PN532 NFC Breakout Board Guide

Last update: April 11, 2022

This document guides the user through:

1. Wiring the PN532 breakout board to the beaglebone
2. Rust program read card/device UID using the PN532 breakout board

Table of Contents

1. Introduction

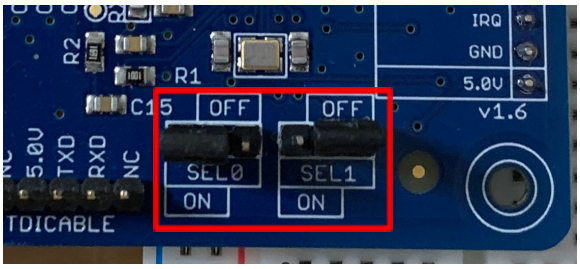
This document will guide you on how to set up the NFC scanner with the Beaglebone Green via I2C. NFC (near field communication) is a wireless data transfer method that enables two devices in close proximity to communicate. Through this guide, you will be able to use the NFC breakout board to read the UID of specific cards including compass cards, credit cards as well as Android devices.

2. Wiring

We will be using the NFC/RFID controller breakout board (using the PN532 NFC chip) which can be found here: https://www.adafruit.com/product/364.

2.1 I2C

We will be using I2C to communicate with the PN532 chip. To select I2C as the interface, insert the jumper into the ON position for SEL0 and the OFF position for SEL1 as shown below.

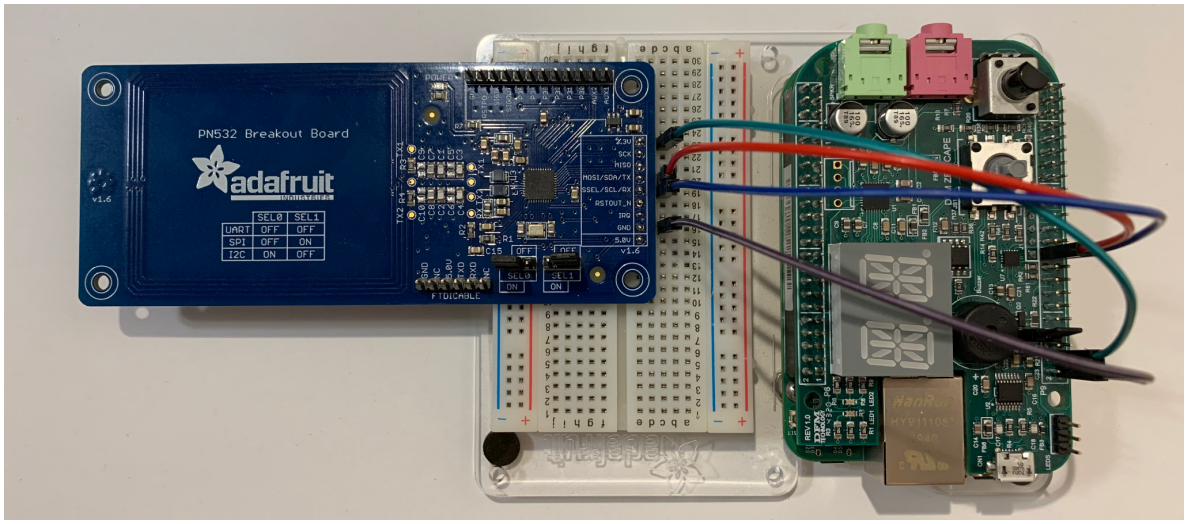


To connect the NFC breakout board to the beaglebone we will use the I2C2 bus with the SDA and SCL pin located at P9_20 and P9_19 respectively.

2.2 Wiring Steps

1. Connect the breakout board **SDA** to BBG **P9_20**
2. Connect the breakout board **SCL** to BBG **P9_19**
3. Connect the breakout board **3.3V** to BBG **3.3V** (either P9_3 or P9_4)
4. Connect the breakout board **GND** to BBG **GDND** (either P9_1 or P9_2)

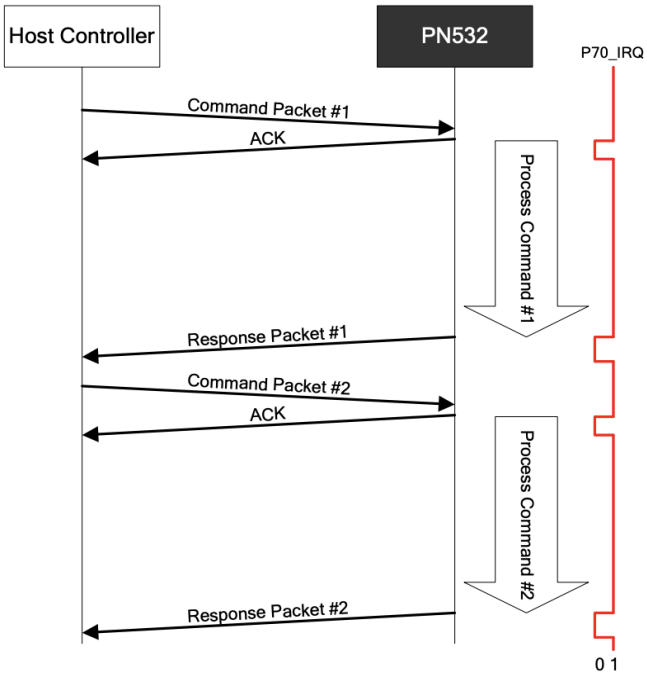
(Image on the next page)



SDA (red wire), SCL (blue wire), 3.3V (green wire), GND (purple wire)

3. PN532

This section reviews some of the command codes and specific frame structures the PN532 requires for input. Upon receiving the command, PN532 will respond with an ACK message and response packet as shown in the diagram below (source: user manual). More detailed information can be found in the user manual.



