PN532 NFC Reader Guide - C++

Jan Mazurek, Maheeppartap Singh, Ryan Wittkopf
CMPT 433, Spring 2024
Department of Computer Science
Simon Fraser University
Burnaby, BC
{ jma185, msa206, rwa116 } @sfu.ca

I. Introduction

This guide is here to help you integrate the PN532 Breakout shield module from Arduino into your C++ project. Although all of the code is written in C++, the protocols that are defined here can be implemented in any programming language.

The PN532 Breakout shield operates in 2 modes: I2C and SPI. For this guide, we will use I2C, because it's easier to work with in C++, and requires less wiring. Note that this guide was written to work with Zen Cape Red.

II. Wiring it up

The PN532 needs power, and can work with 5.0V and 3.3V. For this, we will use the 3.3V. Make sure the BBG is powered down before attempting this.

Steps:

- Configure the breaker on the NFC module, to make it ready for I2C. Refer to image 1 for this. Note that I did not take this picture, and was taken from the internet.
- Connect the 3.3V pin on the module to your Beaglebone's 3.3V power delivery (on zen cape red, that would be P9-3 or P9-4).
- Then connect the ground to your zen cape's ground pin (On zen cape red, that would be P9-1)
- Because we are using I2C, we only need to connect the MOSI/SDA/TX and SSEL/SCL/RX pins. Connect them to a I2C configurable pin on your device. For this tutorial, we connect the RX to P9-19, and TX to P9-20

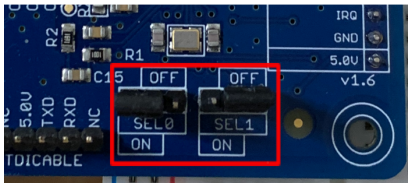

Fig. 1. I2C Breaker configuration

This should be enough to get the device configured and running. Don't forget to run config-pin on the RX and TX pins (P9-19 and P9-20 in our case) and configure them to I2C

```
(bbg) $ config-pin p9_19 i2c
(bbg) $ config-pin p9_20 i2c
```

A. Debugging

Note that any of the "NFC" modules that are developed and supported by the community can't find this device. So if you find any command-line tools, they won't work. Our theory is that the drivers cannot recognize the Beaglebone green properly, and that breaks things. To quickly test if the device is up and running, here is a small script we used to test the configurations:

```python
import board
import busio
from digitalio import DigitalInOut
import adafruit_pn532.i2c as PN532_I2C

# Create I2C bus.
i2c = busio.I2C(board.SCL, board.SDA)

# Create a PN532 I2C object.
pn532 = PN532_I2C.PN532_I2C(i2c, debug=False)

# Configure PN532 to communicate with MiFare cards.
pn532.SAM_configuration()

print("Waiting for RFID/NFC card...")
while True:
    # Check if a card is available to read.
    uid = pn532.read_passive_target(timeout=0.5)
    # Try again if no card is available.
    if uid is None:
        continue
    print("Found card with UID:", [hex(i) for i in uid])
    break
```

Listing 1. Python Test script for PN532

When running this script, tap a NFC enabled card, and the code should print the UID of the card. If the script does not work, and says that it cannot recognize the board, that means the drivers for the PN532 are still not fixed and cannot recognize the BeagleBone. In that case, run the following command to forcefully set the BLINK_FORCEBOARD variable

```
export BLINKA_FORCEBOARD="BEAGLEBONE_GREEN"
```

This will export the variable, and now re-running the script should fix your problem.

B. How to use PN532 in C++

The PN532 supports multiple states, but for now, we will focus on just reading the data from a card. If you're interested knowing more about what modes are supported, and how the communication protocol works, refer to the official documentation. They explain it very well.

