting.

It consists of two portions:

1. NFC Controller (PN7120)
2. Antenna

For the purposes of this project, the antenna does not need to be tuned or configured, all necessary design is interfacing with and configuring the NFC Controller.

To interface with the NFC controller NXP has provided an NFC library which consists of two sub-libraries:

1. NDEF library
2. NXP-NCI library

The first of these libraries is used for packaging data into the proper NFC format, NDEF (NFC Data Exchange Format). Whereas the second is specifically used for interfacing with the NFC controller. A summary of the software stack between the Pico and the NFC controller is as follows:

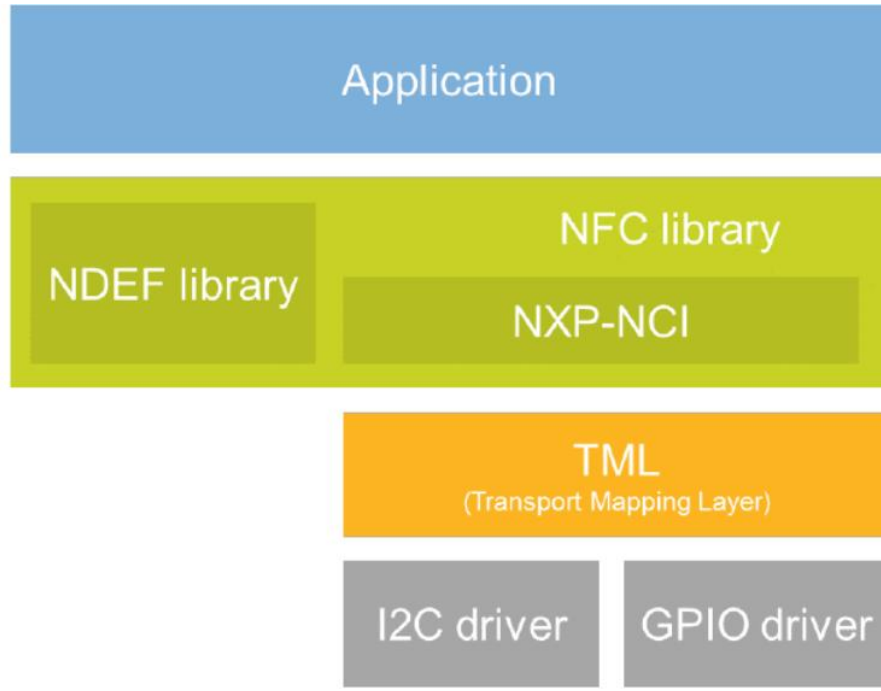


Figure 35: Software stack for NFC controller interface.

As can be seen, the application layer has access to the NDEF library for formatting data as well as the NXP-NCI library for interfacing with the NFC Controller. Therefore, this design will cover the Application, Transport Mapping Layer (TML), and the I2C and GPIO drivers.

Application Layer

The application layer is the top-level user code that utilizes the libraries for desired functions. The file source code structure for this project will look like this:

- main.c
 - nfc_task.c

The main() program will consist one task in the while(1) loop that will process all things related to NFC; nfc_task().

```

1  int main(void) {
2      while(1){
3          nfc_task()
4      }
5  }
```

Figure 36: Code snippet for nfc_task().

Where nfc_task() will perform the following actions:

Table 4: Functionsn called by nfc_task().

Description	Existing function name from libraries	Found in library
-------------	---------------------------------------	------------------

Setting up NDEF message	P2P_NDEF_SetMessage()	NDEF
Connecting to NFCC	NxpNci_Connect()	NXP-NCI
Configuring NFCC	NxpNci_ConfigureSettings()	NXP-NCI
Setting configuration mode	NxpNci_ConfigureMode	NXP-NCI
Starting RF discovery	NxpNci_StartDiscovery()	NXP-NCI
Wait for discovery notification	NxpNci_WaitForDiscoveryNotification()	NXP-NCI
Process P2P mode and transmit	NxpNci_ProcessP2pMode()	NXP-NCI

Transport Mapping Layer

All the above functions are not automatically directed to the proper I2C ports, which is what the Transport Mapping Layer is for; to connect the outputs of the NXP-NCI library functions to their respective drivers (either I2C or GPIO).

To do this we will need the following functions (keeping the names of these functions the same is paramount for successful mapping):

tml_Init()	
Description	Initialize the GPIO pins connected to VEN and IRQ
Parameters	void
Implementation	Set IRQ pin as INPUT
	Set VEN pin as OUTPUT
Return	SUCCESS

tml_Reset()	
Description	Reset the NFC controller
Parameters	void
Implementation	Set VEN pin HIGH
	Sleep
	Set VEN pin LOW
	Sleep
	Set VEN pin HIGH
Return	SUCCESS

tml_Tx()	
Description	Connection between NXP-NCI transmit function and I2C write function
Parameters	message buffer ; buffer length
Implementation	i2c_write_blocking()

Return	if failed – FAILED if successful – SUCCESS
--------	---

tml_Rx()	
Description	Connection between NXP-NCI receive function and I2C read function
Parameters	message buffer ; buffer length
Implementation	i2c_read_blocking()
Return	if failed – FAILED if successful – SUCCESS

tml_WaitForRx()	
Description	Waits for IRQ pin
Parameters	length of timeout before error
Implementation	while IRQ pin is low, keep looping until timeout reached, then error
Return	if failed – FAILED if successful – SUCCESS

tml_Connect()	
Description	Initializes TML and then reset
Parameters	void
Implementation	call tml_Init() call tml_Reset()
Return	none

tml_Send()	
Description	More intuitive function for sending data. Adds functionality to tell how many bytes were sent.
Parameters	data buffer ; buffer length ; bytes sent
Implementation	Call tml_Tx() if there is an error, set “bytes sent” to 0 If no error, set “bytes sent” to “buffer length”
Return	none

tml_Receive()	
Description	More intuitive function for receiving data. Adds functionality to tell how many bytes were sent.

Parameters	data buffer ; buffer length ; bytes received
Implementation	Call tml_WaitForRx() if error, set “bytes received” to 0 else Call tml_Rx() set “bytes sent” to “buffer length”
Return	none

Drivers

The I2C and GPIO drivers will be setup based on their desired functionality. The i2c.c will consist of one function:

i2c_Init()	
Description	Initialize the I2C
Parameters	void
Implementation	Enable I2C0 block Set clock speed Set GPIO function of default SDA pin to I2C Set GPIO function of default SCL pin to I2C Enable pull-up resistor to SDA pin Enable pull-up resistor to SCL pin Enable I2C IRQ
Return	Success or Failure

The functions for reading and writing from the I2C bus are pre-configured in the Pico SDK library as:

- i2c_write_timeout_us
- i2c_read_timeout_us

These functions will be setup to be used directly by the NXP-NCI and NDEF library functions.

The gpio.c will consist of one function as well:

gpio_Init()	
Description	Initialize the GPIO
Parameters	void
Implementation	Enable LED GPIO

	Enable NFCC Reset GPIO
	Enable NFCC IRQ GPIO
Return	Success or Failure

Test Plans

The following test plans will be implemented to ensure robustness, verification, and validation of the functionality of this functional block.

Functional Block	NFC to Mobile interface
Unit Under Test	NFC - Format data to NDEF
Function	format_to_NDEF()
Success Condition	Correct file location is passed to function.
Procedure	<ol style="list-style-type: none"> 1. Build and Flash debug version onto Pico 2. Add file to mass storage location on Host (Splitter) 3. Check serial debug log for memory location print <ol style="list-style-type: none"> 1. Confirm that memory location sent to this function is the same referred to in USB functional block

Functional Block	NFC to Mobile interface
Unit Under Test	NFC – Configuration
Function	NxpNci_ConfigureSettings() ; NxpNci_ConfigureMode()
Success Condition	Correct configuration is set *Note: Configuration is defined in Nfc_settings.h
Procedure	<ol style="list-style-type: none"> 1. Build and Flash debug version onto Pico 2. Add file to mass storage location on Host (Splitter) 3. Check serial debug log for configuration settings <ol style="list-style-type: none"> 1. Confirm that the mode is P2P

Functional Block	NFC to Mobile interface
Unit Under Test	NFC – Mobile device detected
Function	NxpNci_StartDiscovery() NxpNci_WaitForDiscoveryNotification()
Success Condition	Mobile device is detected
Procedure	<ol style="list-style-type: none"> 1. Build and Flash debug version onto Pico 2. Add file to mass storage location on Host (Splitter) 3. Check serial debug log for successful detection

	1. Confirm that the log shows mobile detection
--	--

Functional Block	NFC to Mobile interface
Unit Under Test	NFC – File transferred
Function	NxpNci_ProcessP2pMode()
Success Condition	File is successfully transferred
Procedure	<ol style="list-style-type: none"> 1. Build and Flash debug version onto Pico 2. Add file to mass storage location on Host (Splitter) 3. Check serial debug log for successful transfer <ol style="list-style-type: none"> 1. Confirm that the log shows the file has been transferred 4. Check mobile app (NFC Tools)* for successful file transmission

*Note: <https://play.google.com/store/apps/details?id=com.wakdev.wdnfc&hl=en&gl=US>

NFC to Splitter Interface

The NFC to Splitter Interface includes all interaction and power management between the NFC device and the Splitter. These include data transfer via USB, and the power supply for the NFC device which is also transmitted via USB. This results in both power and data being transmitted by the same USB cable/port.

The Raspberry Pi Pico utilizes a micro-USB 2.0 Full Speed (12 Mbps) device which can be configured/programmed to different use cases (classes). Some of these include:

- Communication Class (USB-CDC). The USB can be used to communicate serially with host computer.
- Mass Storage Class (MSC). In this configuration, the Pico will look like an external drive to the host computer.
- Hardware Interface Device (HID). It will connect to a host computer and look like a hardware device (such as a mouse or keyboard).

During the initial investigation into the Pico's USB capabilities, it was not clear if the Pico's USB needed to be customized, or what kind of USB class would work best for the project. All that was needed was for data transfer to exist between the Pico and the Splitter seamlessly during runtime. It seemed that either USB-CDC or MSC would work best for the project. However, a few calculations later it became abundantly clear that communicating serially would be far too slow to attain our file transfer speed goals, therefore MSC was chosen in order to maximize the throughput.

Currently, there are two options for developing the USB peripheral portion for this project which allows for this level of customization.

1. Low-level hardware design
2. Third-party software (TinyUSB)

There is some decent documentation provided by Raspberry Pi that shows the process for initializing and customizing the Pico, but they do not cover setting it up as a mass storage device. The design calls for mass storage device to make it easy for the Splitter to transfer files to the NFC device. It also doesn't go into file transfers that are over the max buffer size. On the other hand, "TinyUSB" has recently released support for the Pico which includes setting it up as a mass storage device, with fragmentation of packets built into the application. The next sections will go into how to implement TinyUSB's mass storage class configuration.

