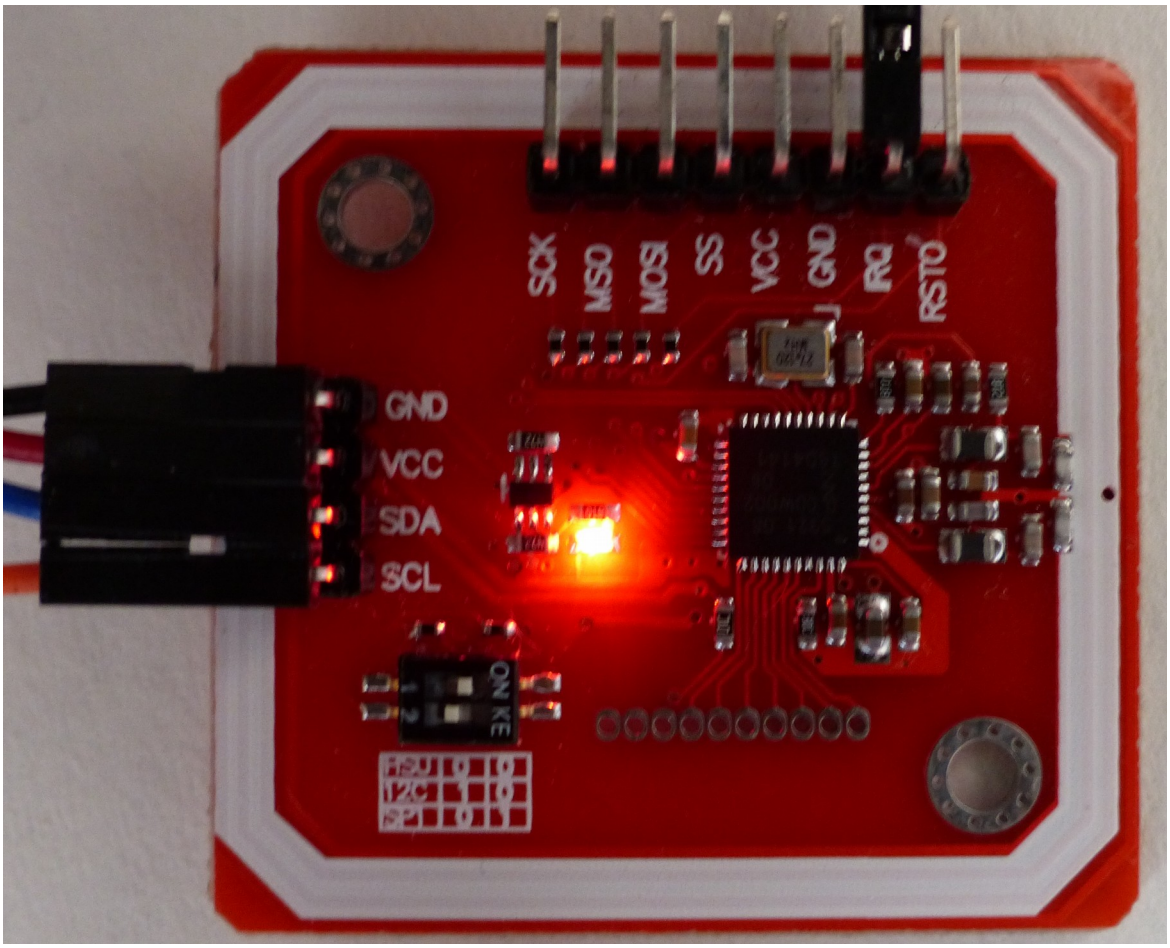
Test plan pn532 library



Created by : Nathan Houwaart
License : Boost licence
Project : IPASS 2018 -2019
Data : 02 – 07 – 2019

Introduction

NOTE: This test plan is written with the expectation that the reader has some general knowledge of way the pn532 communicates with a host controller. Information about the different frames can be found in the user manual (p.28 – 39 – section 6.2).

Throughout this test plan, references will be made to the user manual of the pn532. The user manual can be fount here: https://www.nxp.com/docs/en/user-guide/141520.pdf. After every reference, a page number and paragraph number will be included.

Every section in this file tackles the testing of different functions of the library. Refer to the index to find the section you are looking for.

Table of Contents

protocol testing:

The pn532 library implements all communication protocols (SPI, i2c and UART). SPI and i2c makes use of the bit banged versions provided by HWLIB. The UART protocol is a hardware implementation and is therefore a lot quicker.

In order to test the communication protocols, several monitor tests have been performed to see whether the communication busses perform as intended. The tests are done using a logic analyser and the Saleae Locic software : https://www.saleae.com/downloads/.

I2C

When communication between the pn532 and host controller is handled via i2c, there are a few general things to note. According to the user manual (p. 24 & 26 – section 6.1.1.3) the pn532 is configured when to i2c mode when the physical pins on the chip are set to the following configuration: P0 = 1, P1 = 0. Furthermore:

- The i2c address of the chip is: 0x48
- The chip is configured as a slave
- The chip is able to support a frequency of 400khz (in i2c mode)
- Data order used is MSB (Most Significant Bit) first
- The pn532 makes use of clock stretching

i2c communication is handled by a bus that consists of two wires. **Hence** why i2c is referred to as Two Wire interface. Additionally, the IRQ pin may also be used. This is used by the handshake mechanism implemented by the pn532. This will inform the host controller when it's ready to send a frame to the host controller.

The frames used when communicating over the i2c protocol are slightly modified compared to the once defined in the user manual's frame section. When receiving a response from the pn532, a additional byte is added in front of the frame. This byte simply contains a 0 (not ready) or a 1 (ready).

- When the RDY byte is 0 the pn532 has no frame avaialable for the host controller
- When the RDY byte is 1 the pn532 has a frame available ready to be send to the host controller

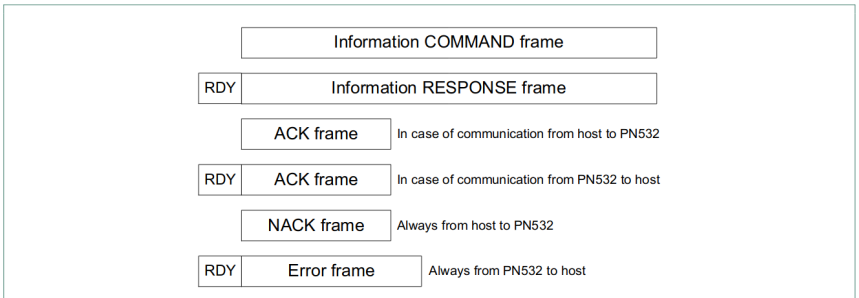


Fig 30. I2C link: frames

Set requirements

The following tests are performed to verify that the i2c bus is working correctly:

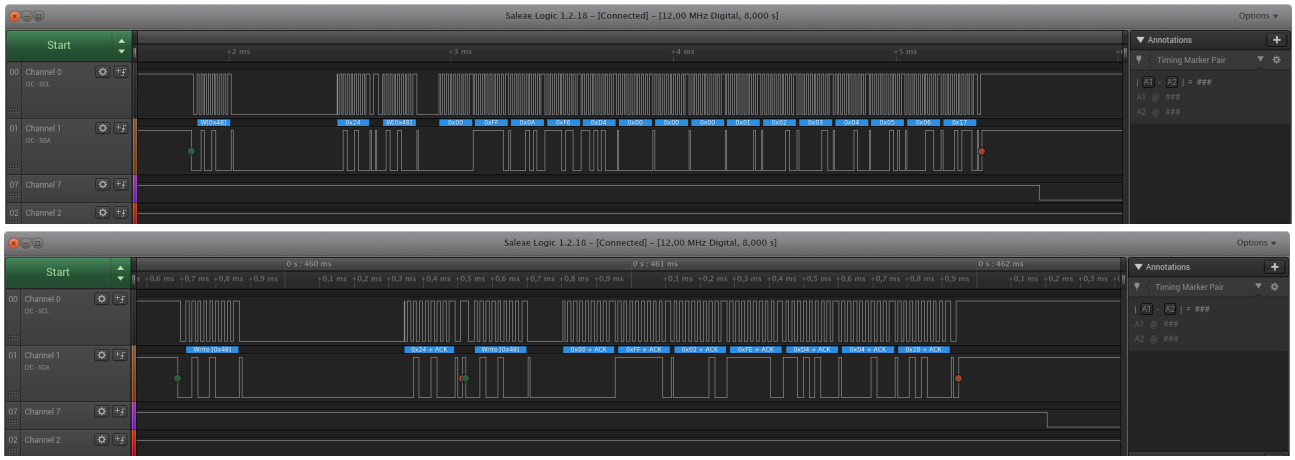
Communication will be monitored with the logic analyzer. There are several criteria the tests need to meet:

- The i2c address along with read or write command needs to be send correctly
- The SCL and SDA lines need to be in sync
- Start and stop condition needs to be right
- The command frame needs to be send correctly
- The IRQ pin must fall when a frame has been send
- The pn532 needs to send an ACK frame
- Correct data is received

The pn532 also has a build in communication line test. A frame with n bytes of random data will be send to the pn532. If the pn532 responds with the same frame, it can be concluded that the communication between host controller and pn532 is working correctly. More information about the communication line test can be found in the user manual (p. 69 – section 7.2.1).

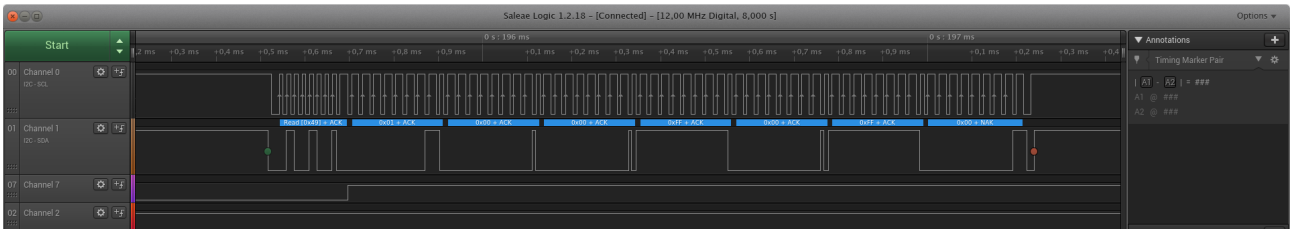
i2c test results:

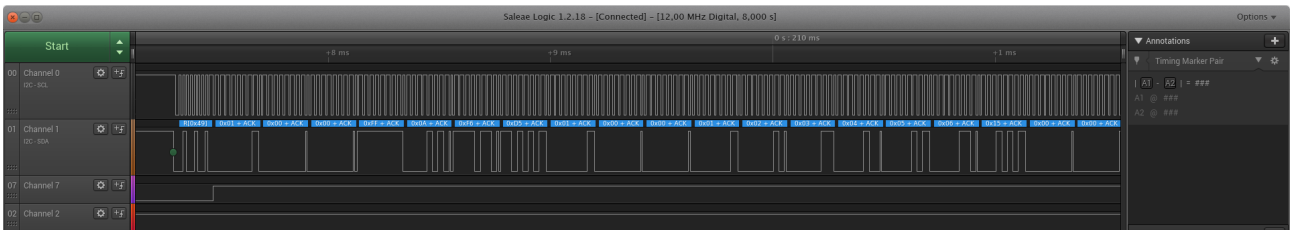
Sending data:



The two images above show that before sending a command, a write (w (0x48)) byte is send to the pn532. After that the command is written. This is the way it is supposed to go. The bytes can be individually worked out by the logic analyzer which is an indication that the format is send. The green and red circles show a start and stop condition. This is also working properly. Finally, the IRQ pin on channel 7 falls after the command has been send. This means the handshake mechanism is working as well.

Receiving data:





The top image shows the process of receiving a ACK frame. The pn532 has Acknowledged the frame, which means that the frame structure was correct and it was successfully send to the pn532. The acknowledged frame has the following structure when using i2c: { 0x01, 0x00, 0x00, 0xFF, 0x00, 0xFF, 0x00 }. As can be seen on the top image, the pn532 sends exactly this frame. The handshake mechanism is working here as well.

The bottom image shows the process of receiving a response frame from the pn532 with response data. This process is basically the same as receiving the ACK /NACK frame, but with more data. No errors are thrown here.

Conclusion

- Address is correctly send (read or write)
- SCL and SDA lines are in sync
- Start / stop conditions are working correctly
- Correct frame is send
- Handshake mechanism is working correctly
- pn532 sends ACK frames
- Correct data can be deduced by the logic analyzer
- The communication line test comes back positive

After these results, it can be concluded that i2c is working flawlessly.

SPI

When communication between the pn532 and host controller is handled via SPI, there are a few general things to note. According to the user manual (p. 24 & 25 – section 6.1.1.1) the pn532 is configured when to SPI mode when the physical pins on the chip are set to the following configuration: P0 = 0, P1 = 1. Furthermore:

- The chip is configured as a slave
- The mode for the clock used is mode 0
- Data is sampled on the first clock edge of SCK
- SCK is active high
- Data order used is LSB (Least Significant Bit) first
- The pn532 makes use of clock stretching
- The chip is able to support a frequency of 5MHz (in spi mode)

SPI communication is handled with four wires. Take care when connecting the wires. When not done properly, this can cause undefined behavior. When using SPI, the IRQ pin may also be used. This is used by the handshake mechanism implemented by the pn532. This will inform the host controller when it's ready to send a frame to the host controller.

The frames used when communicating over the SPI protocol are slightly modified compared to the once defined in the user manual's frame section. When using SPI the same RDY frame form i2c is also used. Additionally, initializing an exchange (either from host to pn532 or pn532 to host), the host controller must write a byte indicating what the following operation is to the pn532.