3.1 PN532 Commands and Frame Structures

We will be sending the following commands:

1. **SAMConfiguration** with command code 0x14
    - Command used for selecting the path of data flow.
    - The mode will have to be specified (we will use 0x01 Normal Mode)

   Data Packet: [command code, mode]

2. **InListPassiveTarget** with command code 0x4A
    - Command to detect as many cards/devices as specified by host.
    - The maximum number of targets (card/devices) to read at once and the card type must be specified.

   Data Packet: [command code, max # of targets, card type]

We will be receiving the following frames:

1. **Information Frame**
    - This frame is used both by the host (BBG) and target (PN532) to send and respond to commands

   Frame structure: [0x00, 0x00, 0xFF, LEN, LCS, TFI, **Data Packet**, DCS, 0x00]
    - LEN: length of data packet
    - LCS: Satisfies [LEN + LCS] = 0x00
    - TFI: 0xD4 (from BBG to PN532), 0xD5 (from PN532 to BBG)
    - DCS: Satisfies [TFI + PD0 + PD1 + … + PDn + DCS] = 0x00

2. **ACK Frame**
    - This frame is used to notify the host (BBG) that the previous packet was received.

   Frame structure: [0x00, 0x00, 0xFF, 0x00, 0xFF, 0x00]

3. **Error Frame**
    - This frame is used to notify the host (BBG) that an application-level error was detected.

   Frame structure: [0x00, 0x00, 0xFF, 0x01, 0xFF, …]

4. Getting UID from Cards/Devices using PN532 via Rust

4.1 Initialization

The following code configures the i2c pins (enable_bus()) and initializes the NFC breakout board, sending the SAMConfiguration command to PN532.

```
use i2cdev::core::I2CDevice;
use i2cdev::linux::LinuxI2CDevice;
use std::io::Result as IOResult;
use std::process::{Command, Output};
use std::thread::sleep;
use std::time::Duration;

pub struct NFCdev {
       i2cdev: LinuxI2CDevice,
}

enum Commands {
    SAMConfiguration            = 0x14,
    InListPassiveTarget         = 0x4A,
}

fn enable_bus() -> IOResult<Output> {
    Command::new("config-pin")
        .arg("P9_19")
        .arg("i2c")
        .output()?;
    Command::new("config-pin").arg("P9_20").arg("i2c").output()
}

impl NFCdev {
    pub fn new() -> Self {
        match enable_bus() {
            _ => (),
        }
        let i2cdev = LinuxI2CDevice::new("/dev/i2c-2",
           PN532_ADDRESS.into()).unwrap();
        let mut nfc = Self {i2cdev: i2cdev};
        match nfc.init_nfc() {
            _ => (),
        }
        nfc
```

```
    }

    pub fn init_nfc(&mut self) -> IOResult<()> {
        // function to send information frame to PN532
        self.send_command_to_nfcdev(&[Commands::SAMConfiguration as u8, 0x01])?;
        // expected to receive either ack or error frame
        self.sync_packets()
    }
}
```

4.2 Sending Data to PN532

The following function for NFCdev allows the program to send information frames to PN532.

```
fn send_command_to_nfcdev(&mut self, data: &[u8]) -> IOResult<()> {
        let length = data.len() as u8;
        let checksum_lcs = !length as u8;
        let tfi = 0xd4;
        let mut checksum_dcs = tfi;
        for i in data {
            checksum_dcs = checksum_dcs + i;
        }

        checksum_dcs = !(checksum_dcs & 0xFF) as u8 + 1;
        let mut frame = vec![0x00, 0x00, 0xff, length + 1, checksum_lcs, tfi];

        // place data packet into the information frame
        frame.extend_from_slice(data);
        frame.push(checksum_dcs);
        frame.push(0x00);

        // send frame to PN532
        match self.i2cdev.write(&frame) {
            _ => Ok(()),
        }
    }
```

