1.2 Case study: Using the PN532 RFID/NFC Module with STM32 via SPI

1.2.1 Introduction to RFID and NFC Communication

Radio-Frequency Identification (RFID) is a widely used wireless communication technology that enables data to be read from or written to tags using radio waves. These tags can be passive, meaning they do not contain a battery and are powered by the electromagnetic field generated by the reader device. The communication typically occurs over short distances and is commonly used in applications such as access control, asset tracking, and contactless payment systems. Near Field Communication (NFC) is a specialized subset of RFID that operates at a frequency of 13.56 MHz. Unlike traditional RFID systems that usually support one-way communication from tag to reader, NFC also supports two-way communication. This makes it suitable for a broader range of interactive applications, including data exchange between smartphones, secure authentication, and emulation of RFID cards. When an NFC-enabled tag or device is brought into proximity with an NFC reader, the reader generates an electromagnetic field that powers the tag and enables it to respond. This interaction allows data to be transmitted without the need for physical contact or an internal power source in the tag.

1.2.2 The PN532 NFC Controller: Overview and Features

The PN532 is a highly versatile NFC controller developed by NXP Semiconductors. Based on the 80C51 microcontroller core, it integrates everything needed for NFC communication and supports a wide variety of protocols and operational modes. This controller can operate in several modes, including:

- **Reader/Writer Mode**, which allows the PN532 to read and write to passive NFC tags.
- **Card Emulation Mode**, where the PN532 behaves like an NFC tag, allowing it to be read by external readers.
- **Peer-to-Peer Mode**, which enables direct communication between two NFC devices.

It supports widely adopted NFC standards, such as:

- ISO/IEC 14443 Type A and B
- MIFARE Classic 1K/4K
- FeliCa (used in Japan and other regions)
- ISO/IEC 18092, also known as NFCIP-1 (NFC Interface and Protocol)

Another important feature of the PN532 is its flexibility in host communication interfaces. It can communicate with a microcontroller or host system via one of the following:

- Serial Peripheral Interface (SPI)
- I²C (Inter-Integrated Circuit)
- High-Speed UART (Universal Asynchronous Receiver-Transmitter)

This flexibility makes it easy to integrate into different hardware platforms, including popular microcontrollers like those in the STM32 family.

Communicating with the PN532 via SPI

The Serial Peripheral Interface (SPI) is a common communication protocol used in embedded systems to allow a microcontroller to exchange data with peripheral devices. It is a full-duplex, master-slave protocol, meaning both devices can send and receive data simultaneously, and the master (in this case, the STM32 microcontroller) controls the clock and flow of data.

When using the SPI interface, the PN532 acts as a slave device. Communication is established through four signal lines:

- **SCLK** (Serial Clock): Generated by the master to synchronize data transmission.
- **MOSI** (Master Out, Slave In): Carries data from the STM32 to the PN532.

- **MISO** (Master In, Slave Out): Sends data from the PN532 back to the STM32.
- **NSS** (Chip Select): Enables communication with the PN532 when pulled low.

The PN532 uses SPI Mode 0, where the clock polarity (CPOL) is 0 (low) and the clock phase (CPHA) is also 0. This means data is sampled on the rising edge of the clock signal and the clock idles in the low state. It's also important to note that data is transmitted with the least significant bit (LSB) first, which is a somewhat uncommon behavior in SPI communication and should be configured accordingly in the STM32 SPI peripheral. To ensure reliable operation, the SPI clock frequency should not exceed 1.2 MHz, as stated in NXP's Application Note AN133910 (`https://www.nxp.com/docs/en/nxp/application-notes/AN133910.pdf`).

In the following subsection, we briefly present only selected features of the PN532 module relevant to our application. For a comprehensive overview of all functionalities, the reader is referred to the official PN532 user manual (UM0701-02), available at: `https://www.nxp.com/docs/en/user-guide/141520.pdf`.

1.2.3 The Communication Protocol and Frame Structure

Communication between the host microcontroller and the PN532 follows a well-defined frame-based protocol (Figure 1.1). Each interaction involves specific steps, typically structured as follows:

1. **Sending a Command:** The STM32 constructs and sends a command frame to the PN532, instructing it to perform a specific action (e.g., read a tag, configure settings).
2. **Polling the status register:** After sending the command, the STM32 continuously polls the PN532's status register to check whether the PN532 has received and processed the frame, and the ACK frame is ready.
3. **Read Acknowledgment:** Once the status register equals `0x01`, the host controller reads the ACK frame (Acknowledgment Frame).
4. **Polling the status register:** Once the ACK is received, the host continues to poll the status register until the PN532 indicates that a response is ready.
5. **Read a Response:** The STM32 reads the response frame containing the requested data or status information.