First byte = 0x01 : Write data from host to pn532
First byte = 0x02 : Read the status byte
First byte = 0x03 : Data reading

The image shows the altered SPI frames the pn532 uses. (User manual p. 45 – section 6.2.5)

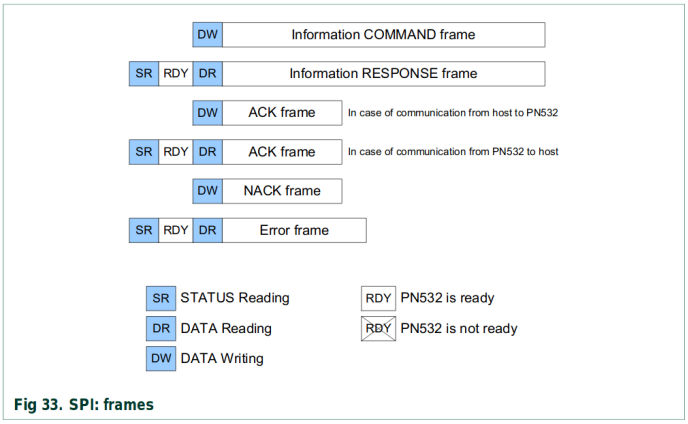


Fig 33. SPI: frames

Set requirements

The following tests are performed to verify that the SPI bus is working correctly:

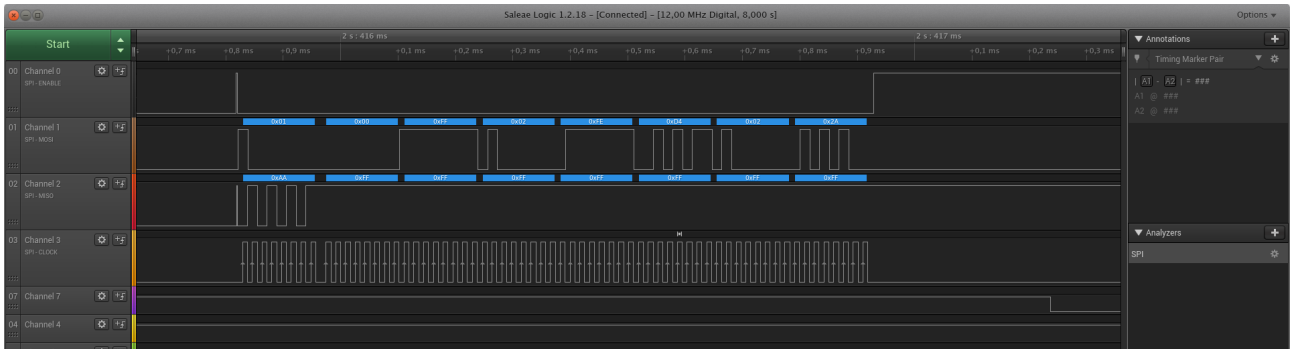
Communication will be monitored with the logic analyzer. There are several criteria the tests need to meet:

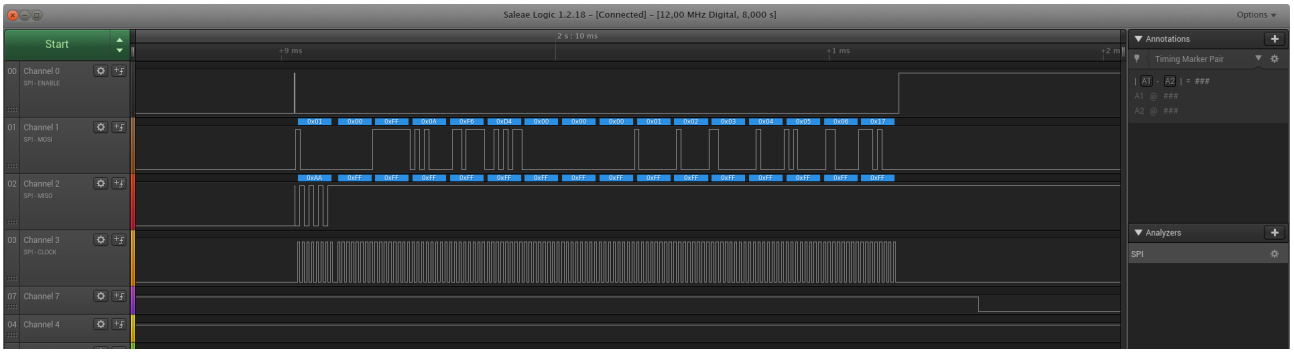
- The correct first byte is send to indicate a read, write or status reading
- The SCK is active high
- Data is send in order LSB
- The command frame needs to be send correctly
- The IRQ pin must fall when a frame has been send
- The pn532 needs to send an ACK frame
- Correct data is received and can be deduced by the Loginc Analyser

The pn532 also has a build in communication line test. A frame with n bytes of random data will be send to the pn532. If the pn532 responds with the same frame, it can be concluded that the communication between host controller and pn532 is working correctly. More information about the communication line test can be found in the user manual (p. 69 – section 7.2.1).

SPI test results:

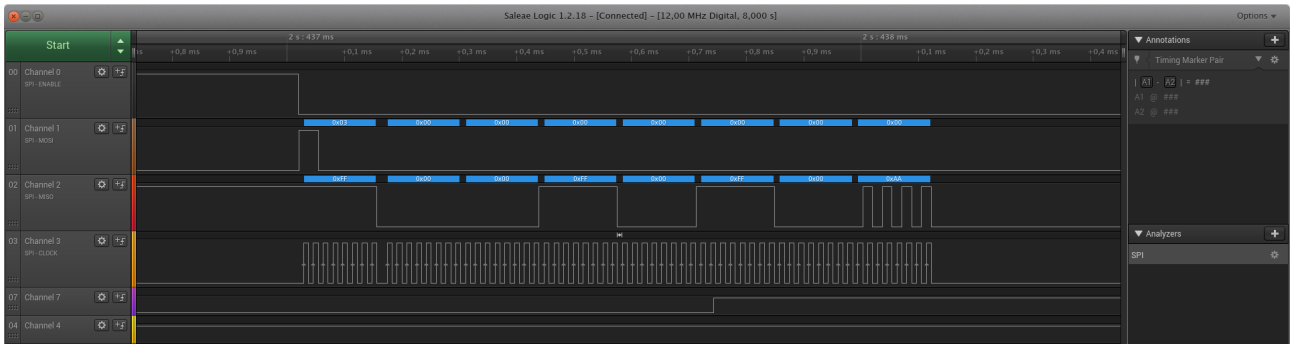
Sending data:

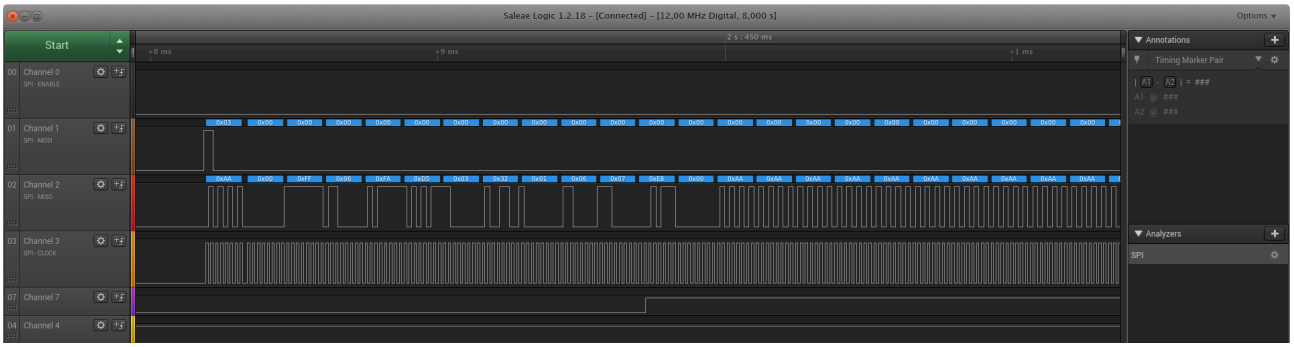




The two images above show that before sending a command, an additional byte is send to indicate the status operation. In both these cases, the byte send is 0x01 (write). The logic analyzer is set to interpret the data in LSB order. With this parameter set, the correct data can be deduced by the analyzer. Furthermore. The clock is in sync with the data and the enable line is working propery. The handshake mechanism (channel 7) also activates as soon as the command is send to the pn532.