Data Transfer (TinyUSB)

The TinyUSB source repository can be found at <https://github.com/hathach/tinyusb>. The libraries from this directory that are included in the build tree using CMake are:

- tinyusb_device
- tinyusb_board

This can be setup in a CMakeLists.txt which resides in the main source directory (along with main.c). The setup looks like this:

```

1  # Create INTERFACE libraries
2  add_library(usb INTERFACE)
3
4  # Target the sources that are to be included in this library
5  target_sources(usb INTERFACE
6      ${CMAKE_CURRENT_LIST_DIR}/usb/usb_descriptors.c
7      ${CMAKE_CURRENT_LIST_DIR}/usb/usb_msc.c
8      ${CMAKE_CURRENT_LIST_DIR}/usb/tusb_config.h
9  )
10
11 # Target other library dependencies
12 target_link_libraries(usb INTERFACE
13     pico_stdlib
14     tinyusb_device
15     tinyusb_board
16 )
17
18
19
20 add_executable(main
21     main.c
22 )
23
24 target_include_directories(main PRIVATE ${CMAKE_CURRENT_LIST_DIR})
25
26 # Pull in our pico_stdlib which pulls in commonly used features
27 target_link_libraries(main PRIVATE
28     pico_stdlib
29     hardware_i2c
30     usb
31 )
32 # create map/bin/hex file etc.
33 pico_add_extra_outputs(main)
34
35 add_subdirectory(usb)

```

Figure 37: Code snippet build setup using CMake for adding TinyUSB libraries.

The three main files that are used for configuring TinyUSB are:

File	Description
tusb_config.h	This file is where the use can configure different settings for the USB

usb_descriptors.c	This is used for setting up the USB with proper descriptions for the host USB to understand.
usb_msc.c	This is used for properly setting up the Mass Storage configuration

Now that the proper files are added into the library, the rest will go into detail how the individual files will be setup. The code snippets are found in the resources folder attached to this report.

A demonstration of the Pico working as a mass storage device can be found in the Appendix folder under NFC-Device > usb_mass_storage_sample.mp4.

Power Supply

By default, USB will deliver 5V at 100 mA. However, since our application will require more current, it is necessary to configure the USB to ask for 500 mA. This configuration happens within the usb_descriptors.c file. This part of the design process was not completed during the design phase and will need to be implemented in the next phase (semester)

USB and Memory

This section of software is regarding how the information retrieved by the USB task is sent to the NFC controller for processing. Since the memory is going to be a shared space between the USB buffer and the NFC controller functions, it is imperative that the necessary blocks are put in place to avoid race conditions. However, since we are only using a single core system, these blocks don't need to be IPCs, but rather they'll just need to flow in a logical manner that won't overlap the shared memory.

If we find the speed of the system is too slow, we may implement the Pico's second core. If so, we would need to implement an IPC. The Pico's SDK provides IPC (Interprocess Communication) library functions to assist. They include:

- critical_section.c
- lock_core.c
- mutex.c
- sem.c

For now, we will follow a logical approach for the contents of the USB buffer to be sent to the NFC controller when applicable.

It's relatively simple, once the USB has retrieved the full file and has stored it into RAM, the get_file() function will redirect that information to be formatted into NDEF format. It will also determine the size of the file.

Essentially, the pointer location of the contents in RAM will be copied and use as the paramter for the *pMessage, and it's size for the Message_size.

```
P2P_NDEF_SetMessage(unsigned char *pMessage, unsigned short Message_size,
void *pCb);
```

The USB driver will store information using the callback function:

```
int32_t tud_msc_write10_cb(uint8_t lun, uint32_t lba, uint32_t offset,
uint8_t* buffer, uint32_t bufsize)
```

This function will be invoked when the USB driver receives a “WRITE10” command from the USB host. The “lba” that provided will be the Logical Block Address in the mass storage FAT file.

The FAT file will be configured to FAT12. With block sizes of 512 bytes, and a file size of 100kB (we have room for 256kB of RAM), we will aim at using 200kB of RAM for storage. That equals ~390 blocks. Therefore, we will partition 390 blocks when initializing the storage space.

Much of this code will be taken from TinyUSB open source software.

The TinyUSB MSC is setup with Fat12. To get the necessary information from the Fat12 directory, retrieve the file, and then generate an NDEF message, it required parsing through the Fat12 Root Directory to retrieve the location and size of the file contents (stored in Data Area).

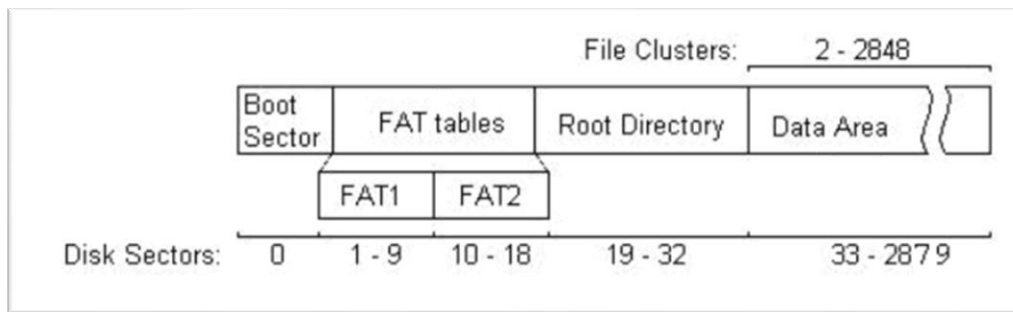


Figure 38: Fat12 memory allocation table.

The data stored in the Root Directory are in 32 bit entries, which have all the necessary information for a file. I created a struct to store the necessary information. The most important of these are the file_name, file_ext, file_location, and file_size.


```

struct Receipts {
    char file_name[8];           // Offset 0x00
    char file_ext[3];           //      0x08
    char file_attr[1];          //      0x0B
    char reserved[1];           //      0x0C
    char create_time_fine[1];    //      0x0D
    char create_time[2];        //      0x0E
    char create_date[2];        //      0x10
    char last_accessed[2];       //      0x12
    char ea_index[2];           //      0x14
    char last_modified_time[2];  //      0x16
    char last_modified_date[2]; //      0x18
    char file_location[2];       //      0x1A
    char file_size[4];          //      0x0C
} receipts;

```

Figure 39: Struct for holding fat12 root directory data entry.

Implementation

Hardware



Figure 40: Final device showing Raspberry Pi Pico, OM5577 NFC board, and the interface board underneath.

Software

The codebase for the implementation of the NFC device is too large to show here. Therefore, the software can be found at the Github repository <https://github.com/koolohms/nfc/tree/main/src>.

Mobile Android Application

High Level Design

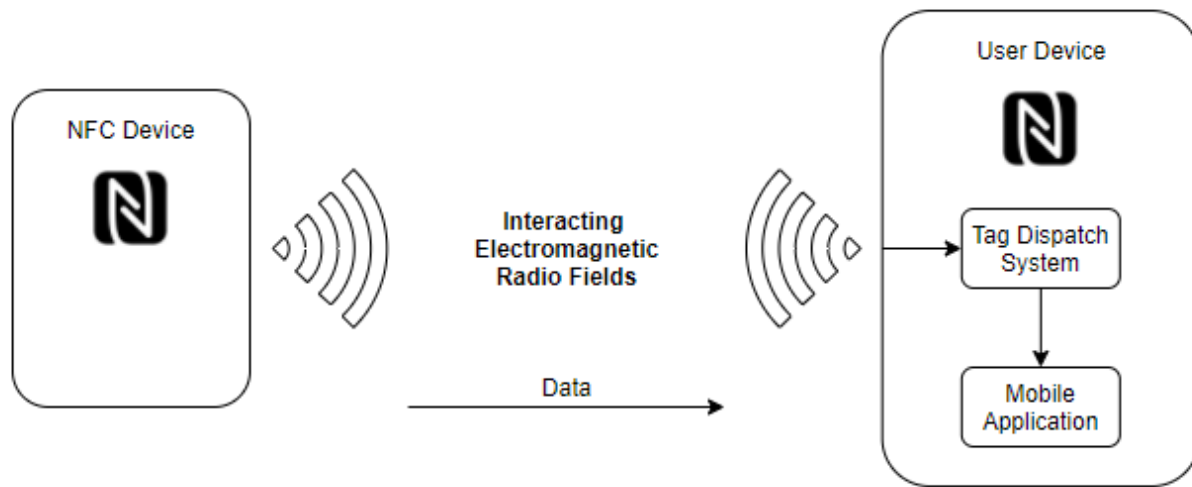


Figure 41: High Level Mobile Design

Mobile Platform

Android vs. iOS

- Android
- iOS

- ✓ Design Decision: Mobile solution will support Android platform only

Integrated Development Environment

Android Studio

- The official IDE for Google's Android OS

Features

- Visual Layout Editor
 - <https://developer.android.com/studio/write/layout-editor>
 - The Layout Editor enables you to quickly build layouts by dragging UI elements into a visual design editor instead of writing layout XML by hand. The design editor can preview your layout on different Android devices and versions, and you can dynamically resize the layout to be sure it works well on different screen sizes.
 - The Layout Editor is especially powerful when building a layout with `ConstraintLayout`, a layout manager that is compatible with Android 2.3 (API level 9) and higher.

- ✓ Design decision: Using Android Studio for mobile application development

Supported Programming Languages

Android studio supports development in Java, Kotlin, and C++.

- Kotlin
 - A relatively new programming language with modern features
 - Good choice for developing server-side applications
 - Allows users to write concise and expressive code
 - No experience
- Java
 - The official language for Android development
 - Expected to have the most support
 - No experience
- C++
 - General-purpose programming language
 - Extension of the C programming language
 - Most familiar

✓ Design decision: Using Java for code development

Application Components

There are four different types of app components [18]:

1. Activities
2. Services
3. Broadcast receivers
4. Content providers

Activities

An activity is the entry point for interacting with the user.

Services

A service is a general-purpose entry point for keeping an app running in the background for all kinds of reasons.

Manifest File declares:

- Components
 - Activities
- Component capacities
 - Intent filters
- App requirements
- Permissions

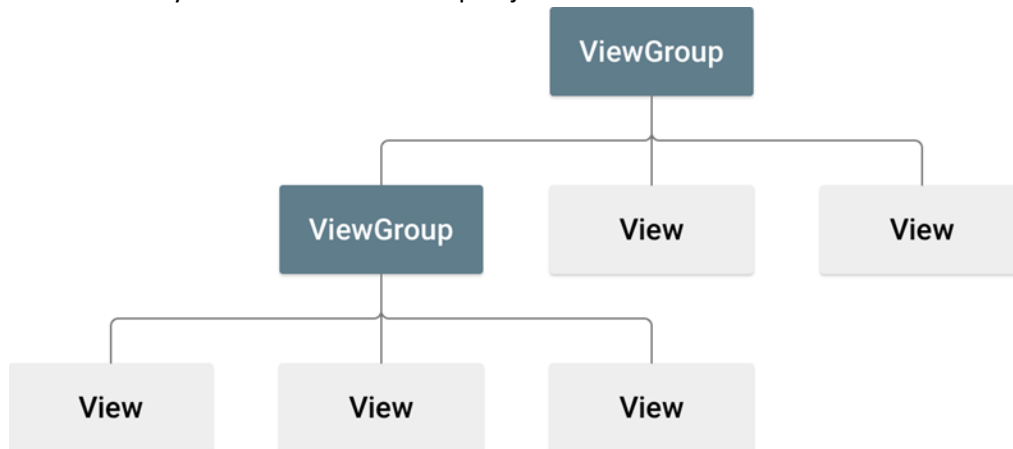
```

36  <activity android:name=".MainActivity">
37      <intent-filter>
38          <action android:name="android.intent.action.MAIN" />
39
40          <category android:name="android.intent.category.LAUNCHER" />
41      </intent-filter>
42  </activity>

```

Layouts [19]

- Defines the structure for a user interface in your app
- Uses a hierarchy of View and ViewGroup objects



ViewGroup (“Layouts”)

- LinearLayout
- RelativeLayout
- DrawerLayout

Linear Layout



Relative Layout



List View



View (“Widgets”)

- TextView
- ImageView

NFC

Added in API level 9, `android.nfc` provides access to Near Field Communication (NFC) functionality, allowing applications to read NDEF message in NFC tags. A "tag" may actually be another device that appears as a tag [20].