At the start of the program, we want to get I2C initialized, and tell the PN532 that we are ready to listen. Here is a simple initI2C function:

```cpp
/**
 * @brief Initializes the I2C bus for the NFC reader.
 *
 * This function opens the I2C bus and sets the slave address for communication with the NFC reader.
 *
 * @return The file descriptor of the opened I2C bus, or -1 if an error occurred.
 */
int NFCReader::initI2C()
{
    int file;
    if ((file = open(device, O_RDWR)) < 0)
    {
        std::cerr << "Failed to open the I2C bus" << std::endl;
        return -1;
    }
    if (ioctl(file, I2C_SLAVE, address) < 0)
    {
        std::cerr << "Failed to acquire bus access and/or talk to slave." << std::endl;
        close(file);
        return -1;
    }
    return file;
}
```

Listing 2. Init I2C function for PN532

The PN532 is a I2C Slave, that is configured at address 0x48 and can support a max clock frequency of 400KHz. You can send 2 kinds of commands to the PN532: Normal Information Frame and Extended information Frame. For a detailed look at the packet structure, refer to image 2.

Next, you must send the setup command to the PN532, and get it reading.

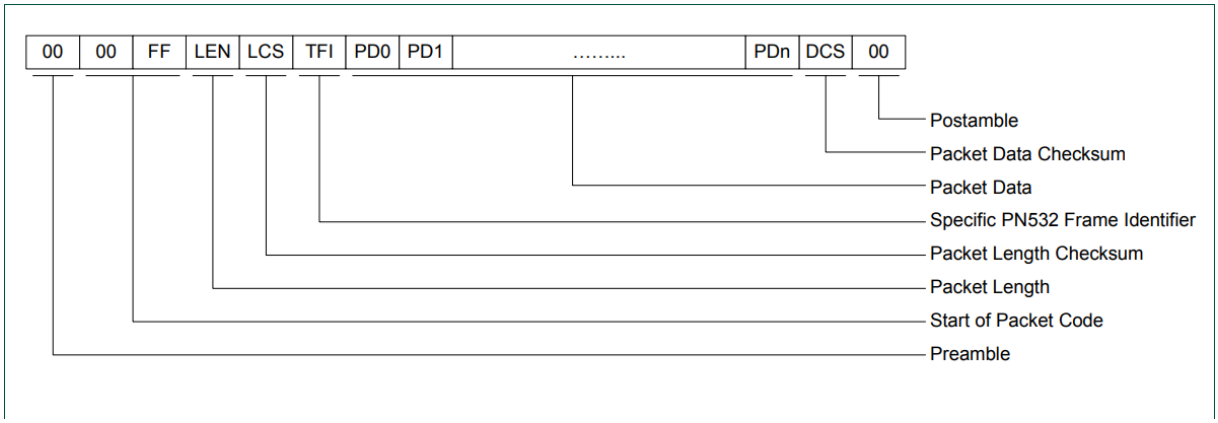

Fig. 2. PN532 Packet configuration

```cpp
/**
 * @brief Constructs an NFCReader object.
 *
 * This constructor initializes the NFCReader object with the specified device and address.
 * It also initializes the file descriptor and sends a setup command to the NFC reader.
 *
 * @param device The device name or path.
 * @param address The address of the NFC reader.
 */
NFCReader::NFCReader(const char *device, int address, ShutdownManager *shutdownManager)
    : device(device), fileDescriptor(-1), address(address), shutdownManager(shutdownManager)
{
    // config pin
    system("config-pin P9_19 i2c");
    system("config-pin P9_20 i2c");


    fileDescriptor = initI2C();

    unsigned char response[256];
    unsigned char setupCommand[] = {
        0x00, // Reserved byte
        0x00, // Reserved byte
        0xFF, // Start of packet
        0x03, // Packet length
        0xFD, // Length checksum (0x100 - 0x03 = 0xFD)
        0xD4, // Data Exchange command
        0x14, // SAMConfiguration command
        0x01, //
        0x17, // Checksum for the command bytes (0x100 - 0xD4 - 0x14 - 0x01 = 0x17)
        0x00, // Reserved byte
    };

    if (sendCommandAndWaitForResponse(setupCommand, sizeof(setupCommand), response, sizeof(response),
    false))
    {
        std::cout << "NFC Reader setup complete" << std::endl;
    } else {
        std::cerr << "Failed to setup the NFC reader" << std::endl;
    }
}
```