Receiving data:





The top image shows the process of receiving a ACK frame. The pn532 has Acknowledged the frame, which means that the frame structure was correct and it was successfully send to the pn532. The acknowledged frame has the following structure when using SPI: { 0x00, 0x00, 0xFF, 0x00, 0xFF, 0x00 }. As can be seen on the top image, the pn532 sends exactly this frame. The handshake mechanism is also working properly. This is because the IRQ line rises again after the frame is read by the host controller

The bottom image shows the process of receiving a response frame from the pn532 with response data. This process is basically the same as receiving the ACK /NACK frame, but with more data. No errors are thrown here and the handshake mechanism is also working properly.

Conclusion:

- The correct first status byte is send
- SCK is indeed active high
- Data is properly send in the order of LSB
- Correct frame is send
- Handshake mechanism is working correctly
- pn532 sends ACK frames
- Correct data can be deduced by the logic analyzer
- The communication line test comes back positive

After these results, it can be concluded that SPI is working flawlessly.

HSU (High Speed UART)

When communication between the pn532 and host controller is handled via HSU, there are a few general things to note. According to the user manual (p. 24 & 26 – section 6.1.1.2) the pn532 is configured to HSU mode when the physical pins on the chip are set to the following configuration: P0 = 0, P1 = 0. The HSU configuration is as follows:

- Data bits: 8 bits
- No parity bit
- 1 stop bit
- Initial baudrate is 115200 bauds
- Data order used is LSB (Least Significant Bit) first
- Full duplex
- The chip is able to support a frequency of 1.288 Mbaud

Be sure that the Rx pin of the host controller is connected with the Tx pin of the pn532 and vice versa. The irq pin is not used in the handshake mechanism when communicating over HSU. The pn532 will simply send the response over the corresponding pin. (Tx)

The frames used when communicating over HSU are exactly the same as the frames described in the user manual. Therefore they will not be discussed here.

Set requirements

The following tests are performed to verify that the HSU bus is working correctly:

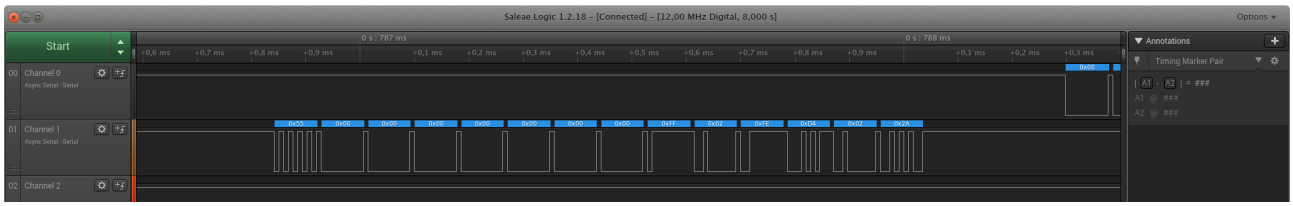
Communication will be monitored with the logic analyzer. There are several criteria the tests need to meet:

- The baudrate used between host controller and pn532 must be 115200 baud
- Data is send in order of LSB
- only one stop bit is send
- there are 8 data bits per transfer
- The command frame needs to be send correctly
- The pn532 needs to send an ACK frame
- Correct data is received and can be deduced by the Loginc Analyser

The pn532 also has a build in communication line test. A frame with n bytes of random data will be send to the pn532. If the pn532 responds with the same frame, it can be concluded that the communication between host controller and pn532 is working correctly. More information about the communication line test can be found in the user manual (p. 69 – section 7.2.1).

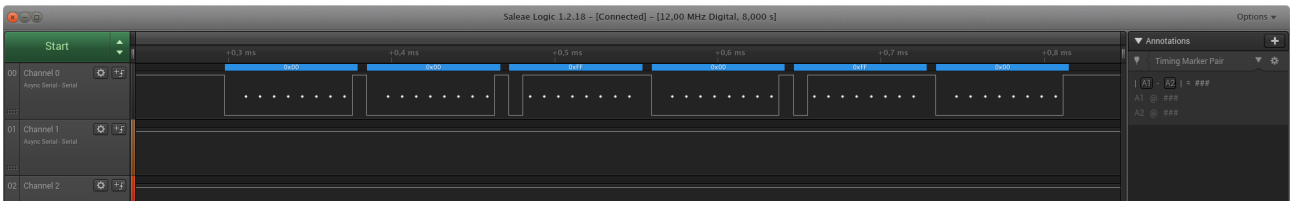
HSU test results:

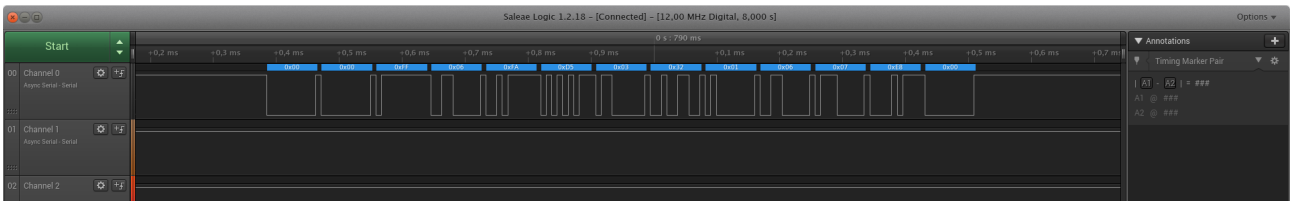
Sending data:



As can be seen in the picture above. Data is send correctly over the HSU bus. The logic analyzer has a build in feature to auto detect a baudrate. The auto detection detects a baudrate of 117kbaud, which is close enough to 115200 baud to work properly. It can also be seen that there is incoming data over channel 0. This means that the data send by the host controller is properly send with the correct settings.

Receiving data:





The two images above show the communication form pn532 to the host. The pn532 clearly responds with an ACK frame. Therefore we can conclude that that the send part of the bus is properly working and meets all criteria needed for sending a command.

However, receiving a command does not properly work. This has to do with the internal HSART buffer of the Arduino overflowing. To overcome this, an rx interrupt needs to be implemented to handle the incoming byte as soon as it is received in the internal register. Right know, with the current knowledge I have, this is not possible.

NFC library testing:

In order to test the nfc library written for the pn532, every function will be tested individually. Since tests are done on the Arduino, it is not possible to use the catch library in order to test the functions. Therefore, during the tests, debug cout messages will be implemented in the functions in order to see the way the data is being processed. Tests will be shown and discussed here.