The API has the following classes:

- `NfcManager`
 - This is the high-level manager, used to obtain this device's `NfcAdapter`. You can acquire an instance using `getSystemService(Class)`.
- `NfcAdapter`

- This represents the device's NFC adapter, which is your entry-point to performing NFC operations. You can acquire an instance with `getDefaultAdapter()`, or `getDefaultAdapter(android.content.Context)`.
- **NdefMessage**
 - Represents an NDEF data message, which is the standard format in which "records" carrying data are transmitted between devices and tags. Your application can receive these messages from an `ACTION_TAG_DISCOVERED` intent.
- **NdefRecord**
 - Represents a record, which is delivered in a `NdefMessage` and describes the type of data being shared and carries the data itself.

Table 5: Descriptions of android.nfc classes

Class	Description
NdefMessage	Represents an immutable NDEF Message.
NdefRecord	Represents an immutable NDEF Record.
NfcAdapter	Represents the local NFC adapter.
NfcEvent	Wraps information associated with any NFC event.
NfcManager	High level manager used to obtain an instance of an <code>NfcAdapter</code> .
Tag	Represents an NFC tag that has been discovered.

Android has the most support for the NDEF standard, which is defined by the NFC Forum. Android also supports other types of tags that do not contain NDEF data. Working with these other types of tags involves writing your own protocol stack to communicate with the tags. It is recommended to use NDEF when possible for ease of development and maximum support for Android-powered devices [21].

✓ Design Decision: Reading NDEF data from an NFC tag

Android devices are usually looking for NFC tags when the screen is unlocked and NFC is enabled in the device's Settings menu. When an NFC tag is discovered, the desired behavior is to have the most appropriate activity handle the intent automatically. This is accomplished by the tag dispatch system.

Tag Dispatch System

The tag dispatch system analyzes scanned NFC tags, parses them, and attempts to locate applications that are interested in the scanned data. It does this by:

1. Parsing the NFC tag and figuring out the MIME type or a URI that identifies the data payload in the tag.
2. Encapsulating the MIME type or URI and the payload into an intent.
3. Starting an activity based on the intent.

How NFC tags are mapped to MIME types and URIs

NDEF data is encapsulated inside a message (`NdefMessage`) that contains one or more records (`NdefRecord`). When an Android-powered device scans an NFC tag containing NDEF formatted data, it parses the message and tries to figure out the data's MIME type or identifying URI. To do this, the

system reads the first `NdefRecord` inside the `NdefMessage` to determine how to interpret the entire NDEF message. In a well-formed NDEF message, the first `NdefRecord` contains the following fields:

- 3-bit TNF (Type Name Format)
 - Indicates how to interpret the variable length type field. Valid values are described in Table 7.
- Variable length type
 - Describes the type of the record. If using `TNF_WELL_KNOWN`, use this field to specify the Record Type Definition (RTD). Valid RTD values are described in Table 8.
- Variable length ID
 - A unique identifier for the record. This field is not used often, but if you need to uniquely identify a tag, you can create an ID for it.
- Variable length payload
 - The actual data payload that you want to read or write. An NDEF message can contain multiple NDEF records, so do not assume the full payload is in the first NDEF record of the NDEF message.

The tag dispatch system uses the TNF and type fields to try to map a MIME type or URI to the NDEF message. If successful, it encapsulates that information inside of an `ACTION_NDEF_DISCOVERED` intent along with the actual payload. However, there are cases when the tag dispatch system cannot determine the type of data based on the first NDEF record. This happens when the NDEF data cannot be mapped to a MIME type or URI, or when the NFC tag does not contain NDEF data to begin with. In such cases, a `Tag` object that has information about the tag's technologies and the payload are encapsulated inside of an `ACTION_TECH_DISCOVERED` intent instead.

Table 6: Supported TNFs and their mappings

Type Name Format (TNF)	Mapping
<code>TNF_ABSOLUTE_URI</code>	URI based on the type field.
<code>TNF_EMPTY</code>	Falls back to <code>ACTION_TECH_DISCOVERED</code> .
<code>TNF_EXTERNAL_TYPE</code>	URI based on the URN in the type field. The URN is encoded into the NDEF type field in a shortened form: <code><domain_name>:<service_name></code> . Android maps this to a URI in the form: <code>vnd.android.nfc://ext/<domain_name>:<service_name></code> .
<code>TNF_MIME_MEDIA</code>	MIME type based on the type field.
<code>TNF_UNCHANGED</code>	Invalid in the first record, so falls back to <code>ACTION_TECH_DISCOVERED</code> .
<code>TNF_UNKNOWN</code>	Falls back to <code>ACTION_TECH_DISCOVERED</code> .
<code>TNF_WELL_KNOWN</code>	MIME type or URI depending on the Record Type Definition (RTD), which you set in the type field. See Table 8 for more information on available RTDs and their mappings.

Table 7: Supported RTDs for `TNF_WELL_KNOWN` and their mappings

Record Type Definition (RTD)	Mapping
------------------------------	---------

RTD_ALTERNATIVE_CARRIER	Falls back to ACTION_TECH_DISCOVERED.
RTD_HANDOVER_CARRIER	Falls back to ACTION_TECH_DISCOVERED.
RTD_HANDOVER_REQUEST	Falls back to ACTION_TECH_DISCOVERED.
RTD_HANDOVER_SELECT	Falls back to ACTION_TECH_DISCOVERED.
RTD_SMART_POSTER	URI based on parsing the payload.
RTD_TEXT	MIME type of text/plain.
RTD_URI	URI based on payload.

How NFC tags are dispatched to applications

After the tag dispatch system is creates an intent that encapsulates the NFC tag and its identifying information, it sends the intent to an interested application that filters for the intent. If more than one application can handle the intent, the Activity Chooser is presented so the user can select the Activity. The tag dispatch system defines three intents, which are listed in order of highest to lowest priority:

1. ACTION_NDEF_DISCOVERED: This intent is used to start an Activity when a tag that contains an NDEF payload is scanned and is of a recognized type. This is the highest priority intent, and the tag dispatch system tries to start an Activity with this intent before any other intent, whenever possible.
2. ACTION_TECH_DISCOVERED: If no activities register to handle the ACTION_NDEF_DISCOVERED intent, the tag dispatch system tries to start an application with this intent. This intent is also directly started (without starting ACTION_NDEF_DISCOVERED first) if the tag that is scanned contains NDEF data that cannot be mapped to a MIME type or URI, or if the tag does not contain NDEF data but is of a known tag technology.
3. ACTION_TAG_DISCOVERED: This intent is started if no activities handle the ACTION_NDEF_DISCOVERED or ACTION_TECH_DISCOVERED intents.

The basic way the tag dispatch system works is as follows:

1. Try to start an Activity with the intent that was created by the tag dispatch system when parsing the NFC tag (either ACTION_NDEF_DISCOVERED or ACTION_TECH_DISCOVERED).
2. If no activities filter for that intent, try to start an Activity with the next lowest priority intent (either ACTION_TECH_DISCOVERED or ACTION_TAG_DISCOVERED) until an application filters for the intent or until the tag dispatch system tries all possible intents.
3. If no applications filter for any of the intents, do nothing.

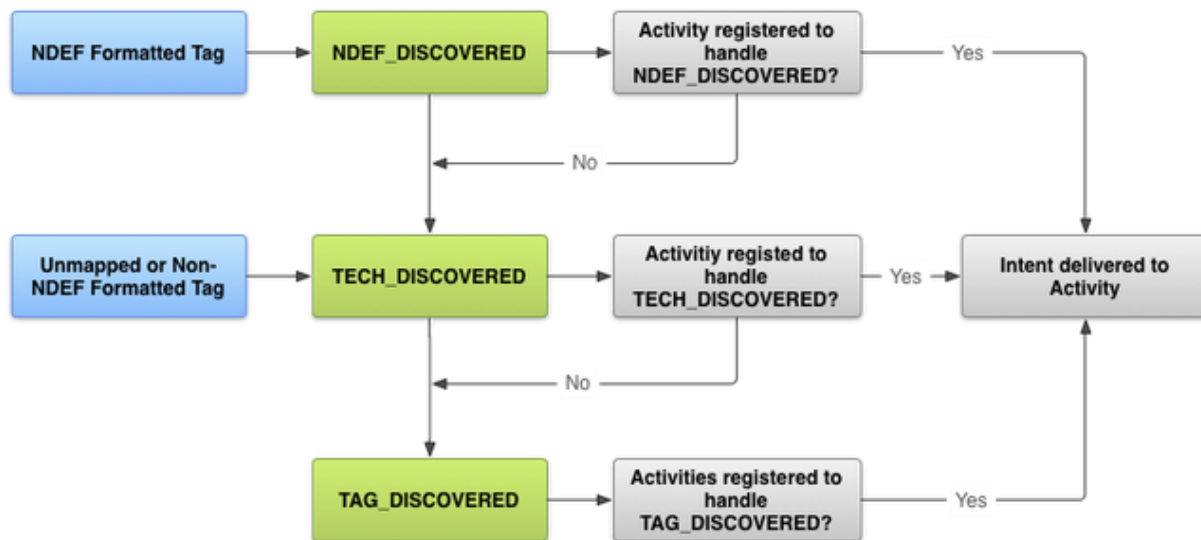


Figure 42: Tag Dispatch System

Whenever possible, work with NDEF messages and the `ACTION_NDEF_DISCOVERED` intent, because it is the most specific out of the three. This intent allows you to start your application at a more appropriate time than the other two intents, giving the user a better experience.

✓ Design Decision: Mobile application will filter for the `ACTION_NDEF_DISCOVERED` intent
Request NFC Access in the Android Manifest

Before you can access a device's NFC hardware and properly handle NFC intents, declare these items in your `AndroidManifest.xml` file:

- The NFC `<uses-permission>` element to access the NFC hardware:

```
1 <uses-permission android:name="android.permission.NFC" />
```

- The minimum SDK version that your application can support. API level 9 only supports limited tag dispatch via `ACTION_TAG_DISCOVERED`, and only gives access to NDEF messages via the `EXTRA_NDEF_MESSAGES` extra. No other tag properties or I/O operations are accessible. API level 10 includes comprehensive reader/writer support as well as foreground NDEF pushing, and API level 14 provides an easier way to push NDEF messages to other devices with Android Beam and extra convenience methods to create NDEF records.

```
1 <uses-sdk android:minSdkVersion="10"/>
```

- ✓ Design Decision: Minimum SDK version support is API level 10
- The `uses-feature` element so that your application shows up in Google Play only for devices that have NFC hardware:

```
1 <uses-feature android:name="android.hardware.nfc" android:required="true" />
```

- ✓ Design Decision: Application only available for devices that have NFC hardware

Filter for NFC Intents

To start your application when an NFC tag that you want to handle is scanned, your application can filter for one, two, or all three of the NFC intents in the Android manifest. However, you usually want to filter for the `ACTION_NDEF_DISCOVERED` intent for the most control of when your application starts. The `ACTION_TECH_DISCOVERED` intent is a fallback for `ACTION_NDEF_DISCOVERED` when no applications filter for `ACTION_NDEF_DISCOVERED` or for when the payload is not NDEF. Filtering for `ACTION_TAG_DISCOVERED` is usually too general of a category to filter on. Many applications will filter for `ACTION_NDEF_DISCOVERED` or `ACTION_TECH_DISCOVERED` before `ACTION_TAG_DISCOVERED`, so your application has a low probability of starting. `ACTION_TAG_DISCOVERED` is only available as a last resort for applications to filter for in the cases where no other applications are installed to handle the `ACTION_NDEF_DISCOVERED` or `ACTION_TECH_DISCOVERED` intent.