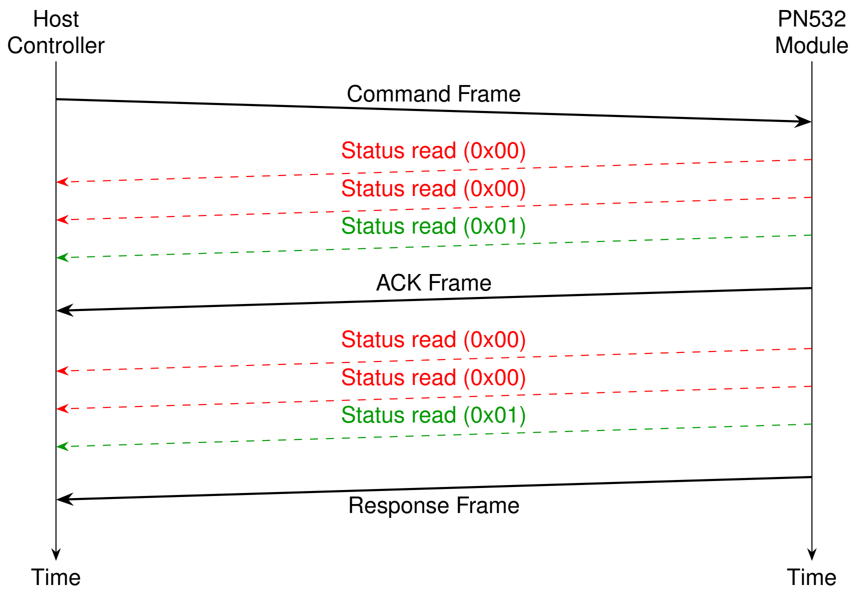

Fig. 1.1: General principle of SPI communication between Host Controller and PN532

Types of communication frames

Before initiating any SPI communication between the host controller and the PN532 (whether the communication is from the host to the PN532 or vice versa), the host must first send a byte that indicates the type of operation to be performed. This byte informs the PN532 about the nature of the upcoming communication and is as follows:

- `xxxxxx10b` – Indicates the host wants to read the status.
- `xxxxxx01b` – Indicates the host is writing data to the PN532.
- `xxxxxx11b` – Indicates the host is reading data from the PN532.

After this initial byte, a corresponding frame is transmitted. There are three types of frames that can be exchanged: command frames, ACK frames, and response frames.

For instance, when sending a command to the PN532, the host first sends the byte `0x01` over the MOSI line (indicating a data write operation), followed by the command frame, which is also transmitted over the MOSI line. To acknowledge the receipt of a command, the host sends `0x03` (indicating a data read operation) over the MOSI line, after which the PN532 responds with the ACK frame over the MISO line. Finally, to receive the actual response to the command, the host again sends `0x03` (data read) over the MOSI line, and the PN532 responds with the response frame.

As said, the PN532 recognizes different types of frames:

- **Command Frames:**

0x00	0x00	0xFF	LEN	LCS	TFI=0xD4	Cmd	D1 ... Dn	DCS	0x00

Fig. 1.2: Command frame.

Command frames are sent by the host to initiate an operation. The command frame is a structured sequence of bytes transmitted from the host to the PN532 to initiate an operation. Each byte in the frame serves a specific purpose, ensuring proper synchronization, data integrity, and clear instruction to the PN532. The frame begins with a **Preamble** byte `0x00`, followed by a **Start Code** composed of two fixed bytes `0x00` and `0xFF`. These three bytes help the PN532 to identify the beginning of a valid frame. Next, the **Length** byte (LEN) specifies the number of bytes in the data payload (TFI + Cmd + No. of Data bytes), and is followed by its 2'complement, **Length Checksum (LCS)**, such that the sum of the Length and LCS equals `0x00` modulo 256. This acts as a first level of validation for frame integrity. The frame continues with the **Frame Identifier**, known as TFI (Target Frame Identifier). When sending a command from the host to the PN532, this byte is always `0xD4`. It indicates the direction of the communication and marks the start of the actual command data. Following the TFI, the frame contains the **Cmd + Data** bytes. The first byte (Cmd) is the command code (e.g., `0x02` for *GetFirmwareVersion*) followed by any required parameters for the given command in the data byres. The number and content of these bytes depend on the specific command being issued. To ensure the reliability of data transmission, a **Data Checksum (DCS)** byte follows the data. This checksum is calculated such that the sum of the TFI, Cmd, Data bytes, and DCS equals `0x00` modulo 256. The final byte of the frame is the **Postamble**, another fixed value `0x00`, signifying the end of the frame.