PN532_chip class test

Function init(): Init will call the _protocol.wakeup function(). This function is used for when the pn532 is asleep and needs to be woken up. The i2c wakeup function writes it's address on the bus. This way the pn532 will acknowledge its own adress en wake up from sleep mode. The spi wakeup will send a quick pulse over the SELection pin. The HSU will send a long preamble with the format { 0x55, 0x00, 0x00, 0x00, 0x00, … }. When testing the protocol, we have already included that these functions are working. The different wake ups can be seen in the provided images.

Function sendData() This will call the _protocol.sendata function(). This function has already been tested in the protocol test section.

Function getData() This will call the _protocol.getData function(). This function has already been tested in the protocol test section.

Function writeRegister() This function writes a specific register of the pn532. According to the user manual, the response frame of the pn532 needs to be { 0xD5, 0x09 }. If the pn532 sends this response, it can be concluded that this function works properly.

```
write 9A to register 0xFFF4

2 . FE . D5 . 9 . 22 . 0 .
register written succesfully:
```

```
write 39 to register 0x6332

2 . FE . D5 . 9 . 22 . 0 .
register written succesfully:
```

As seen in the provided screenshots, the frame does indeed meet the requirement of { 0xD5, 0x09 }. Therefore it can be concluded that this function works properly Refer to the manual for more details(p. 76– section 7.2.5).

Function readRegister() This function reads from a specific register and returns that value that it has read. According to the user manual, the response frame of the pn532 needs to be { 0xD5, 0x07 }. If the pn532 sends this response, it can be concluded that this function works properly.

```
Read register 0xFFF4

3 . FD . D5 . 7 . 2 . 22 . 0 .
register content: 2
```

```
Read register 0x6332

3 . FD . D5 . 7 . 39 . EB . 0 .
register content: 39
```

As seen in the provided screenshots, the frame does indeed meet the requirement of { 0xD5, 0x07 }. Therefore it can be concluded that this function works properly. Refer to the manual for more details(p. 69 – section 7.2,4).

Function writeGPIO() This function is not tested for the simple reason that I only have one pn532 available. With this function the pn532 may be damaged and need a hardware reset. Therefore I don't want to damage my only nfc chip. Refer to the manual for more details(p. 80 – section 7.2.7).

Function readGPIO() This function reads the current status of the GPIO pins of the pn532. According to the user manual, the response frame of the pn532 needs to be { 0xD5, 0x0D }. If the pn532 sends this response, it can be concluded that this function works properly.

```
Reading GPIO
5 . FB . D5 . D . 3F . 3 . 1 . DB . 0 .
GPIO state: 7F
```

The response of the pn532 shows a correct command frame of { 0xD5, 0x0D }. The GPIO state of 8 pins are stored in one return byte. The individual sate of the pins can be deduced by shifting bits and do an and operation with 0x01. This function works correctly. Refer to the manual for more details(p. 78 – section 7.2,6).

Funciton waitForChip() This function checks if the pn532 responds within its given timeout parameter. With the irq pin connected and proper frames send, this should always return in a: chip has responded response. However, if the pn532 is asleep, it can also reach it's maximum timeout. This function is tested with and without the IRQ pin connected. If the pin is not connected, the program should give a timeout error. If the IRQ pin is connected. It should say the chip has responded.

```
waiting for chip to respond

chip has responded

waiting for chip to respond

chip has responded
```

IRQ pin connected

```
waiting for chip to respond

timeout

Error configuring SAM
```

IRQ pin not connected

Results come back positive. This function works correctly.

Function checkAck() This function is used to check if the pn532 has send an acknowledge frame. An ack frame has the following structure: { 0x01, 0x00, 0x00, 0xFF, 0x00, 0xFF, 0x00 }.

```
checking ACK

received buffer: 1 . 0 . 0 . FF . 0 . FF . 0 .

ACK frame
```

The pn532 has send an ACK frame

```
checking ACK

received buffer: 1 . 0 . 0 . FF . FF . 0 . 0 .

NACK frame
```

The pn532 has send a NACK frame

Function sendCommandAndCheckAck() This function handles the complete communication between host controller and pn532. This function will make use of sendData, getData, checkAck and the wait for chip command. All these functions work properly, and all this function does is simply calling them and comparing the statuscode to statusOK. Therefore this function will not be tested more than this.

Function getFirmwareVersion() This function is used to get the firmware version of the pn532. The required receive frame has a format of {0xD5, 0x03 } followed by its firmware version.

```
Getting firmware version

6 . FA . D5 . 3 . 32 . 1 . 6 . 7 . E8 . 0 .
Found device:     pn532
Firmware version: 1.6
Support version:  7
```

The received frame shows it contains { 0xD5, 0x03} the command has been received successfully and the firmware version is stored in the additional bytes of the frame. This function works correctly. Refer to the manual for more details(p. 73 – section 7.2.2).

Function performSelftest() This function lets the pn532 perform a self test. In this particular test case, the following string will be send to the pn532: { 0x00 , 0x01, 0x02, 0x03, 0x04, 0x05, 0x06 }. The respond frame should contain the exact same information with a 0x00 byte in front.

```
Performing self test

A . F6 . D5 . 1 . 0 . 0 . 1 . 2 . 3 . 4 . 5 . 6 . 15 . 0 .
Comminication line test: Passed
```

The screenshot of the received buffer shows that the exact same data is send back to the host controller, only with an additional 0x00 in front. Therefore the communication line test is working correctly. Refer to the manual for more details(p. 69 – section 7.2).

Function getGeneralStatus() This function returns the current state of the pn532. The received frame needs to have the following format: { 0xD5, 0x05 }. If this is the case, the following bytes will indicate the status of the pn532. Refer to the manual for more details(p. 74 – section 7.2.3).

The received frame meets the requirement of a correct frame. Therefore the following bytes can be interpreted as the information concerning the general state of the pn532. As of this test (just after booting up) no current errors have been detected, SAM is idle and no cards are controlled. This function works correctly

```
getting general status

6 . FA . D5 . 5 . 0 . 0 . 0 . 80 . A6 . 0 .
General Status pn532
Last error: 0x00
External RF field detected: 0
Amount of cards controlled: 0
Sam status connection: 80
```

Function SAMConfiguration() This function sets the mode of the pn532. If this function works properly, the pn532 should respond with a frame that contains { 0xD5, 0x15 }. An error can also occur when not using a correct SAM mode. This is shown in the images below.

```
Sam configuration

2 . FE . D5 . 15 . 16 . 0 .

Sam configuration complete
```

Correct SAM configuration

```
Sam configuration

1 . FF . 7F . 81 . 0 .

Sam configuration error
```

Incorrect SAM config

If an incorrect SAM mode is send to the pn532, it will return an error informing the user that the SAM mode entered was invalid. Refer to the manual for more details(p. 89 – section 7.2.10). This function works correctly,

Function setMaxRetries() This function sets the max retries the pn532 will perform in order to activate a target card. According to the user manual, the received frame should be { 0xD5, 0x33 } The output of this function is shown below.