ACTION_NDEF_DISCOVERED

To filter for `ACTION_NDEF_DISCOVERED` intents, declare the intent filter along with the type of data that you want to filter for. The following example filters for `ACTION_NDEF_DISCOVERED` intents with a MIME type of `text/plain`:

```
1 <intent-filter>
2     <action android:name="android.nfc.action.NDEF_DISCOVERED"/>
3     <category android:name="android.intent.category.DEFAULT"/>
4     <data android:mimeType="text/plain" />
5 </intent-filter>
```

The following example filters for a URI in the form of <https://developer.android.com/index.html>.

```
1 <intent-filter>
2     <action android:name="android.nfc.action.NDEF_DISCOVERED"/>
3     <category android:name="android.intent.category.DEFAULT"/>
4     <data android:scheme="https"
5         android:host="developer.android.com"
6         android:pathPrefix="/index.html" />
7 </intent-filter>
```

Obtain information from intents

If an activity starts because of an NFC intent, you can obtain information about the scanned NFC tag from the intent. Intents can contain the following extras depending on the tag that was scanned:

- `EXTRA_TAG` (required): A `Tag` object representing the scanned tag.

- EXTRA_NDEF_MESSAGES (optional): An array of NDEF messages parsed from the tag. This extra is mandatory on ACTION_NDEF_DISCOVERED intents.
- EXTRA_ID (optional): The low-level ID of the tag.

To obtain these extras, check to see if your activity was launched with one of the NFC intents to ensure that a tag was scanned, and then obtain the extras out of the intent. The following example checks for the ACTION_NDEF_DISCOVERED intent and gets the NDEF messages from an intent extra.

```

1  @Override
2  protected void onNewIntent(Intent intent) {
3      super.onNewIntent(intent);
4      ...
5      if (NfcAdapter.ACTION_NDEF_DISCOVERED.equals(intent.getAction())) {
6          Parcelable[] rawMessages =
7              intent.getParcelableArrayExtra(NfcAdapter.EXTRA_NDEF_MESSAGES);
8          if (rawMessages != null) {
9              NdefMessage[] messages = new NdefMessage[rawMessages.length];
10             for (int i = 0; i < rawMessages.length; i++) {
11                 messages[i] = (NdefMessage) rawMessages[i];
12             }
13             // Process the messages array.
14             ...
15         }
16     }
17 }

```

Alternatively, you can obtain a Tag object from the intent, which will contain the payload and allow you to enumerate the tag's technologies:

```

1  Tag tag = intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);

```

Android Application Records

Introduced in Android 4.0 (API level 14), an Android Application Record (AAR) provides a stronger certainty that your application is started when an NFC tag is scanned. An AAR has the package name of an application embedded inside an NDEF record. You can add an AAR to any NDEF record of your NDEF message because Android searches the entire NDEF message for AARs. If it finds an AAR, it starts the application based on the package name inside the AAR. If the application is not present on the device, Google Play is launched to download the application.

AARs are useful if you want to prevent other applications from filtering for the same intent and potentially handling specific tags that you have deployed. AARs are only supported at the application level, because of the package name constraint, and not at the Activity level as with intent filtering. If you want to handle an intent at the Activity level, use intent filters.

If a tag contains an AAR, the tag dispatch system dispatches in the following manner:

1. Try to start an Activity using an intent filter as normal. If the Activity that matches the intent also matches the AAR, start the Activity.
2. If the Activity that filters for the intent does not match the AAR, if multiple Activities can handle the intent, or if no Activity handles the intent, start the application specified by the AAR.
3. If no application can start with the AAR, go to Google Play to download the application based on the AAR.

Note: You can override AARs and the intent dispatch system with the foreground dispatch system, which allows a foreground activity to have priority when an NFC tag is discovered. With this method, the activity must be in the foreground to override AARs and the intent dispatch system.

If you still want to filter for scanned tags that do not contain an AAR, you can declare intent filters as normal. This is useful if your application is interested in other tags that do not contain an AAR. For example, maybe you want to guarantee that your application handles proprietary tags that you deploy as well as general tags deployed by third parties. Keep in mind that AARs are specific to Android 4.0 devices or later, so when deploying tags, you most likely want to use a combination of AARs and MIME types/URIs to support the widest range of devices. In addition, when you deploy NFC tags, think about how you want to write your NFC tags to enable support for the most devices (Android-powered and other devices). You can do this by defining a relatively unique MIME type or URI to make it easier for applications to distinguish.

Android provides a simple API to create an AAR, `createApplicationRecord()`. All you need to do is embed the AAR anywhere in your `NdefMessage`. You do not want to use the first record of your `NdefMessage`, unless the AAR is the only record in the `NdefMessage`. This is because the Android system checks the first record of an `NdefMessage` to determine the MIME type or URI of the tag, which is used to create an intent for applications to filter. The following code shows you how to create an AAR:

```
1 NdefMessage msg = new NdefMessage(  
2     new NdefRecord[] {  
3         ...,  
4         NdefRecord.createApplicationRecord("com.example.android.beam")}  
5     );  
6 )
```

Detailed Level Design

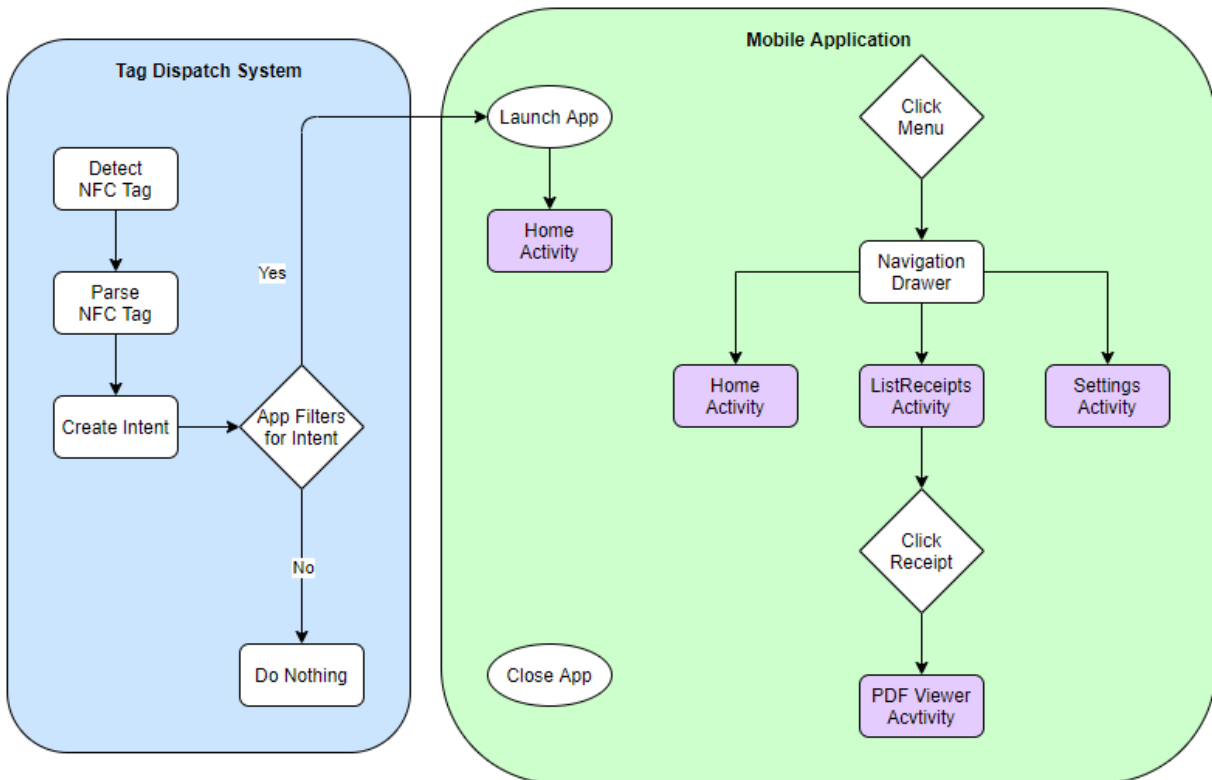


Figure 43: Detailed Level Mobile Design

Implementation

Home Activity Layout

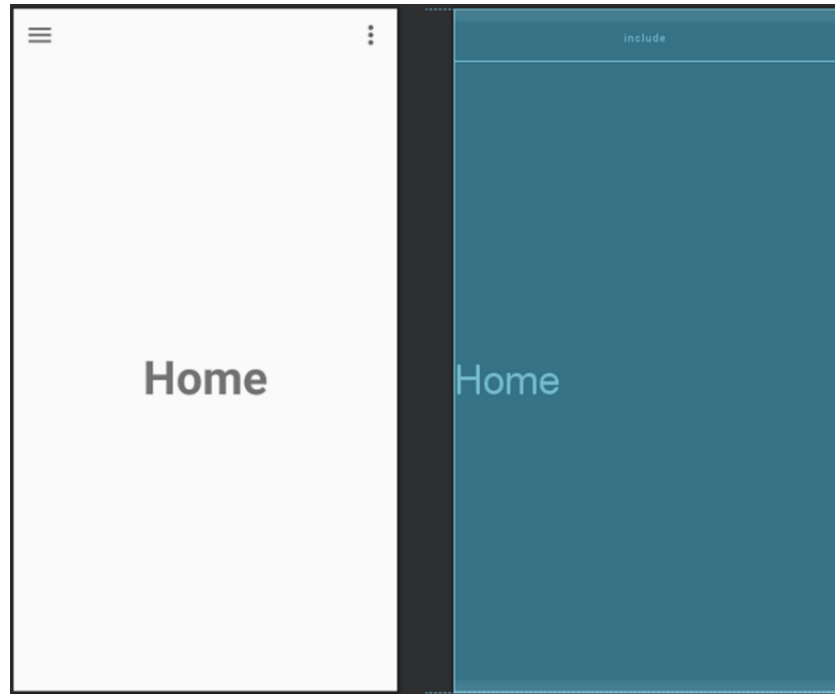


Figure 44: Home Activity Design

```

1 |<?xml version="1.0" encoding="utf-8"?>
2 |<androidx.drawerlayout.widget.DrawerLayout
3 |    xmlns:android="http://schemas.android.com/apk/res/android"
4 |    xmlns:tools="http://schemas.android.com/tools"
5 |    android:layout_width="match_parent"
6 |    android:layout_height="match_parent"
7 |    android:id="@+id/drawer_layout"
8 |    tools:context=".MainActivity">
9 |
10 |    <LinearLayout
11 |        android:layout_width="match_parent"
12 |        android:layout_height="match_parent"
13 |        android:orientation="vertical">
14 |
15 |        <include
16 |            layout="@layout/toolbar_layout"/>
17 |
18 |        <TextView
19 |            android:layout_width="match_parent"
20 |            android:layout_height="match_parent"
21 |            android:text="Home"
22 |            android:textSize="50sp"
23 |            android:textStyle="bold"
24 |            android:gravity="center"/>
25 |
26 |    </LinearLayout>
27 |
28 |    <RelativeLayout
29 |        android:layout_width="300dp"
30 |        android:layout_height="match_parent"
31 |        android:layout_gravity="start"
32 |        android:background="@android:color/white">
33 |
34 |        <include
35 |            layout="@layout/nav_drawer_layout"/>
36 |
37 |    </RelativeLayout>
38 |
39 |</androidx.drawerlayout.widget.DrawerLayout>