- **ACK Frames:**

0x00	0x00	0xFF	0x00	0xFF	0x00

Fig. 1.3: ACK frame.

These frames are sent by the PN532 to confirm that a command has been received successfully. The ACK frame sent by the PN532 is a 6-byte fixed frame used to acknowledge the successful receipt of a command frame from the host. It has a constant format and does not contain any dynamic fields such as length or checksums. The structure of the PN532 ACK frame is always `0x00,0x00,0xFF,0x00,0xFF,0x00`

- **Response Frames:**

0x00	0x00	0xFF	LEN	LCS	TFI=0xD5	Cmd+1	D1 ... Dn	DCS	0x00

Fig. 1.4: Response frame.

Response frames are sent by the PN532 to provide feedback or results following a command sent by the host. The response frame is a structured sequence of bytes transmitted from the PN532 back to the host, and each byte serves a specific role in conveying the operation's outcome. The frame begins with a **Preamble** byte `0x00`, followed by a **Start Code** composed of two fixed bytes: `0x00` and `0xFF`. These initial bytes help the host identify the beginning of a valid response frame from the PN532. Next, the **Length** byte (LEN) specifies the total length of the response frame, excluding the Preamble and Start Code. This is followed by its 2's complement, **Length Checksum (LCS)**, such that the sum of the Length and LCS equals `0x00` modulo 256. This ensures the integrity of the length information and serves as an initial check for the frame's validity. The frame then contains the **Frame Identifier**, which in response frames is always `0xD5`. This byte marks the response's direction and identifies it as a valid response frame from the PN532. Following the Frame Identifier (TFI), the **Cmd+1** byte follows. It corresponds to the command that was originally issued by the host incresed by 1. It indicates which command is being responded to. For instance, if the host sent a `0x02` command (e.g., *GetFirmwareVersion*), this byte in the response will be `0x03`, ensuring the host knows which response corresponds to which command. After the **Cmd+1** byte, the frame contains the **Data Field**, which contains the result or additional data from the operation. The number of bytes in the data field depends on the specific command and the operation performed. These bytes carry the actual response from the PN532. To ensure data integrity, a **Data Checksum (DCS)** byte follows the data. The DCS is calculated such that the sum of the FTFI, Cmd+1, Data bytes, and DCS equals `0x00` modulo 256. This checksum ensures that the transmitted response data has not been corrupted during transmission. Finally, the frame ends with the **Postamble**, which is another fixed value `0x00`, marking the end of the response frame.

Before any data can be read from the PN532, the STM32 must check the PN532's status by polling the status register and waiting for the READY bit to be set. This ensures that the PN532 has completed its previous operation and is ready to communicate again.

1.2.4 *Minimal Set of Commands to Communicate with the PN532 and Read ISO 14443-A Passive Tags*

In order to communicate with the PN532 and read ISO 14443-A passive tags, the following three commands need to be issued in sequence: `GetFirmwareVersion`, `SAMConfiguration`, and `InListPassiveTarget`. Each command plays a critical role in setting up the communication and obtaining tag information.

GetFirmwareVersion Command

The `GetFirmwareVersion` command is used to obtain the firmware version of the PN532. This is the first command issued to ensure that the PN532 is present and functioning. The command frame for `GetFirmwareVersion` contains the command code `0x02`, and it does not require any parameters. After issuing this command, the PN532 will respond with a frame containing the firmware version information in the data field. The response consists of the following four bytes, each with a specific meaning:

- **Byte 1: IC Version**
  This byte represents the version of the integrated circuit (IC) in the PN532. For the PN532, this value is always `0x32`.
- **Byte 2: Firmware Version**
  This byte indicates the version of the firmware currently running on the PN532.

- **Byte 3: Firmware Revision**
  This byte represents the revision number of the firmware.
- **Byte 4: Supported Functionalities**
  This byte indicates which functionalities are supported by the firmware. The least significant three bits (bits 0, 1, and 2) represent the support for specific ISO standards:
  - **Bit 2 (ISO18092)**: If this bit is set to 1, the PN532 supports ISO/IEC 18092 (NFC). If it is set to 0, the functionality is not supported.
  - **Bit 1 (ISO/IEC 14443 Type B)**: If this bit is set to 1, the PN532 supports ISO/IEC 14443 Type B. If it is set to 0, the functionality is not supported.
  - **Bit 0 (ISO/IEC 14443 Type A)**: If this bit is set to 1, the PN532 supports ISO/IEC 14443 Type A. If it is set to 0, the functionality is not supported.