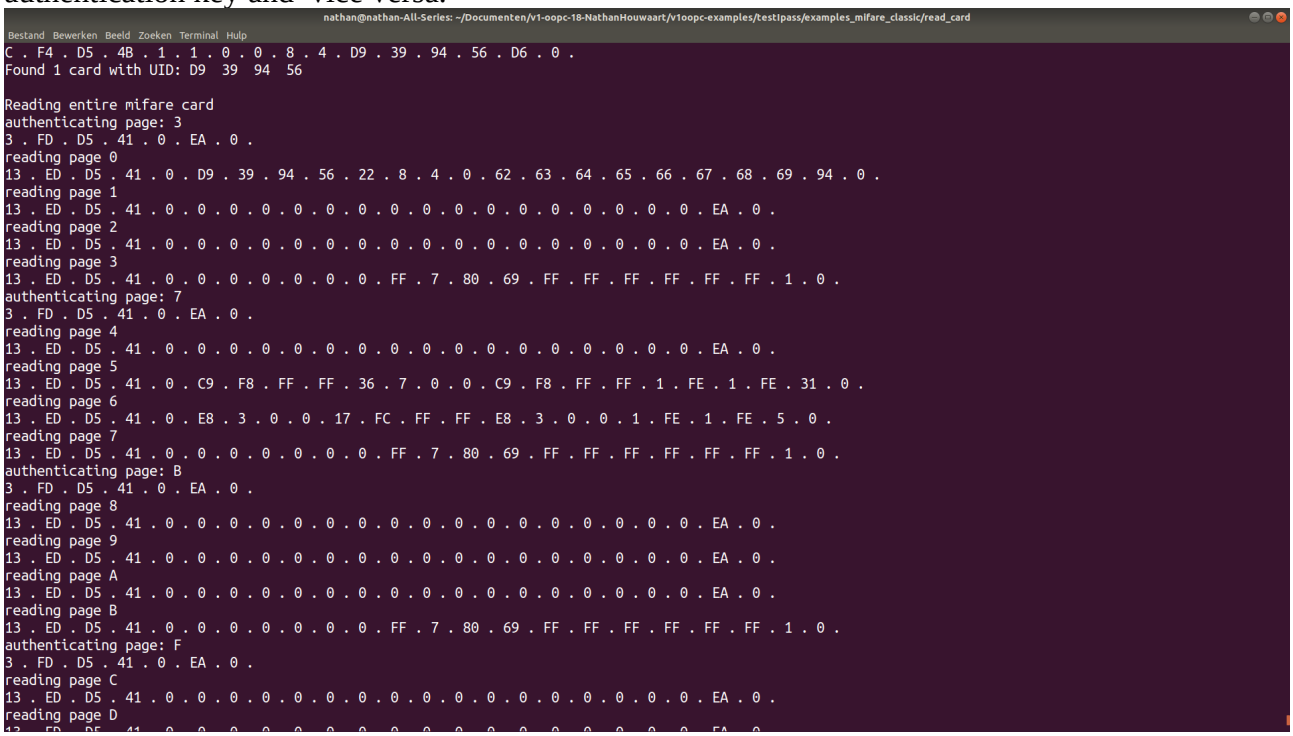
The received frame contains { 0xD5, 0x33}. Therefore it can be concluded that this function works correctly. For more information about this option, refer to the user manual (p. 101 – section 7.3.1)

```
Setting max retries

2 . FE . D5 . 33 . F8 . 0 .
Successfully set max retries
```

Functions mifareRead(), mifareAuthenticate(), mifareReadCard()

Before a specific page can be read, the sector trailer block needs to be authenticated first. This is the first action taken by the pn532. As seen below sector 3 will be authenticated before sectors 0,1,2 and 3 can be read. This process continues for the entire card. According to the user manual the read functions should have the following structure { 0xD5, 0x41, 0x00} where the 0x00 indicates whether an error has occurred. Errors thrown by this function most certainly relate to trying to authenticate with a wrong key or trying to authenticate key A whilst key B needs to be the authentication key and vice versa.



Tests show that the pn532 can correctly read and extract data from a card. This is then stored in a cardbuffer and read out. When reading this card with a phone nfc reader, it will show the exat same data. Therefore it can be concluded that the readCard and its authenticate functions work correctly. After a succesfull read, the carddata will be dumped in the terminal:

```
0xD9 0x39 0x94 0x56 0x22 0x08 0x04 0x00 0x62 0x63 0x64 0x65 0x66 0x67 0x68 0x69
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0xFF 0x07 0x80 0x69 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF

0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xC9 0xF8 0xFF 0xFF 0x36 0x07 0x00 0x00 0xC9 0xF8 0xFF 0xFF 0x01 0xFE 0x01 0xFE
0xE8 0x03 0x00 0x00 0x17 0xFC 0xFF 0xFF 0xE8 0x03 0x00 0x00 0x01 0xFE 0x01 0xFE
0x00 0x00 0x00 0x00 0x00 0x00 0xFF 0x07 0x80 0x69 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF

0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0xFF 0x07 0x80 0x69 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF

0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0xFF 0x07 0x80 0x69 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF

0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0xFF 0x07 0x80 0x69 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF

0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0xFF 0x07 0x80 0x69 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF

0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0xFF 0x07 0x80 0x69 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF

0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0xFF 0x07 0x80 0x69 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF

0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0xFF 0x07 0x80 0x69 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF

0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0xFF 0x07 0x80 0x69 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF

0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0xFF 0x07 0x80 0x69 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF

0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0xFF 0x07 0x80 0x69 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF

0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0xFF 0x07 0x80 0x69 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF

0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0xFF 0x07 0x80 0x69 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF

0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0xFF 0x07 0x80 0x69 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF

0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0xFF 0x07 0x80 0x69 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF

0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0xFF 0x07 0x80 0x69 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF
reading complete
```

Function makeValueBlock() This function is used to create a value block. A value block can be incremented and decremented by a given value. The test below shows the creation of a value block:

```
Making value block on page: 5
authenticate
succesWriting page: 5
3 . FD . D5 . 41 . 0 . EA . 0 .
write succesfull
authenticate
succes0x64  0x00  0x00  0x00  0x9B  0xFF  0xFF  0xFF  0x64  0x00  0x00  0x00  0x01  0xFE  0x01  0xFE
```

In this particular test the value block is created with a balance of 100 (hex value of 0x64). The test is consisted of: authenticating sector trailer block. Overwriting the content of page 5 . Check whether the write was successful and finally read the newly created value block.

The value block has the correct format. This function works correctly

Function decrement(), transfer() These functions are used to decrement the value of a value block. In this test, the decrement value should be 25.

```
Found 1 card with UID: D9  39  94  56

authenticate
succesOld valueblock:
0xE8  0x03  0x00  0x00  0x17  0xFC  0xFF  0xFF  0xE8  0x03  0x00  0x00  0x01  0xFE  0x01  0xFE

authenticate
succes3 . FD . D5 . 41 . 0 . EA . 0 .
authenticate
succesauthenticate
succesNew valueblock:
0xCF  0x03  0x00  0x00  0x30  0xFC  0xFF  0xFF  0xCF  0x03  0x00  0x00  0x01  0xFE  0x01  0xFE
```

The old value block has a value of 0x03E8 (1000)
the new value block has a value of 0x03CF (975)

The decrement and transfer functions are working correctly.