```

Figure 45: Home Activity Code

Navigation Drawer Layout

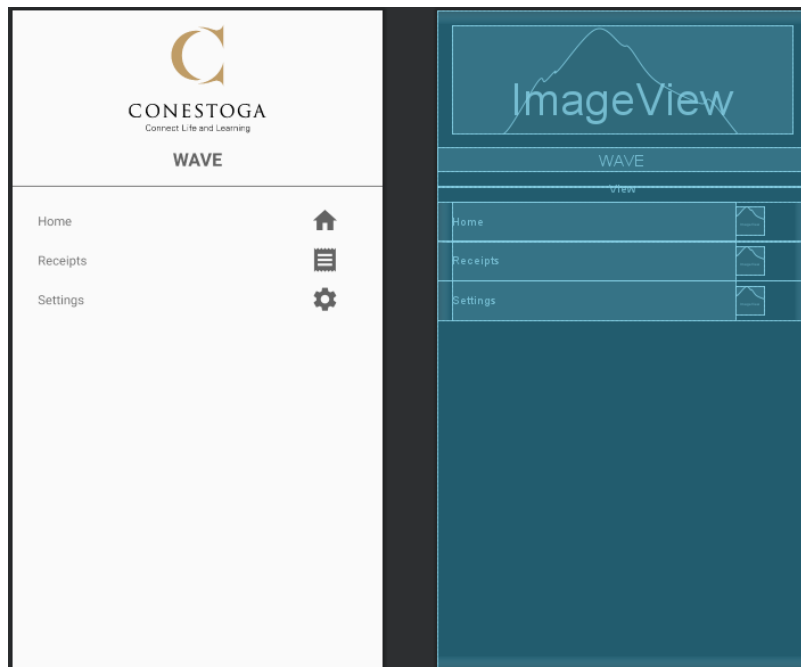


Figure 46: Navigation Drawer Design

```
31 <LinearLayout
32     android:layout_width="match_parent"
33     android:layout_height="wrap_content"
34     android:orientation="horizontal"
35     android:gravity="center_vertical"
36     android:onClick="ClickHome">
37
38     <TextView
39         android:layout_width="0dp"
40         android:layout_height="wrap_content"
41         android:layout_weight="1"
42         android:text="Home"
43         android:padding="12dp"
44         android:layout_marginStart="16dp"/>
45
46     <ImageView
47         android:layout_width="wrap_content"
48         android:layout_height="wrap_content"
49         android:layout_marginEnd="48dp"
50         android:src="@drawable/ic_home_32"/>
51
52 </LinearLayout>
```

Figure 47: Navigation Drawer Activity Code Snippet

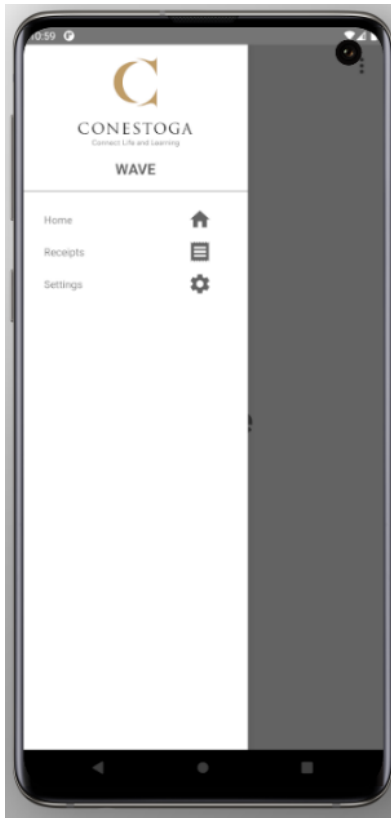


Figure 48: Navigation Drawer Layout Example

Table 8: Unit Tests for Navigation Drawer Functions

Unit Under Test	Success Condition	Test Procedure
clickMenu()	Calls openDrawer()	<ol style="list-style-type: none"> 1. Check the menu button can be pressed on every activity 2. Verify openDrawer() is called
openDrawer()	Opens nav. drawer	<ol style="list-style-type: none"> 1. Verify the navigation drawer opens on every activity
clickLogo()	Calls closeDrawer()	<ol style="list-style-type: none"> 1. Check the logo button can be pressed on every activity 2. Verify closeDrawer() is called
closeDrawer()	Closes nav. drawer	<ol style="list-style-type: none"> 1. Verify the navigation drawer closes on every activity
clickHome()	Navigates to home activity	<ol style="list-style-type: none"> 1. Check the home button can be pressed on every activity 2. Verify the app redirects to home activity from every activity
clickSettings()	Navigates to settings activity	<ol style="list-style-type: none"> 1. Check the settings button can be pressed on every activity 2. Verify the app redirects to settings activity from every activity
clickReceipts()	Navigates to list receipts activity	<ol style="list-style-type: none"> 1. Check the receipts button can be pressed on every activity

		2. Verify the app redirects to list receipts activity from every activity
redirectActivity()	Opens specified activity	1. Verify the app redirects to the specified activity

PDF Viewer Activity

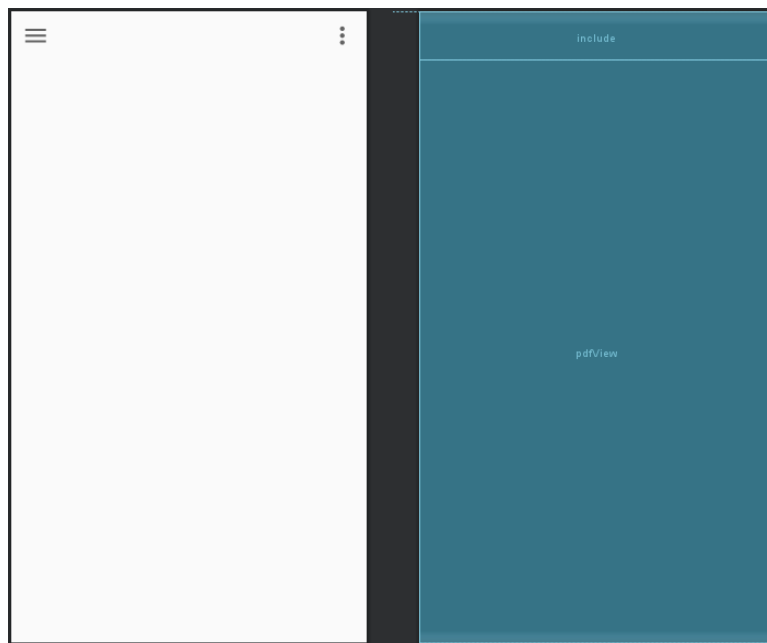


Figure 49: PDF Viewer Activity Design

```
43      implementation 'com.github.barteksc:android-pdf-viewer:2.8.2'
```

Figure 50: PDF Viewer Code Dependency

```
19      <com.github.barteksc.pdfviewer.PDFView
20          android:id="@+id/pdfView"
21          android:layout_width="match_parent"
22          android:layout_height="match_parent"
23          app:layout_constraintBottom_toBottomOf="parent"
24          app:layout_constraintEnd_toEndOf="parent"
25          app:layout_constraintHorizontal_bias="0.0"
26          app:layout_constraintStart_toStartOf="parent"
27          app:layout_constraintTop_toBottomOf="@+id/pdf_viewer_toolbar"
28          app:layout_constraintVertical_bias="0.0" />
```