  In the case of PN532, the response is aleays `0x32,0x01,0x06,0x07`. The value `0x07` of the last byte means that all three functionalities are supported (since the least significant three bits are set to 1).

SPI frames for the GetFirmwareVersion command are :

- **Command Frame (sent by host to PN532):**

```
Preamble:      0x00
Start Code:    0x00 0xFF
Length:        0x02  (7 bytes in total)
LCS:           0xFE  (2'complement of Length)
TFI:           0xD4  (target frame identifier)
Cmd:           0x02  (GetFirmwareVersion)
DCS:           0x2A  (2'complement of (TFI+Cmd))
Postamble:     0x00
```

- **Response Frame (received from PN532):**

```
Preamble:      0x00
Start Code:    0x00 0xFF
LEN:           0x06  (6 bytes in total)
LCS:           0xFA  (2'complement of Length)
TFI:           0xD5  (response frame identifier)
Cmd:           0x03  (response to GetFirmwareVersion)
Data:          0x32 0x01 0x06 0x07
Data Checksum: 0xE8  (2'complement of (TFI+Cmd+Data))
Postamble:     0x00
```

SAMConfiguration Command

The `SAMConfiguration` command is used to configure the PN532's Secure Access Module (SAM) to operate in passive target mode. This is an essential step before attempting to communicate with any ISO 14443-A tags. The `SAMConfiguration` command configures the SAM (Secure Access Module) to work in either passive or active mode. It takes three parameters:

- **First Parameter (Mode)**: This parameter sets the SAM to normal (passive) mode. For passive mode operation, the first parameter is set to `0x01`. The passive mode is used for communication with passive tags in the environment. An alternative is Active Mode, which would be configured with a different parameter value (e.g., `0x02` for Virtual Card Mode).
- **Second Parameter (Timeout)**:
  The Timeout parameter defines the time period after which the PN532 should stop waiting for a tag response. The timeout is specified in units of 50 milliseconds (ms). For example, a Timeout value of 0x01 would represent 50 ms, 0x02 would represent 100 ms, and so on.

  - If the Timeout parameter is set to 0x00, it means no timeout control is applied. In this case, the PN532 will wait indefinitely for a response, and there is no timeout after which the operation will automatically terminate.
  - The maximum timeout value is `0xFF`, which equals 12.75 seconds (calculated as `0xFF` * 50 ms = 12.75 seconds)).

- **Third Parameter (IRQ)**:
  This parameter determines how the PN532 handles the P70_IRQ pin. If the value is `0x00`, the P70_IRQ pin remains at a high level and is not managed by the PN532. If the value is `0x01`, the PN532 actively drives the P70_IRQ pin, which can be used for interrupt-driven communication with the host.

SPI Frames for SAMConfiguration Command:

- **Command Frame (sent by host to PN532):**

```
Preamble:      0x00
Start Code:    0x00 0xFF
LEN:           0x05           (5 bytes in total)
LCS:           0xFB           (2'complement of LEN)
TFI:           0xD4           (target frame identifier)
Cmd:           0x14           (SAMConfiguration)
Data:          0x01 0x14 0x01  (passive mode, timeout=1000ms, P70_IRQ)
DCS:           0x02
Postamble:     0x00
```

- **Response Frame (received from PN532):**

```
Preamble:      0x00
Start Code:    0x00 0xFF
LEN:           0x02
LCS:           0xF9  (2'complement of LEN)
TFI:           0xD5
Cmd:           0x15  (response to SAMConfiguration)
Data Checksum: 0x16  (2'complement of TFI+Cmd)
Postamble:     0x00
```

InListPassiveTarget Command

The `InListPassiveTarget` command is used to initiate the detection of ISO 14443-A passive tags. After the SAM is configured, the PN532 listens for a passive target (ISO 14443-A tag) and returns information about any detected tag. The `InListPassiveTarget` command is used to check for passive tags. This command requires two parameters:

1. **MaxTg**: This parameter specifies the maximum number of targets the PN532 should detect. The PN532 can handle a maximum of 2 targets at once, so this field should not exceed `0x02`.
2. **BrTy**: This parameter specifies the baud rate and modulation type to be used during the initialization. For example,`0x00` is used for 106 kbps with Type A modulation (ISO/IEC14443 Type A).