Function increment(), transfer() These functions are used to increment the value of a value block. In this test, the increment value should be 50.

```
Found 1 card with UID: D9  39  94  56

authenticate
succesOld valueblock:
0x64  0x00  0x00  0x00  0x9B  0xFF  0xFF  0xFF  0x64  0x00  0x00  0x00  0x01  0xFE  0x01  0xFE

authenticate
succesauthenticate
succesauthenticate
succesNew valueblock:
0x96  0x00  0x00  0x00  0x69  0xFF  0xFF  0xFF  0x96  0x00  0x00  0x00  0x01  0xFE  0x01  0xFE
```

The old value block has a value of 0x64 (100)
the new value block has a value of 0x96(150)

The increment and transfer functions are working correctly.

Conclusion:
The functions of the pn532 class are all working correctly.

Application testing

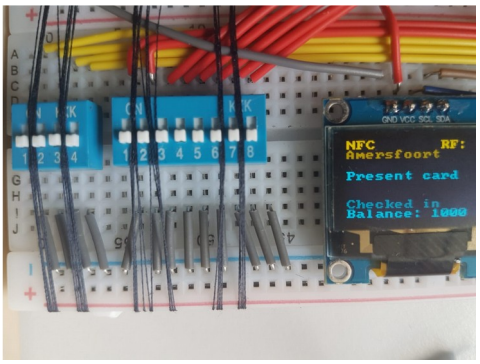
Since the nfc library is already tested and working correctly, only the application functions are tested here.

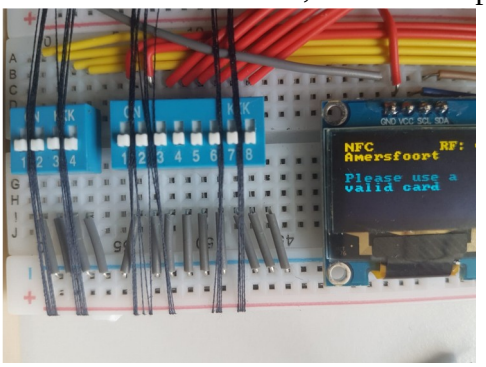
The application will be tested on the following points:

1. A card that is not correctly formatted, will not be accepted
2. A card with a proper value block but a balance below the set threshold will not be accepted
3. If the card buffer is full, no cards will be accepted
4. The correct distance must be calculated between different stations using the Haversine formula. The distance will be compared to online results
5. The card balance must be properly calculated. Cards with negative balance are stored in two's complement.
6. A card may not exceed a balance of 1000
7. If a card is removed when being in the process of checking in or out, the transaction must be aborted and the application must throw an error.
8. If the card does not travel to a different station, the display must show: canceled
9. Otherwise the display must show the check-in station and check-out station along with the new balance of the card.
10. Using the mode selection pins, a different station can be chosen. The new station must be displayed on the card.
11. …

Every test will be individually discussed here.

Test 1: A card that is not formatted properly should not be allowed to check in. This is either because the card does not contain a value block or of the access rights / access key is not valid. The function that is responsible for checking this, was tested with 15 Mifare Classic card. 10 of them were properly formatted, 5 of them were not. If a card is valid, it should display a screen that looks





like the right screen. However, if a card is not valid, the display should let the user know by throwing an error message.

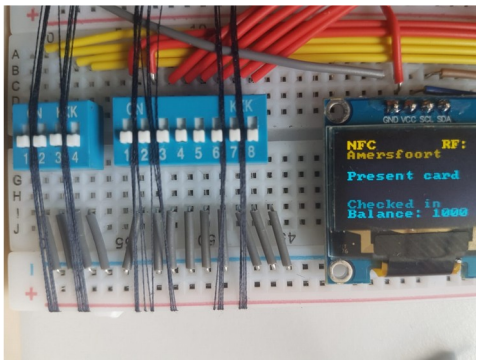
Card 1 through 10 are valid. Card 11 through 15 are not valid.

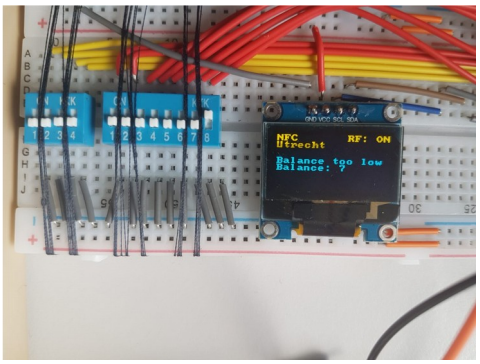
The results:

Card NR.	Display message	Pass?
1	Checked in	Pass
2	Checked in	Pass
3	Balance too low	Pass (will be discussed later)
4	Checked in	Pass
5	Checked in	Pass
6	Checked in	Pass
7	Balance too low	Pass
8	Balance too low	Pass
9	Checked in	Pass
10	Checked in	Pass
11	Invalid card	Pass
12	Invalid card	Pass
13	Invalid card	Pass
14	Invalid card	Pass
15	Invalid card	Pass

As can be seen in the table above, the valid card function works correctly. This test has passed the requirements.

Test 2: Test two is check whether a card can check in when having a balance below the threshold check in value defined in main.cpp. In this case, the threshold is 10. Which means that a card with a balance lower than 10, can not check in. This is tested with 5 cards with a balance that is high enough and 5 cards with a balance too low. The screen should display a similair screen as the right image if the balance is high enough. If the balance is too low, a message like the one on the left should appear.





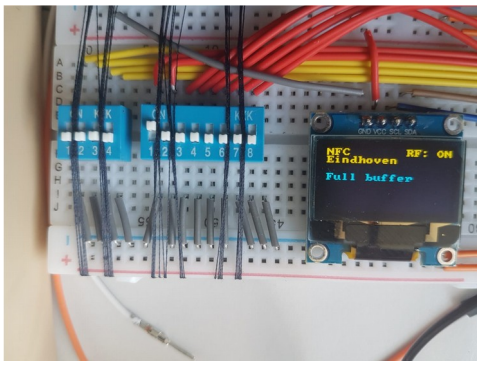
The uneven cards have a balance too low. The even cards have a balance that is higher than the threshold value.

The results:

Card NR.	**Display message**	**Pass?**
1	Balance too low	Pass
2	Checked in	Pass
3	Balance too low	Pass
4	Checked in	Pass
5	Balance too low	Pass
6	Checked in	Pass
7	Balance too low	Pass
8	Checked in	Pass
9	Balance too low	Pass
10	Checked in	Pass

The function to check whether the card balance is below a certain threshold works correctly.

Test 3: Because there is no server available to store the checked-in cards into, a local buffer is used instead. This buffer is not variable expandable and needs to be set before compiling. For this test the buffer size is set to hold a maximum of 5 cards. That means that the 6th card should not be able to check in unless one of the 5 cards checks out. The display should display a message like so:



For this test, 5 cards will be checked in normally. Then, a 6th card will be tried to check in. This should throw an error. Aftwerwards one of the checked in cars will be checked out again and the 6th card will be checked in. The card should get checked in

Test results:

- Check in of the five cars: PASS
- Check in of the sixth card: cannot be checked in – PASS
- Check out of a checked in card: PASS
- Check in of the sixth card: checked in – PASS

This test was repeated five times. All five times the test passed. Therefore this function works correctly.

Test 4: The haversine formula is tested with catch. Test results:
PASS (reference, see test folder in library).

Test 5: Correct card balance must be calculated. This function is tested with an external phone card reader that can read the Mifare card. The hex value will be transformed in to a hex value and compared to the value displayed on the oled.