4.3 Receiving Data from PN532

The following function for NFCdev allows the program to receive information and error frames from PN532.

```
fn receive_from_nfcdev(&mut self) -> IOResult<Vec<u8>> {
    for _ in 0..10 {
        sleep(Duration::from_millis(4));
        let mut data = [0u8; 128];

        // receive data
        self.i2cdev.read(&mut data)?;
        let mut index = 0;

        // Check if the received data is ack, error, or information frame
        for j in 0..data.len() {
            match index {
                0 | 1 => {
                    if data[j] == 0x00 {index += 1;continue;}
                    else {index = 0;}
                }
                2 => {
                    if data[j] == 0xFF {index += 1;continue;}
                    else {
                        return Err(std::io::Error::new(
                            std::io::ErrorKind::Other,
                            "Ack Response Err2"
                        ).into());
                    }
                }
                3 => match data[j] {
                    0x01 => {
                        return Err(std::io::Error::new(
                            std::io::ErrorKind::Other,
                            "App Error 2"
                        ).into());
                    }
                    size => {return Ok(
                          data[j + 3..j + 3 + (size as usize - 1)].to_vec());
                    }
                },
                _ => {index = 0;}
            }
        }
    }
    Err(std::io::Error::new(std::io::ErrorKind::Other, "timeout").into())
}
```

4.4 Receiving Synchronization Packets from PN532

The following function for NFCdev allows the program to receive ack and error frames to verify if command has been sent properly.

```
// function to receive one of the synchronization packets (ack or error)
fn sync_packets(&mut self) -> IOResult<()> {
    sleep(Duration::from_millis(1));
    for _ in 0..5 {
        let mut data = [0u8; 128];
        self.i2cdev.read(&mut data)?;
        let mut index = 0;

        // verify if received packet is ack or error
        for j in 0..data.len() {
            match index {
                0 | 1 => {
                    if data[j] == 0x00 {index += 1;continue;}
                    else {index = 0;}
                }
                2 => {
                    if data[j] == 0xFF {index += 1;continue;}
                    else {
                        return Err(std::io::Error::new(
                            std::io::ErrorKind::Other,"Ack Response Err1"
                        ).into());
                    }
                }
                3 => {
                    if data[j] == 0x00 {
                        return Ok(());
                    } else if data[j] as u8 == 0x01 {
                        return Err(std::io::Error::new(
                            std::io::ErrorKind::Other,"App Error 1"
                        ).into());
                    }
                }
                _ => {index = 0;}
            }
        }
    }

    Err(std::io::Error::new(std::io::ErrorKind::Other, "timeout").into())
}
```

4.5 Getting UID

The following function for NFCdev finds the UID of cards/devices. receive_from_nfcdev() returns the data packet portion of the information frame.

The data packet output for the InListPassiveTarget has the structure:
[0x4b, # of targets, …(4 bytes), UID Length, UID]

```
pub fn get_uid(&mut self) -> IOResult<Vec<Vec<u8>>> {
    self.send_command_to_nfcdev(&[
        Commands::InListPassiveTarget as u8,
        0x01, // maximum number of targets
        CardTypes::IsoTypeA as u8,
    ])?;

    self.sync_packets()?; // wait to receive ACK
    let reply = self.receive_from_nfcdev()?;
    let reply_length = reply.len();
    let mut index = 6;
    let mut id = Vec::new();
    let id_length = reply[index] as usize;
    index += 1;
    if index + id_length > reply_length {
        return Ok(Vec::new());
    }
    id.push(reply[index..index + id_length].to_vec());
    Ok(id)
}
```

4.6 Main Program

The following code prints the UID of cards/devices read by the NFC breakout board assuming the above code was placed in nfc.rs.

```
mod nfc;
pub fn main() {
    let mut nfc = nfc::NFCdev::new();
    println!("UID = {:x?}", nfc.get_uid().unwrap());
}
```

Output Example: `UID = [[a2, f6, 8, a9, 3, f2, 51, 2a]]`

4.7 Dependencies

Add the `i2cdev = "0.4.2"` under dependencies in the Cargo.toml file.

5. Troubleshooting

1. The SCL and SDA pins must be set to i2c mode. Check if they are configured for i2c:

   ```
   (bbg)$ config-pin -q P9_19
   (bbg)$ config-pin -q P9_20
   ```

   This should produce the following output:

   ```
   debian@beaglebone:/mnt/remote/myApps$ config-pin -q P9_20
   P9_20 Mode: i2c
   debian@beaglebone:/mnt/remote/myApps$ config-pin -q P9_19
   P9_19 Mode: i2c
   ```

   In case the program was unable to configure the pins, set the pins to i2c manually:

   ```
   (bbg)$ config-pin P9_19
   (bbg)$ config-pin P9_20
   ```

   Check if the NFC breakout board is properly connected:

   ```
   (bbg)$ i2cdetect -y -r 2
   ```

   You should see the device at address 0x24.

   ```
   debian@beaglebone:/mnt/remote/myApps$ i2cdetect -y -r 2
        0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
   00:          -- -- -- -- -- -- -- -- -- -- -- -- --
   10: -- -- -- -- -- -- -- -- 18 -- -- -- -- -- -- --
   20: -- -- -- -- 24 -- -- -- -- -- -- -- -- -- -- --
   30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
   40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
   50: -- -- -- -- UU UU UU UU -- -- -- -- -- -- -- --
   60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
   70: -- -- -- -- -- -- -- --
   ```

2. Make sure the NFC breakout board is wired to the BBG correctly.
   - 3.3V to BBG 3.3V
   - GND to BBG GDND
   - SDA to BBG P9_20
   - SCL to BBG P9_19