The PN532 will return the following response data:

1. **NbTg (Number of Targets)**: This byte indicates the number of targets detected by the PN532. The value can range from 0 (no targets detected) to 2 (maximum number of targets detected).
2. **Tg (Target Identifier)**: The target number, which helps identify which target's information follows in the response. If multiple targets are detected, each one will have a corresponding entry in the response.
3. **SENS_RES (Sens Response)**: This byte contains the response to the sensor command. It provides information about the detected target's capabilities. For example, it can indicate whether the target is compatible with the PN532 for communication.

4. **SEL_RES (Select Response)**: This byte contains the response to the selection process. It provides information about the selected target's status, such as whether it is ready for further communication or if there was an error during the selection process.
5. **NFCIDLength (NFC ID Length)**: This byte indicates the length of the NFCID in bytes. The NFCID is the unique identifier for the detected NFC tag, and this byte tells how many bytes of the NFCID will follow in the response. This length can either be 4 bytes (for Type A) or 7 bytes (for Type B) depending on the type of tag detected.
6. **NFCID (NFC Identifier)**: This field contains the unique identifier of the detected NFC tag. The NFCID is either 4 bytes or 7 bytes long, depending on the type of tag detected. The length of the NFCID is determined by the NFCIDLength field. The NFCID is used to identify the specific tag in communication processes.

SPI Frames for InListPassiveTarget Command (assuming the NFC Identifier is `0x53,0x2E,0xF4,0x1A`):

- **Command Frame (sent by host to PN532):**

```
Preamble:      0x00
Start Code:    0x00 0xFF
LEN:           0x04
LCS:           0xFC       (2'complement of LEN)
TFI:           0xD4       (target frame identifier)
Cmd:           0x4A       (InListPassiveTarget)
Data:          0x01 0x00  (MaxTg, BrTy)
DCS:           0xE1       (2'complement of (TFI+Cmd+Data))
Postamble:     0x00
```

- **Response Frame (received from PN532):**

```
Preamble:      0x00
Start Code:    0x00 0xFF
LEN:           0x0C
LCS:           0xF4       (complement of LEN)
TFI:           0xD5       (response frame identifier)
Cmd:           0x4B       (response to InListPassiveTarget)
Data:          0x01 0x01 0x00 0x04 0x08 0x04 0x53 0x2E 0xF4 0x1A
DCS:           0x3F       (2'complement of (TFI+Cmd+Data))
Postamble:     0x00
```

In the above response, the data field contains `0x01 0x01 0x00 0x04 0x08 0x04 0x53 0x2E 0xF4 0x1A`, which can be interpreted as follows:

- **NbTg** = 1: This indicates that one target was detected.
- **Tg** = 0: This is the target identifier, indicating that the first target is being referred to.
- **SENS_RES** = 0x04: This is the sensor response byte, indicating the capabilities or type of the detected target.
- **SEL_RES** = 0x08: This is the select response byte, indicating the status of the selected target.
- **NFCIDLength** = 0x04: This byte indicates that the NFCID is 4 bytes long.
- **NFCID** = `0x53,0x2E,0xF4,0x1A`: This is the unique identifier (NFCID) of the detected NFC tag.

1.2.5 Interfacing PN532 with STM32H750 Using SPI

In this section, we will discuss the implementation of basic SPI functions required for communication with the PN532 NFC module using the STM32H750. The code provided demonstrates how to interact with the PN532 over the SPI interface by implementing simple functions for transmitting and receiving data, as well as controlling the chip select (SS) pin. These functions are essential for ensuring proper communication between the STM32H750 microcontroller and the PN532 NFC module, enabling tasks such as sending commands, receiving responses, and managing the connection.

1.2.5.1 SPI Functions

```c
/****************************************************
 *                                                  *
 *                    SPI Functions                 *
 *                                                  *
 ****************************************************/

/*
 * This is just a wrapper around HAL_GPIO_WritePin used to set the pin for Slave Select
 */
static inline void SS_HIGH (void) {
  HAL_GPIO_WritePin(PN532_SS_GPIO_Port, PN532_SS_Pin, GPIO_PIN_SET);
}

/*
 * This is just a wrapper around HAL_GPIO_WritePin used to reset the pin for Slave Select
 */
static inline void SS_LOW (void) {
  HAL_GPIO_WritePin(PN532_SS_GPIO_Port, PN532_SS_Pin, GPIO_PIN_RESET);
}

/*
 * SPI_read : it receives one byte over SPI in blocking mode
 */
static uint8_t SPI_read(SPI_HandleTypeDef *hspi){
  uint8_t RxData[1];
  HAL_StatusTypeDef status;

  status = HAL_SPI_Receive(hspi, RxData, 1, HAL_MAX_DELAY);
  return RxData[0];
}

/*
 * SPI_write : it sends one byte over SPI in blocking mode
 */
static void SPI_write(SPI_HandleTypeDef *hspi, uint8_t data){
  uint8_t TxData[1];
  HAL_StatusTypeDef status;

  TxData[0] = data;
  status = HAL_SPI_Transmit(hspi, TxData, 1, HAL_MAX_DELAY);
}
```