This will be done with five cars with both negative and positive balance. Negative balance is an unsigned int that is wrapped around. The actual balance can be calculated by subtracting 0xFFFFFFFF from the given value.

Test results:

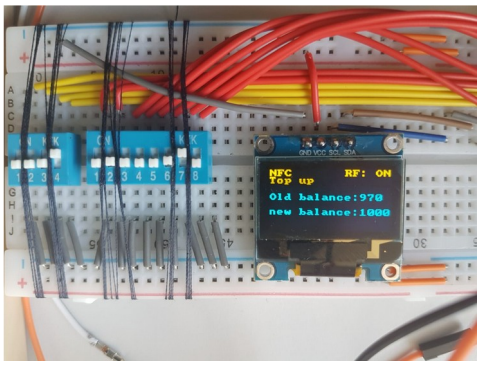
Card	**(Hex) value read by phone**	**Value interpreted by card reader**	**Pass?**
Card 1	0x84000000 (132)	132	Pass
Card 2	0x44000000(68)	68	Pass
Card 3	0x52000000(82)	82	Pass
Card 4	0xEDFFFFFF(– 18)	-18	Pass
Card 5	0xFDFFFFFF(-2)	-2	Pass

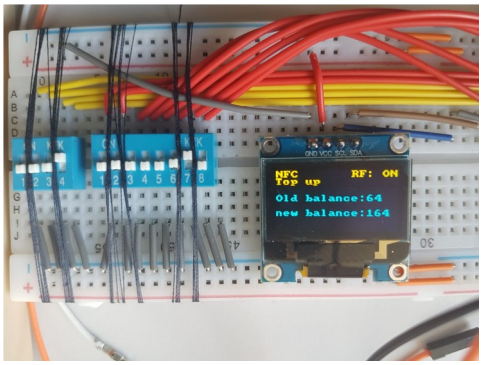
This function passed all its tests.

Test 6: When topping up a card, there is certain ceiling value set that the cards cannot exceed. If trying to top up the card by an amount that will surpass the ceiling, only the difference from the current card balance and ceiling will be incremented. For example: if the maximum card balance is set to 1000, a card has a current balance of 950 and gets topped up by 100. This will surpass the ceiling set at initialization. Therefore the function is expected to only increment 50. So the result should be no more than 1000.

This test was conducted with three cars. A card with a negative balance, a card with a positive balance, and a card that would surpass the max card value if it would be incremented by the full amount. The increment value is 100.

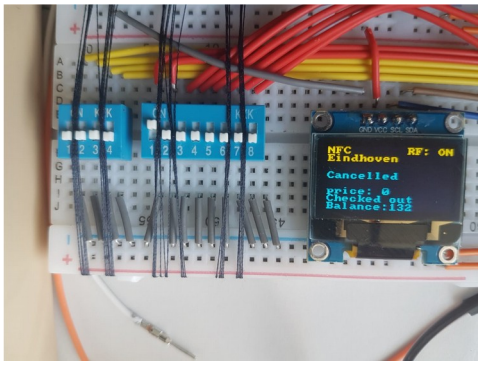
Test results:
Test results show that all three top ups have passed. Below are two images of the test result. As can be seen in the images, a card value with a balance of 64 gets incremented correctly by 100. However the card with a balance of 970 would exceed the set ceiling. Therefore the card reader only accepts the card to be topped up by 30. This function works correctly.





Test 8: If a card does not travel to a different station, it must display cancelled instead of travelled form X to Y.

To perform this test, five cards will be checked in at different stations. At a random order, all cards will be checked out again at their starting station. This test will pass if all check outs dispay: cancelled.

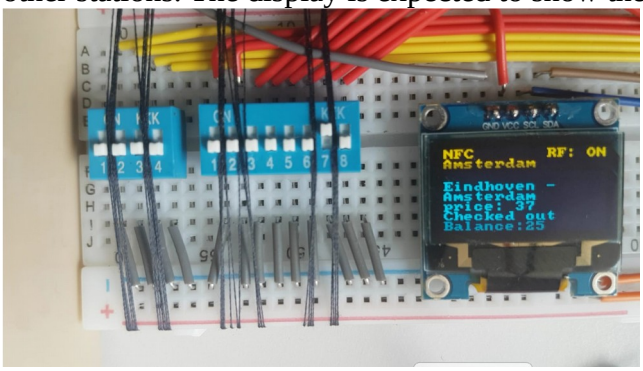


Test resuts

Card NR.	Checkin / out station	Message	Pass?
1	Haarlem	Cancelled	Pass
2	Amersfoort	Cancelled	Pass
3	Eindhoven	Cancelled	Pass
4	Den Haag	Cancelled	Pass
5	Rotterdam	Cancelled	Pass

The test results show that upon checking out at the same station, all cards will produce a message saying that the journey is canceled.

Test 9: Application must show the correct travel path the card has taken. From station x to staion y. This is tested with five cards all checked in at different stations and then randomly checked out at other stations. The display is expected to show the following format:



Test results:

Card NR.	Checkin	Checkout	Message	Pass?

1	Haarlem	Utrecht	Haarlem – Utrecht	Pass
2	Amersfoort	Rotterdam	Amersfoort – Rotterdam	Pass
3	Eindhoven	Gouda	Eindhoven – Gouda	Pass
4	Den Haag	Schiphol	Den Haag – Schiphol	Pass
5	Rotterdam	Eindhoven	Rotterdam – Eindhoven	Pass

The application shows the correct travel path. Therefore this function works correctly

Test 10: The display shows the correct modes and station when switching mode and selection pins. This test is conducted to see whether the oled display shows the right information regarding station and mode.

Pin configuration(pin 1, 2, 3, 4, 5, 6, 7,8):

00000000 = Station amersfoort
00000001 = Station Utrecht
00000010 = Station Amsterdam
00000100 = Station Schiphol
00001000 = Station Haarlem
00010000 = Station Den Haag
00100000 = Station Rotterdam
01000000 = Station Gouda
10000000 = Station Eindhoven

Mode:
0000 = Station name
0001 = Top up
0000 = Switch back to right station

Test results:

Pin configuration	**Oled display**	**Pass?**
00000000	Amersfoort	Pass
00000001	Utrecht	Pass
00000010	Amsterdam	Pass
00000100	Schiphol	Pass
00001000	Haarlem	Pass
00010000	Den Haag	Pass
00100000	Rotterdam	Pass
01000000	Gouda	Pass
1000000	Eindhoven	Pass
Mode = 000	Station name	Pass
Mode = 001	Top up	Pass
Mode = 000	Top up	**Failed**

After conducting the tests it can be concluded that most of this function is working properly. However: If the mode was set to top up mode and then switched back to travel mode, top up stays written on the screen. This is a bug that needs fixing.

Conclusion

The application works correctly for the most part. There is a minor bug when checking out where the card balance is not displayed correctly. This is however not a harmful bug and will not cause the check-in buffer to be bugged.

There is one other bug when switching back from top up mode to travel mode. This bug is rather annoying and needs fixing if there is time.

A quick side note: *This library is tested on several aspects, but is (due to time constraints) not tested to the fullest potential. In order to test this library further in the future, more aspects of every function need to be tested. In fact, the best tests are conducted with people who are not familiar with the product. If the library still works properly when used by these people, I can conclude that the library is working bug free.*