Figure 51: PDF Viewer Activity Code Snippet

Toolbar Layout

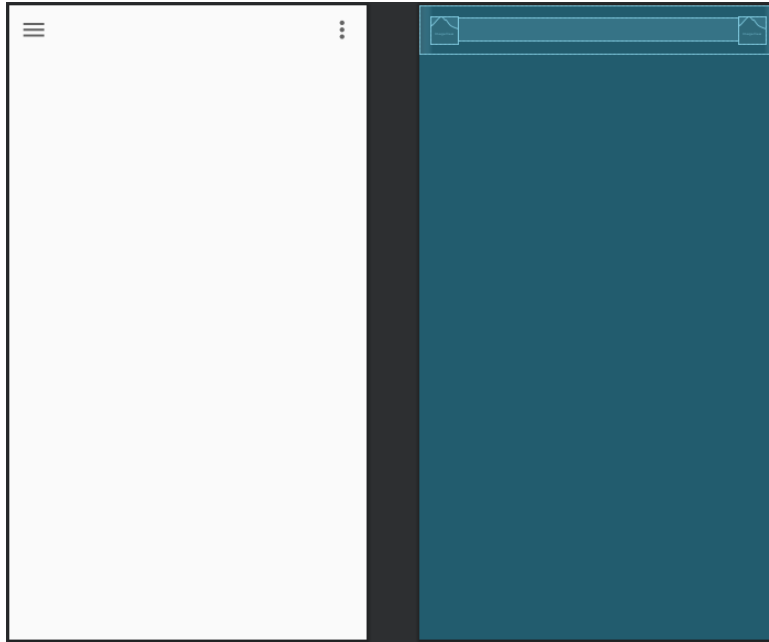


Figure 52: Toolbar Design

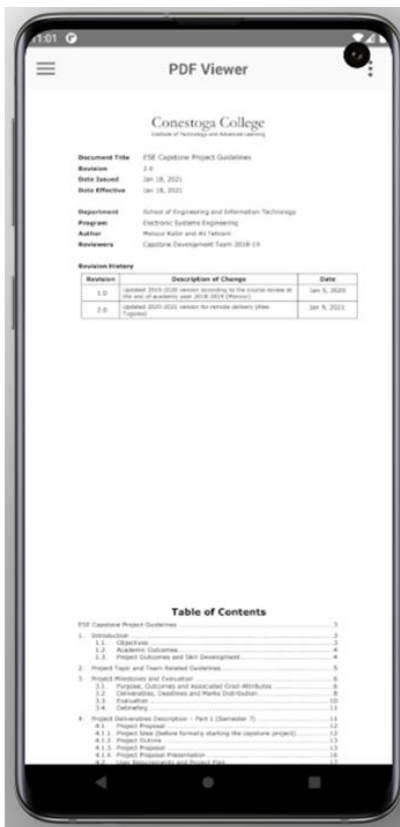
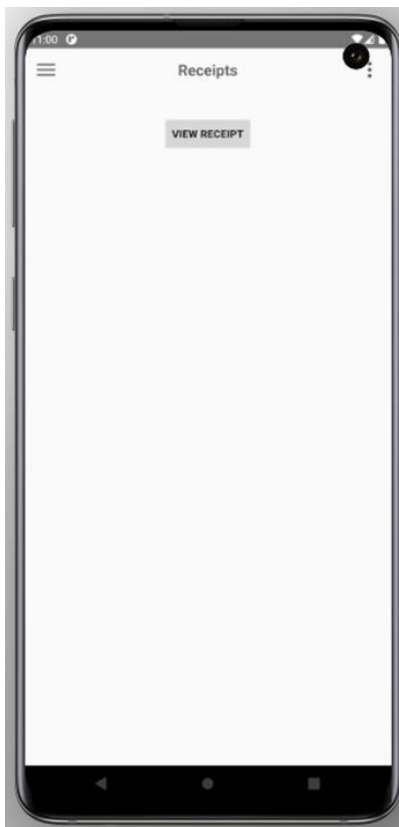
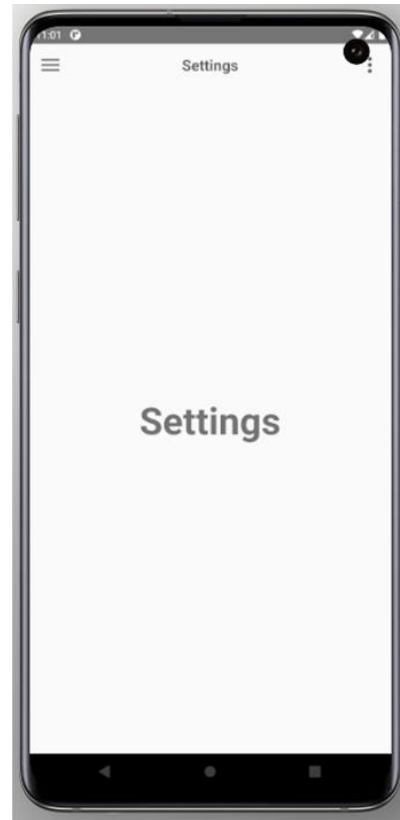
```

1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout
3      xmlns:android="http://schemas.android.com/apk/res/android"
4      android:layout_width="match_parent"
5      android:layout_height="wrap_content"
6      android:orientation="horizontal"
7      android:padding="12dp"
8      android:gravity="center_vertical">
9
10     <ImageView
11         android:layout_width="wrap_content"
12         android:layout_height="wrap_content"
13         android:onClick="ClickMenu"
14         android:src="@drawable/ic_menu_32"/>
15
16     <TextView
17         android:id="@+id/toolbar_title"
18         android:layout_width="0dp"
19         android:layout_height="wrap_content"
20         android:layout_weight="1"
21         android:textAlignment="center"
22         android:textColor="#636363"
23         android:textSize="20sp"
24         android:textStyle="bold" />
25
26     <ImageView
27         android:layout_width="wrap_content"
28         android:layout_height="wrap_content"
29         android:src="@drawable/ic_more_vert_32"/>
30 </LinearLayout>

```

Figure 53: Toolbar Code

Activity Layouts Example



Testing

The methodology used for testing the device will in 3 main stages.

1. Unit tests
2. System tests

The unit tests will be for testing the functionality of a single module, like the POS Switch Interface module. This group of tests are to ensure that the module is truly working as a black-box part of the system.

The final system test will be for testing the functionality of the entire system. It would consist of generating a receipt on the POS and sending the file all the way to the mobile application.

Unit Tests

POS Switch Interface

The following tables outline the overall software test plans that can be done to ensure the POS & Splitter functional units are operating correctly.

POS to Splitter Interface	
Unit Under Test	Get_Receipt()
Success Condition	The POS successfully sends a receipt to the Splitter device and is saved locally on the Splitter.
Test Procedure	<ol style="list-style-type: none">1. Ensure CUPS service is operating correctly.2. Ensure Windows POS environment is configured to print to the virtual printer setup with CUPS.3. Print to the virtual printer.4. Check destination of virtual printer on the Splitter for the printed document.

Splitter	
Unit Under Test	Check_Receipt()
Success Condition	Receipts sent to the Splitter from the POS are recognized and moved to the next stage & deleted from the original location.
Test Procedure	<ol style="list-style-type: none">1. Print a receipt to the Splitter or put a receipt in the expected location.2. Start module.3. Check print location for the receipt, it should be removed.4. Check destination for the receipt, it should be moved there.

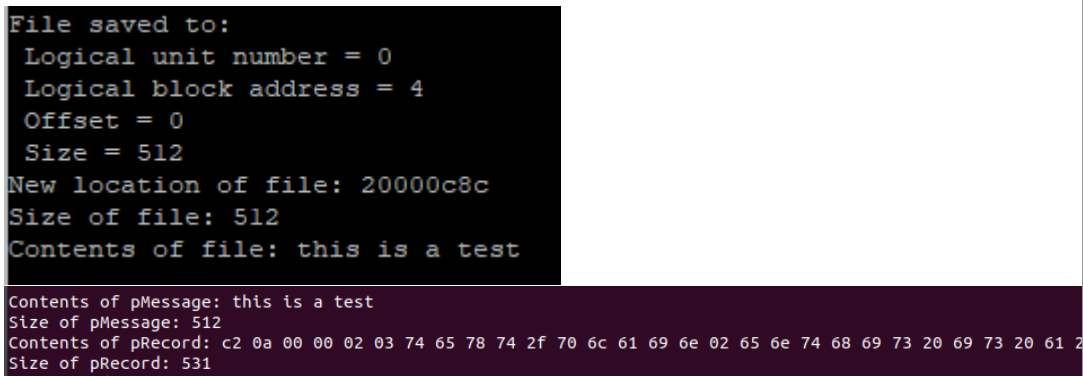
Splitter	
Unit Under Test	Check_GPIO()
Success Condition	The GPIO pin is read as a high or low depending on the switch circuit state.
Test Procedure	<ol style="list-style-type: none"> 1. Start module, print out GPIO state for testing. 2. Change GPIO input condition to be an expected high and check the output. 3. Change GPIO input condition to be an expected low and check the output.

Splitter	
Unit Under Test	Convert_To_PDF() or Convert_To_Printer()
Success Condition	The receipt is converted from either ESC/POS format to PDF format or PDF format to ESC/POS format.
Test Procedure	<ol style="list-style-type: none"> 1. Input either PDF or ESC/POS format receipt 2. Start module 3. Depending on the conversion: <ol style="list-style-type: none"> 1. If PDF → ESC/POS, save result to a text file & examine text format instead of binary data. 2. If ESC/POS → PDF, save output to a .PDF file & examine file to see if the receipt looks appropriate.

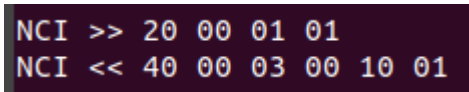
Splitter to NFC Interface	
Unit Under Test	Send_To_NFC()
Success Condition	NFC device receives a PDF receipt from the Splitter device.
Test Procedure	<ol style="list-style-type: none"> 1. Connect NFC device via USB. 2. Start module and input a PDF formatted receipt. 3. Check NFC Device storage for receipt

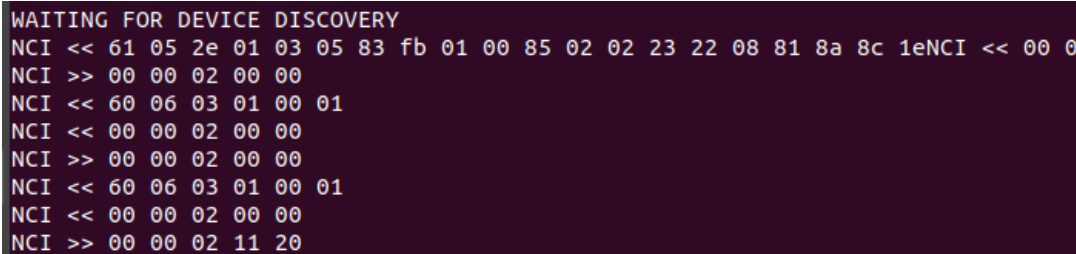
Splitter to Printer Interface	
Unit Under Test	Send_To_Printer()
Success Condition	Printer prints out the given receipt and is formatted correctly.
Test Procedure	<ol style="list-style-type: none"> 1. Connect Printer via USB. 2. Start module & input a receipt in ESC/POS format. 3. Check printer for output. 4. Check receipt for any errors (this can be done by opening the receipt and comparing against the original)

NFC Device

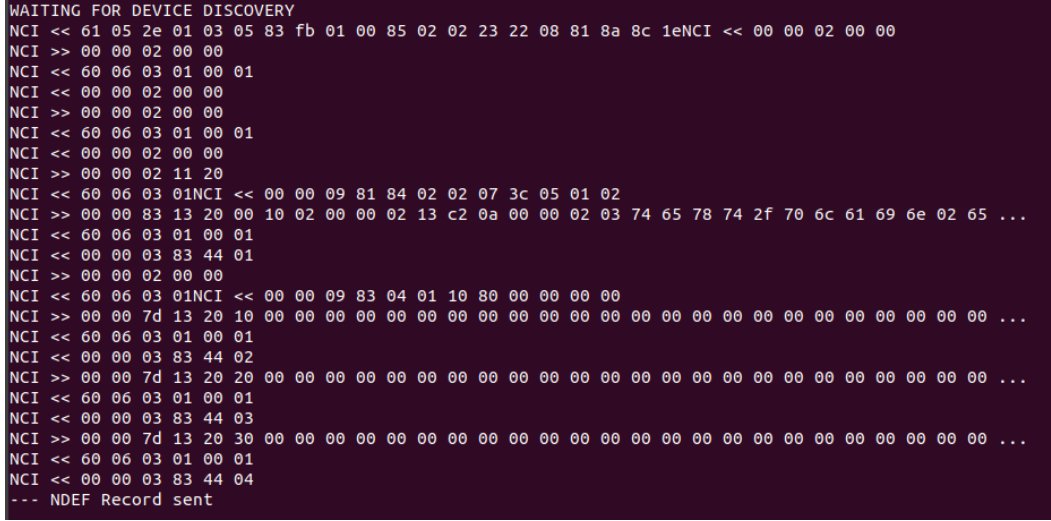
Functional Block	NFC to Mobile interface
Unit Under Test	NFC - Format data to NDEF
Function	format_to_NDEF()
Success Condition	Correct file location is passed to function.
Procedure	<ol style="list-style-type: none"> 1. Build and Flash debug version onto Pico 2. Add file to mass storage location on Host (Splitter) 3. Check serial debug log for memory location print <ol style="list-style-type: none"> 1. Confirm that memory location sent to this function is the same referred to in USB functional block
Result	<p>Success</p>  <pre> File saved to: Logical unit number = 0 Logical block address = 4 Offset = 0 Size = 512 New location of file: 20000c8c Size of file: 512 Contents of file: this is a test Contents of pMessage: this is a test Size of pMessage: 512 Contents of pRecord: c2 0a 00 00 02 03 74 65 78 74 2f 70 6c 61 69 6e 02 65 6e 74 68 69 73 20 69 73 20 61 2 Size of pRecord: 531 </pre>