Listing 1.1: The code for the basic SPI functions.

The code for the basic SPI functions described in this subsection is presented in 1.1. The following SPI functions are defined to facilitate this communication:

1. SS Control Functions

The functions for controlling the slave select (SS) pin are used to manage the chip's active state during communication. The slave select pin must be properly toggled before initiating communication with the PN532 to ensure that the chip is correctly addressed.

- `SS_HIGH`: This function sets the SS pin high, disabling communication with the PN532.
- `SS_LOW`: This function sets the SS pin low, enabling communication with the PN532.

Supposing we are using `GPIOB` Pin 4 for the Slave Select (SS) line, the following constants should be also defined:

```c
#define PN532_SS_GPIO_Port   GPIOB
#define PN532_SS_Pin         GPIO_PIN_4
```

2. SPI Communication Functions

The following functions facilitate the sending and receiving of data via the SPI interface:

- `SPI_read`: This function receives a single byte of data from the PN532 over the SPI interface. It operates in blocking mode, waiting until the data is fully received before returning the received byte.
- `SPI_write`: This function sends a single byte of data to the PN532 over the SPI interface. Like `SPI_read`, it operates in blocking mode, ensuring that the byte is transmitted before proceeding.

These basic SPI functions form the foundation for more complex interactions with the PN532, such as sending commands and receiving responses as part of the NFC tag communication process.

1.2.5.2 PN532 Internal Helper Functions

The PN532 NFC controller requires specific SPI-level communication sequences for transmitting commands and receiving responses. The helper functions described below encapsulate these low-level procedures and are used internally to manage command execution, handshaking, and error checking.

1. Checking Device Readiness: `isReadyToSend()`

```
/*
 * The SPI interface includes a specific register allowing the host
 * controller to know if the PN532 is ready to receive or to send data back.
 *
 * isReadyToSend: Sends a status read command and checks if the PN532 is
 *                ready to send data back.
 *                It is a helper function used within waitToBeReady()
 *
 * @ return: STATUS_532_OK if the PN532 has a frame available to be
 *           transferred to the host controller.
 *
 */
static StatusCode532_t isReadyToSend(void) {
    SS_LOW();

    SPI_write(&SPI_HANDLER, PN532_SPI_STATREAD);  // write SPI STATUS READ command to PN532 module
    uint8_t status = SPI_read(&SPI_HANDLER);      // and read response from PN532

    SS_HIGH();

    if (status == PN532_SPI_READY){  // check if PN532 is ready (LSB=1) and return the result
     return STATUS_532_OK;
    }
    else return STATUS_532_ERROR;
}
```

Listing 1.2: Checking Device Readiness.

This function checks whether the PN532 is ready to send data back to the host controller. It sends a special `STATUSREAD` command over SPI and evaluates the response from the PN532. If the response's least significant bit is set, the device is ready. It returns `STATUS_532_OK` on success or `STATUS_532_ERROR` otherwise. This function is primarily used in the `waitToBeReady()` routine.

2. Waiting for PN532 to Become Ready: `waitToBeReady(uint16_t wait_time)`

```
/*
 * waitToBeReady:
 * Waits for the PN532 to become ready, up to a timeout in milliseconds.
```

```c
 * It is a helper function used within sendCommand() and readResponse
 *
 * @return: STATUS_532_OK if ready, STATUS_532_ERROR if timeout
 */
static StatusCode532_t waitToBeReady(uint16_t wait_time) {
    while (isReadyToSend() != STATUS_532_OK) {
        HAL_Delay(1);
        wait_time--;
        if (wait_time == 0) return STATUS_532_ERROR;
    }
    return STATUS_532_OK;
}
```

Listing 1.3: Waiting for PN532 to Become Ready.