Listing 3. Init I2C function for PN532

By default, every command we send to the PN532 will be ACKed by the device, and will have to be consumed. The ACK frame looks like 0x00 0x00 0xFF 0x00 0xFF 0x00. The fourth and fifth bytes are the ACK packet code. In our code, after sending a command, we wait 0.25ms for an ACK, then consume it. After an ACK has been verified, you can now send a read command, which will make the NFC reader wait for a card to be tapped. This is non-blocking. Here is the command for reading:

```
    unsigned char detectCardCommand[] = {
        0x00, // Reserved byte
        0x00, // Reserved byte
        0xFF, // Start of packet
        0x04, // Packet length
        0xFC, // Length checksum (0x100 - 0x04 = 0xFC)
        0xD4, // Data Exchange command
        0x4A, // InListPassiveTarget command
        0x01, // Max number of targets
        0x00, // Baud rate
        0xE1, // Checksum for the command bytes (0x200 - 0xD4 - 0x4A - 0x01 - 0x00 = 0xE1)
        0x00, // Reserved byte
    };
```

Listing 4. Init I2C function for PN532

The last function to implement would be the sendCommandAndWaitForResponse function. This function is used above in Listing 3, and contains the check for ACK, along with the timings of reading from the buffer. After sending a command to the NFC reader, we must wait 0.25ms for an ACK, verify the ACK, and then wait a few more seconds for the device to get ready.

Here is the implementation of the sendCommandAndWaitForResponse function, along with it's signature:

```
/**
 * Sends a command to the NFC reader and waits for a response.
 *
 * @param command The command to send.
 * @param commandLength The length of the command.
 * @param response The buffer to store the response.
 * @param responseLength The length of the response buffer.
 * @param need_response Flag indicating whether a response is needed.
 * @return True if the command was sent successfully and a response was received, false otherwise.
 */
bool NFCReader::sendCommandAndWaitForResponse(unsigned char *command, int commandLength, unsigned char *
    response, int responseLength, bool need_response)
{
    if (write(fileDescriptor, command, commandLength) != commandLength)
    {
        std::cerr << "Failed to write to the I2C bus." << std::endl;
        return false;
    }

    usleep(250000);

    // Now check for ACK
    unsigned char ackBuffer[7];
    if (read(fileDescriptor, ackBuffer, sizeof(ackBuffer)) != sizeof(ackBuffer) ||
        ackBuffer[1] != 0x00 || ackBuffer[2] != 0x00 || ackBuffer[3] != 0xFF ||
        ackBuffer[4] != 0x00 || ackBuffer[5] != 0xFF || ackBuffer[6] != 0x00)
    {
        std::cerr << "ACK frame not received." << std::endl;
        // print the received data
        for (unsigned int i = 0; i < sizeof(ackBuffer); i++)
        {
            std::cout << "0x" << std::hex << (int)ackBuffer[i] << " ";
        }
        return false;
    }

    sleep(1);   // give her some time

    if (!need_response)
    {
        return true;
    }

    int bytesRead;
    int i = 0;
    do
    {
        usleep(10000); // Wait for 10 milliseconds before retrying
        bytesRead = read(fileDescriptor, response, responseLength);
        i++;

        if (bytesRead < 0)
        {
```

```
            std::cerr << "Failed to read from the device." << std::endl;
            return false;
        }

    } while ((bytesRead == 0 || (int)response[0] == 0 ) && !shutdownManager->isShutdown());

    return true;
}
```

Listing 5. Init I2C function for PN532

Send this command to the I2C, wait for an ACK, and then attempt to read the file descriptor file for a return value! That's it, you now have a working NFC reader.