Functional Block	NFC to Mobile interface
Unit Under Test	NFC – Configuration
Function	NxpNci_ConfigureSettings() ; NxpNci_ConfigureMode()
Success Condition	<p>Correct configuration is set</p> <p>*Note: Configuration is defined in Nfc_settings.h</p>
Procedure	<ol style="list-style-type: none"> 1. Build and Flash debug version onto Pico 2. Add file to mass storage location on Host (Splitter) 3. Check serial debug log for configuration settings <ol style="list-style-type: none"> 1. Confirm that the mode is P2P

Result	<p>Success</p>  <p>Indicates sending and receiving information from mobile device, therefore it's in P2P mode.</p>
---------------	---

Functional Block	NFC to Mobile interface
Unit Under Test	NFC – Mobile device detected
Function	NxpNci_StartDiscovery() NxpNci_WaitForDiscoveryNotification()
Success Condition	Mobile device is detected
Procedure	<ol style="list-style-type: none"> 1. Build and Flash debug version onto Pico 2. Add file to mass storage location on Host (Splitter) 3. Check serial debug log for successful detection <ol style="list-style-type: none"> 1. Confirm that the log shows mobile detection
Result	<p>Success</p>  <p>Indicates that a device has been discovered.</p>

Functional Block	NFC to Mobile interface
Unit Under Test	NFC – File transferred
Function	NxpNci_ProcessP2pMode()
Success Condition	File is successfully transferred
Procedure	<ol style="list-style-type: none"> 1. Build and Flash debug version onto Pico

	<ol style="list-style-type: none"> 2. Add file to mass storage location on Host (Splitter) 3. Check serial debug log for successful transfer <ol style="list-style-type: none"> 1. Confirm that the log shows the file has been transferred 4. Check mobile app (NFC Tools)* for successful file transmission
Result	<p>Success</p>  <p>“NDEF Record sent” indicates that the information was sent to the mobile device.</p> <p>This shows that the file was received on the mobile device:</p> 

--	--

Functional Block	LED Status Indicator
Unit Under Test	LED waiting for discovery
Function	led_waiting()
Success Condition	LED is fast blinking
Procedure	<ol style="list-style-type: none"> 1. Build and Flash debug version onto Pico 2. Add file to mass storage location on Host (Splitter) 3. Do not put mobile close to NFC 4. Check serial debug log for NFC status <ol style="list-style-type: none"> 1. Confirm it's waiting for mobile detection 5. Check LED <ol style="list-style-type: none"> 1. Confirm LED is fast blinking
Result	Not implemented

Functional Block	LED Status Indicator
Unit Under Test	LED transmitting
Function	led_transmitting()
Success Condition	LED is solid
Procedure	<ol style="list-style-type: none"> 1. Build and Flash debug version onto Pico 2. Add file to mass storage location on Host (Splitter) 3. Put mobile close to NFC device with mobile application running 4. Check serial debug log for NFC status <ol style="list-style-type: none"> 1. Confirm is transmitting 5. Check LED <ol style="list-style-type: none"> 1. Confirm LED is solid
Result	Not implemented

Functional Block	LED Status Indicator
Unit Under Test	LED completed transferring
Function	led_done()
Success Condition	LED does x3 slow blinks
Procedure	<ol style="list-style-type: none"> 1. Build and Flash debug version onto Pico 2. Add file to mass storage location on Host (Splitter) 3. Put mobile close to NFC device with mobile application running 4. Check serial debug log for NFC status <ol style="list-style-type: none"> 1. Confirm file has transferred 5. Check LED <ol style="list-style-type: none"> 1. Confirm LED does x3 slow blinks
Result	Not implemented

Functional Block	LED Status Indicator
Unit Under Test	LED waiting for discovery
Function	led_error()
Success Condition	LED does x2 fast blinks, pause, repeat
Procedure	<ol style="list-style-type: none"> 1. Build and Flash debug version onto Pico 2. Add file to mass storage location on Host (Splitter) 3. Wait 10s without putting mobile close to NFC device. 4. Check serial debug log for NFC status <ol style="list-style-type: none"> 1. Confirm there is an error 5. Check LED <ol style="list-style-type: none"> 1. Confirm LED does x2 fast blinks, pause, repeat
Result	not implemented

Mobile Application

Table 9: Unit Tests for Navigation Drawer Functions

Unit Under Test	Success Condition	Test Procedure
clickMenu()	Calls openDrawer()	<ol style="list-style-type: none"> 3. Check the menu button can be pressed on every activity 4. Verify openDrawer() is called
openDrawer()	Opens nav. drawer	<ol style="list-style-type: none"> 2. Verify the navigation drawer opens on every activity
clickLogo()	Calls closeDrawer()	<ol style="list-style-type: none"> 3. Check the logo button can be pressed on every activity 4. Verify closeDrawer() is called
closeDrawer()	Closes nav. drawer	<ol style="list-style-type: none"> 2. Verify the navigation drawer closes on every activity
clickHome()	Navigates to home activity	<ol style="list-style-type: none"> 3. Check the home button can be pressed on every activity 4. Verify the app redirects to home activity from every activity
clickSettings()	Navigates to settings activity	<ol style="list-style-type: none"> 3. Check the settings button can be pressed on every activity 4. Verify the app redirects to settings activity from every activity
clickReceipts()	Navigates to list receipts activity	<ol style="list-style-type: none"> 3. Check the receipts button can be pressed on every activity 4. Verify the app redirects to list receipts activity from every activity
redirectActivity()	Opens specified activity	<ol style="list-style-type: none"> 2. Verify the app redirects to the specified activity

All navigation drawer functionality specifications were met and working properly.

Table 10: Unit Tests for Functional Units

Unit Under Test	Success Condition	Test Procedure
Tag Dispatch System	NFC Tag is handled, an intent is created, and the mobile application is launched	<ol style="list-style-type: none"> 1. Unlock the testing phone 2. Put in range of NFC tag 3. Verify mobile application is started
Mobile Application	Complete functionality	<ol style="list-style-type: none"> 1. Ensure application can be opened and closed 2. Ensure UI displays properly and functions properly (i.e., navigation) 3. Ensure receipt is stored in specified local folder
List Receipts Activity	Displays a list of receipts stored in a specified local directory	<ol style="list-style-type: none"> 1. Ensure there are receipts stored in the specified local folder 2. Launch the mobile application 3. Navigate to the list receipts activity 4. Verify the list of receipts is displayed and the scrolling functionality works
PDF Viewer	Displays a PDF of the selected receipt	<ol style="list-style-type: none"> 1. Ensure there is a receipt stored in the specified local folder 2. Launch the mobile application 3. Navigate to the list receipts activity 4. Select the receipt to open 5. Ensure the application redirects to the PDF Viewer activity 6. Verify the scrolling functionality works
Navigation Drawer	Proper functionality	<ol style="list-style-type: none"> 1. Ensure the navigation drawer can be opened on every activity 2. Ensure the navigation drawer is displayed properly 3. Ensure every activity can navigate to all other activities listed in the navigation drawer
Toolbar	Proper functionality	<ol style="list-style-type: none"> 1. Ensure the toolbar appears on every activity 2. Ensure the current activity appears on the toolbar 3. Ensure the menu button on the toolbar opens the navigation drawer

All functionality specifications of each activity were met, except for the listReceiptsActivity. The intended functionality was to display a list of receipts in the WAVE application itself however we implemented instead a function that opens a 'file explorer' window that allows the user to navigate through their folders to select the PDF receipt of their choice. A default folder location is opened when this activity is launched.

Demo

A demo of the application can be viewed here – [WAVE Mobile Application Demo](#)

System Tests

This section covers the main test plans that will ensure the WAVE system is working correctly. Upon receiving an error, it is advised to use the more detailed test plans for the component that is failing to further diagnose the issue.

Overall Function of WAVEs Digital Receipt	
Unit Under Test	The digital receipt related systems.
Success Condition	<p>A receipt is received and opened on the mobile user's Android device.</p> <ul style="list-style-type: none"> - Receipt received in PDF format. - Total transfer is done within 3 seconds. - LED indicator blinks when processing data. - LED indicator solid when ready to transfer.
Test Procedure	<ol style="list-style-type: none"> 1. Start WAVE. 2. Ensure the toggle switch is in the leaning towards the "Digital" text. 2. Process an order on the POS system. 3. Check NFC for ready LED. 4. Once ready, bring the Android device towards the NFC terminal. 5. The WAVE application should launch, and the receipt should be transferred. 6. Open the newly transferred receipt and verify the contents.
Overall Function of WAVEs Paper Receipt	
Unit Under Test	The paper receipt related systems.
Success Condition	A receipt is printed in proper format on the thermal printer.
Test Procedure	<ol style="list-style-type: none"> 1. Start WAVE.

	2. Ensure the toggle switch is in the leaning towards the “Paper” text. 2. Process an order on the POS system. 6. The receipt should be printed out on the thermal printer. 7. Check the formatting for inconsistencies & errors.
Size Constraint of Devices	
Unit Under Test	Splitter & NFC Devices
Success Condition	Both the Splitter & NFC devices are enclosed in a maximum 15x15cm case. - 10cm maximum height for Splitter device. - 5cm maximum height for NFC device.
Test Procedure	1. Get the Splitter & NFC devices. 2. Measure their dimensions. 3. Compare against the requirements.

Conclusion

WAVE offers an innovative solution to the problems of paper receipts and emailed receipts in the current receipt industry. This system provides customers with the option of receiving their receipts direct to their mobile phones, minimizing the waste, resources, and pollution caused by paper receipts in a process that is more anonymous and private than emailed receipts. The implementation of the POS switch interface allows WAVE to easily be added to existing POS systems while maintaining the option of paper receipts for those customers who prefer it or do not have an NFC capable device. The NFC aspect of this system supplies inherent security of data transfer from NFC transmitter interface to the user’s mobile device. The mobile application allows effortless and intuitive management and viewing of digital receipts on the user’s device. This system accomplishes all this – simply with the WAVE of your phone.