This function continuously checks if the PN532 is ready using `isReadyToSend()`, waiting up to a specified timeout in milliseconds. It delays for 1 ms between checks. If the device becomes ready in time, it returns `STATUS_532_OK`; otherwise, it returns `STATUS_532_ERROR`. It is used by higher-level functions such as `sendCommand()` and `readResponseToCommand()` to ensure that the PN532 is ready before further communication.

3. Validating Acknowledgment from PN532: `readACK()`

```c
/*
 * readACK:
 * Reads the 6-byte ACK frame from the PN532 and checks if
 * it matches the expected ACK values.
 * It is used within the sendCommand()
 * The expected ACK frame is 00 00 FF 00 FF 00
 *
 */
static StatusCode532_t readACK(void) {
    uint8_t ACK_buffer[6];
    SS_LOW();
    // send PN532 the DATA READ SPI byte indicating the SPI READ operation:
    SPI_write(&SPI_HANDLER, PN532_SPI_DATAREAD);
    // read 6-byte ACK frame:
    for (int i = 0; i < 6; i++)
        ACK_buffer[i] = SPI_read(&SPI_HANDLER);

    SS_HIGH();
    // compare the received response to ACK frame
    for (int i=0; i<6; i++) {
      if (ACK_buffer[i] != ACK_frame[i]) return STATUS_532_INVALID_ACK;
    }
    return STATUS_532_OK;
}
```

Listing 1.4: Validating Acknowledgment from PN532.

This function reads a 6-byte ACK (acknowledgment) frame from the PN532 and checks if it matches the expected fixed pattern `00 00 FF 00 FF 00`. If the frame matches, it returns `STATUS_532_OK`, indicating that the PN532 has correctly received and acknowledged the last command. Otherwise, it returns `STATUS_532_INVALID_ACK`. This check is essential for validating that the PN532 correctly processed the sent command.

4. Sending a Command Frame over SPI: `sendFrame(uint8_t *cmd, uint8_t cmd_length)`

```c
/*
 * sendFrame:
 * Constructs and sends a frame over SPI, including SPI operation,
 * preamble, start codes, length, checksums, TFI, and postamble.
 * It is a helper function udsed within sendCommand()
 *
 * Normal frame:
 * 00 00 FF LEN LCS TFI PD0 ... PDn DCS 00
 */
```

```
static void sendFrame(uint8_t *cmd, uint8_t cmd_length) {
    SS_LOW();

    SPI_write(&SPI_HANDLER, PN532_SPI_DATAWRITE);
    SPI_write(&SPI_HANDLER, PN532_PREAMBLE);
    SPI_write(&SPI_HANDLER, PN532_STARTCODE1);
    SPI_write(&SPI_HANDLER, PN532_STARTCODE2);

    cmd_length++;  // length of data field: TFI + DATA
    SPI_write(&SPI_HANDLER, cmd_length);
    SPI_write(&SPI_HANDLER, (~cmd_length + 1));
    SPI_write(&SPI_HANDLER, PN532_HOSTTOPN532);

    // write data and accumulate data checksum
    uint8_t DCS = PN532_HOSTTOPN532;
    for (uint8_t i = 0; i < cmd_length - 1; i++) {
        SPI_write(&SPI_HANDLER, cmd[i]);
        DCS += cmd[i];
    }

    SPI_write(&SPI_HANDLER, (~DCS + 1));
    SPI_write(&SPI_HANDLER, PN532_POSTAMBLE);

    HAL_Delay(1);
    SS_HIGH();
}
```

Listing 1.5: Sending a Command Frame over SPI.

This function constructs and sends a well-formed PN532 frame over SPI. The frame includes a preamble, start codes, length, length checksum (LCS), frame identifier (TFI), payload data, data checksum (DCS), and postamble. It also includes the SPI-specific data write command. The checksum fields ensure data integrity. This function is used internally by `sendCommand()`.

5. Complete Command Transmission with Acknowledgment: `sendCommand(uint8_t *cmd, uint8_t cmd_length)`

```
/*
 * sendCommand:
 * Sends a command to the PN532, waits for PN532 to be
 * ready and checks for a proper ACK response.
 *
 * @return: STATUS_532_OK if ready and ACK received
 */
static StatusCode532_t sendCommand(uint8_t *cmd, uint8_t cmd_length) {
  int status;

    sendFrame(cmd, cmd_length);
    status = waitToBeReady(PN532_ACK_WAIT_TIME);
    if (status != STATUS_532_OK) return STATUS_532_NOTREADY;
    StatusCode532_t ack = readACK();
    if (ack != STATUS_532_OK) return STATUS_532_INVALID_ACK;
    return STATUS_532_OK;
}
```

Listing 1.6: Complete Command Transmission with Acknowledgment.