Abbreviations

NFC	Near Field Communication
NFCC	Near Field Communication Controller
USB	Universal Serial Bus
TML	Transport Mapping Layer
NDEF	NFC Data Exchange Format
POS	Point of Sale
oPOS	Open-source Point of Sale
CUPS	Common Unix Printing System
LED	Light Emitting Diode
OS	Operating System
GPIO	General Purpose Input Output
CSV	Comma Separated Values
PDF	Portable Document Format
GND	Ground
ESC (ESC/POS)	Epson Standard Code for Printers

Appendices

Title	File
Getting started with Raspberry Pi Pico: C/C++ development with Raspberry Pi Pico and other RP2040-based microcontroller boards	getting-started-with-pico.pdf
Raspberry Pi Pico Datasheet: An RP2040-based microcontroller board	pico-datasheet.pdf

Raspberry Pi Pico C/C++ SDK: Libraries and tools for C/C++ development on RP2040 microcontrollers	rasberry-pi-pico-c-sdk.pdf
RP2040 Datasheet: A microcontroller by Raspberry Pi	rp2040-datasheet.pdf
Near Field Communication – Interface and Protocol (NFCPIP-1)	ECMA-240_3 rd _edition_june_2013.pdf
AN1190: NXP-NCI MCUXpresso example	AN11990.pdf
AN11687: PN71xx Linux Software Stack Integration Guidelines	PN71xx Linux Software Stack Integration Guidelines.pdf
AN11646: PN7120 NFC controller SBC kit quick start guide	PN7120 NFC controller SBC kit quick start guide.pdf
UM10878: PN7120 NFC controller SBC kit user manual	PN7120 NFC controller SBC kit user manual.pdf
PN7120: NFC controller with integrated firmware, supporting all NFC Forum modes	PN7120.pdf
UM10819: PN7120 User Manual	UM10819.pdf
Ethernet/IEEE 802.3	Ethernet-IEEE 802.3.pdf
NFC Essentials	NFC-Essentials-v2.0.1.pdf
NFC Logic Diagram	nfc-logic-diagram.jpg
TinyUSB Configuration File	tusb_config.h
USB Descriptors File	usb_descriptors.c
USB Mass Storage File	usb_msc.c
POS to Splitter example video	POS-Splitter-Example.mp4
Splitter check receipt example video	get-receipt-example.mp4
Switch circuit example video	switch-program-example.mp4
Thermal printer example video	Printer-Splitter-Example.mp4
Splitter to NFC device example video	nfc-splitter-example.mp4

Raspberry Pi 4 Computational Module	cm4-datasheet.pdf
Raspberry Pi 4 Product Brief	raspberrypi-4-product-brief.pdf
Raspberry Pi 4 Datasheet	rpi_DATA_2711_1p0.pdf
Raspberry Pi 4 Mechanical	rpi_MECH_4b_4p0.pdf
Raspberry Pi 4 Reduced Schematic	rpi_SCH_4b_4p0_reduced.pdf
Raspberry Pi 4 GPIO Pinout	rpi4-gpio-pinout.webp
Toggle Switch Datasheet	Switch-Datasheet-A101SDCQ04.pdf
Switch Circuit Schematic & PCB Layout	SwitchCircuit-Schematics.PDF.pdf

Table of Figures

Figure 1: WAVE Components.....	4
Figure 2: High level view of the WAVE System	6
Figure 3: Image of Raspberry Pi Pico []	9
Figure 4: Overview of POS & Splitter design.....	13
Figure 5: Switch circuit high state analysis	14
Figure 6: Switch circuit high state calculations	14
Figure 7: Switch circuit low state analysis	14
Figure 8: POS & Splitter software overview.....	15
Figure 9: Unicenta oPOS cashier screen	16
Figure 10: Unicenta oPOS sample receipt	17
Figure 11: GET_RECEIPT(); Example	18
Figure 12: Example getting the switch state.....	19
Figure 13: Example code which will print a pdf to the Thermal Printer	20
Figure 14: Example paper receipt	21
Figure 15: Example code to send the receipt to the NFC Transmitter Interface	22
Figure 16: NFC device - device design option 2 - two separate modules.....	24
Figure 17: High-level design.....	25
Figure 18: Diagram showing placement of interface board.	26
Figure 19: NFC Protocol stack	27
Figure 20: NDEF Message	27
Figure 21: NDEF Record layout	28
Figure 22: DEP frame format.	29
Figure 23: DEP transport data frame format.	29
Figure 24: General protocol flow for initialization [10].	30
Figure 25: Initiator needs to confirm there is no other RF field [10].....	31

Figure 26: Schematic pinout of OM5577 (PN7120) [11]. This design will use the TB5 header, as the TB4 header is basically a mirror of TB5 but used for a BeagleBoard interface.	36
Figure 27: Internal wiring to the PN7120 [11]. Relevant pins are circled.....	36
Figure 28: Physical pinout of OM5577 [11].	37
Figure 29: Actual view of the OM5577 (top view) [11].	37
Figure 30: Pinout of the Raspberry Pi Pico [2].	38
Figure 31: Initial sketch of the interface board. Note that pin 39 and pin 40 on the Raspberry Pi Pico can be shorted if not using a battery supply.	41
Figure 32: Schematic of the Interface Board between the Pico and the OM5577.....	42
Figure 33: Mechanical design of Pico [13].	43
Figure 34: Final PCB design of the Interface Board.....	44
Figure 35: Software stack for NFC controller interface.	47
Figure 36: Code snippet for nfc_task().....	47
Figure 37: Code snippet build setup using CMake for adding TinyUSB libraries.	54
Figure 38: Fat12 memory allocation table.....	56
Figure 39: Struct for holding fat12 root directory data entry.....	57
Figure 40: Final device showing Raspberry Pi Pico, OM5577 NFC board, and the interface board underneath.	57
Figure 41: High Level Mobile Design	58
Figure 42: Tag Dispatch System	64
Figure 43: Detailed Level Mobile Design	68
Figure 44: Home Activity Design	69
Figure 45: Home Activity Code	70
Figure 46: Navigation Drawer Design	71
Figure 47: Navigation Drawer Activity Code Snippet	71
Figure 48: Navigation Drawer Layout Example	72
Figure 49: PDF Viewer Activity Design	73
Figure 50: PDF Viewer Code Dependency	73
Figure 51: PDF Viewer Activity Code Snippet	73
Figure 52: Toolbar Design	74
Figure 53: Toolbar Code	74

References

- [1] M. Breyer, "The Surprising Impact of Paper Receipts," 11 November 2020. [Online]. Available: <https://www.treehugger.com/surprising-impact-paper-receipts-4858146>. [Accessed 09 February 2021].
- [2] M. Breyer, "Treehugger," 11 November 2020. [Online]. Available: <https://www.treehugger.com/surprising-impact-paper-receipts-4858146>. [Accessed 21 April 2021].
- [3] uniCenta, "uniCenta," [Online]. Available: <https://unicenta.com/>. [Accessed 21 April 2021].
- [4] CUPS.org, "CUPS," [Online]. Available: <https://www.cups.org/index.html>. [Accessed 21 April 2021].
- [5] Raspberry Pi, "Raspberry Pi 4," [Online]. Available: <https://www.raspberrypi.org/products/raspberry-pi-4-model-b/>. [Accessed 21 April 2021].
- [6] tatoko store, "Amazon," [Online]. Available: https://www.amazon.com/tatoko-Vibration-Button-Type-Vibrating-Appliances/dp/B07Q1ZV4MJ/ref=sr_1_2?dchild=1&keywords=cell+phone+motor&qid=1624408035&sr=8-2. [Accessed 4 August 2021].
- [7] Raspberry Pi, "cm4-datasheet.pdf," Raspberry Pi, 2021.
- [8] TE Connectivity, "Switch-Datasheet-A101SDCQ04.pdf," Tyco Electronics Corporation, 2013.
- [9] M. S. Blog, "What is ESC/POS, and how do I use it?," 12 November 2014. [Online]. Available: <https://mike42.me/blog/what-is-escpos-and-how-do-i-use-it>. [Accessed 21 April 2021].
- [10] TC51, "ECMA-340: Near Field Communication Interface and Protocol (NFCIP-1)," June 2013.
] [Online]. Available: <https://www.ecma-international.org/publications-and-standards/standards/ecma-340/>. [Accessed 13 August 2021].
- [11] NXP Semiconductors, "UM10878: PN7120 NFC controller SBC kit user manual," 9 February 2021.
] [Online]. Available: <https://www.nxp.com/docs/en/user-guide/UM10878.pdf>. [Accessed 1 March 2021].
- [12] NXP Semiconductors, "PN7120: NFC controller with integrated firmware, supporting all NFC Forum modes," 11 June 2018. [Online]. Available: <https://www.nxp.com/docs/en/data-sheet/PN7120.pdf>. [Accessed 1 March 2021].
- [13] Raspberry Pi (Trading) Ltd., "Raspberry Pi Pico Datasheet," 5 March 2021. [Online]. Available: <https://datasheets.raspberrypi.org/pico/pico-datasheet.pdf>. [Accessed 15 March 2021].
- [14] Raspberry Pi (Trading) Ltd., "Raspberry Pi Pico: Download design files," 2021.
]

- [15 Raspberry Pi (Trading) Ltd., "Raspberry Pi Pico C/C++ SDK: Libraries and tools for C/C++ development on RP2040 microcontrollers," [Online]. Available:
] <https://datasheets.raspberrypi.org/pico/raspberry-pi-pico-c-sdk.pdf>. [Accessed 1 March 2021].
- [16 NXP Semiconductors, "AN11990: NXP-NCI MCUXpresso example," [Online]. Available:
] <https://www.nxp.com/docs/en/application-note/AN11990.pdf>. [Accessed 1 March 2021].
- [17 H. Thach, "tinyusb," Github, 2021.
]
- [18 A. Developers, "Application Fundamentals," Android Developers, 23 February 2021. [Online].
] Available: <https://developer.android.com/guide/components/fundamentals>. [Accessed 22 April 2021].
- [19 A. Developers, "Layouts," Android Developers, 7 January 2020. [Online]. Available:
] <https://developer.android.com/guide/topics/ui/declaring-layout?hl=en>. [Accessed 22 April 2021].
- [20 A. Developers, "android.nfc," Android Developers, 24 February 2021. [Online]. Available:
] <https://developer.android.com/reference/android/nfc/package-summary>. [Accessed 22 April 2021].
- [21 A. Developers, "NFC Basics," Android Developers, 1 December 2020. [Online]. Available:
] <https://developer.android.com/guide/topics/connectivity/nfc/nfc>. [Accessed 22 April 2021].
- [22 Digikey, "MIKROE-2540," MikroElektronika, [Online]. Available:
] <https://www.digikey.ca/en/products/detail/mikroelektronika/MIKROE-2540/7394013>. [Accessed 12 August 2021].
- [23 J. Turcotte, Director, *switch-program-example.mp4*. [Film]. Canada: Justin Turcotte, 2021.
]
- [24 J. Turcotte, "SwitchCircuit-Schematics.PDF.pdf," Justin Turcotte, Kitchener, 2021.
]
- [25 J. Turcotte, Director, *Printer-Splitter-Example.mp4*. [Film]. Canada: Justin Turcotte, 2021.
]
- [26 J. Turcotte, Director, *POS-Splitter-Example.mp4*. [Film]. Canada: Justin Turcotte, 2021.
]
- [27 J. Turcotte, Director, *nfc-splitter-example.mp4*. [Film]. Canada: Justin Turcotte, 2021.
]
- [28 J. Turcotte, Director, *get-receipt-example.mp4*. [Film]. Canada: Justin Turcotte, 2021.
]

- [29 RaspberryTips, "Raspbian: How to add a printer on your Raspberry Pi? (CUPS)," [Online]. Available:
] <https://raspberrytips.com/install-printer-raspberry-pi/>. [Accessed 21 April 2021].
- [30 RF Wireless World, "NFC Tutorial," 2012. [Online]. Available: [https://www.rfwireless-](https://www.rfwireless-world.com/Tutorials/NFC-Near-Field-Communication-tutorial.html)
] [world.com/Tutorials/NFC-Near-Field-Communication-tutorial.html](https://www.rfwireless-world.com/Tutorials/NFC-Near-Field-Communication-tutorial.html). [Accessed 09 February 2021].
- [31 RF Wireless World, "NFC Security," 2012. [Online]. Available: [https://www.rfwireless-](https://www.rfwireless-world.com/Tutorials/NFC-Near-Field-Communication-security.html)
] [world.com/Tutorials/NFC-Near-Field-Communication-security.html](https://www.rfwireless-world.com/Tutorials/NFC-Near-Field-Communication-security.html). [Accessed 09 February 2021].