This high-level helper function sends a command to the PN532 by first using `sendFrame()`, then waiting for the device to become ready using `waitToBeReady()`, and finally validating the ACK frame using `readACK()`. If all steps succeed, it returns `STATUS_532_OK`. Otherwise, it returns one of several error codes, such as `STATUS_532_NOTREADY` or `STATUS_532_INVALID_ACK`.

6. Receiving and Validating Response: `readResponseToCommand()`

```
/**
```

```c
 * readResponseToCommand:
 * Reads a full response frame from the PN532 to a command,
 * checks headers, length, command ID, and validates checksum.
 *
 * Return: number of bytes (>0) received if success, error code otherwise
 */
static int16_t readResponseToCommand(
            uint8_t command,
            uint8_t *data_buffer,
            uint8_t data_length,
            uint16_t wait_time
           ) {


    if (waitToBeReady(wait_time) != STATUS_532_OK)
        return STATUS_532_TIMEOUT;

    SS_LOW();

    SPI_write(&SPI_HANDLER, PN532_SPI_DATAREAD);

    // read 1st to 3rd bytes and check if this is a valid frame (preamble + start codes)
    if (SPI_read(&SPI_HANDLER) != PN532_PREAMBLE   ||
        SPI_read(&SPI_HANDLER) != PN532_STARTCODE1 ||
        SPI_read(&SPI_HANDLER) != PN532_STARTCODE2
        ) {
        SS_HIGH();
        return STATUS_532_INVALID_FRAME;
    }

    /* read 4th and 5th bytes */
    uint8_t LEN = SPI_read(&SPI_HANDLER);
    if (LEN == 0) {
      SS_HIGH();
      return STATUS_532_NODATA;
    }
    uint8_t LCS = SPI_read(&SPI_HANDLER);
    if ((uint8_t)(LEN + LCS) != 0x00 ) {
        SS_HIGH();
        return STATUS_532_INVALID_LCS;
    }

    /* read 6th and 7th bytes */
    uint8_t PD0 = command + 1;
    if (PN532_PN532TOHOST != SPI_read(&SPI_HANDLER) || PD0 != SPI_read(&SPI_HANDLER)) {
        SS_HIGH();
        return STATUS_532_INVALID_FRAME;
    }

    /* check data buffer size before read actual data */

    LEN -= 2;  // subtract TFI and cmd from DATA length
    if (LEN > data_length) {   // if no enough space, just dummy read bytes
        for (uint8_t i = 0; i < LEN; i++) SPI_read(&SPI_HANDLER);
        SPI_read(&SPI_HANDLER);
        SPI_read(&SPI_HANDLER);
        SS_HIGH();
        return STATUS_532_NO_SPACE;
    }

    /* read actual data */
    uint8_t SUM = PN532_PN532TOHOST + PD0;
    for (uint8_t i = 0; i < LEN; i++) {
        data_buffer[i] = SPI_read(&SPI_HANDLER);
        SUM += data_buffer[i];
    }

    /* read data checksum byte */
    uint8_t DCS = SPI_read(&SPI_HANDLER);
    if ((uint8_t)(SUM + DCS) != 0) {
        SS_HIGH();
        return STATUS_532_INVALID_DCS;

    /* read POSTAMBLE */
    SPI_read(&SPI_HANDLER);

    SS_HIGH();
    return (int16_t)LEN; // number of bytes (LEN>0) received
}
```

Listing 1.7: Receiving and Validating Response.

This function reads a complete response frame from the PN532 after a command has been sent. It checks the preamble, start codes, length and its checksum, and validates that the frame originates from the PN532. It also checks the command response code and verifies the data checksum. If all checks pass and there is sufficient space in the user-provided buffer, the function copies the payload data and returns the number of bytes received. If any check fails or the buffer is too small, the function returns an appropriate error code.

Each of these functions encapsulates a specific and critical part of the communication protocol with the PN532—such as waiting for readiness, sending command frames, receiving acknowledgments, and reading responses—while handling the low-level SPI details. They are designed to be used as internal building blocks for the higher-level exported functions, which will be implemented in the following subsection. These exported functions will issue specific PN532 commands and handle the corresponding responses, relying on the internal helpers to ensure correctness, robustness, and protocol compliance.

1.2.5.3 Exported PN532 Command Functions

This subsubsection presents the exported functions that perform key operations with the PN532 NFC controller. Each function encapsulates the sending of a specific PN532 command and the reception and interpretation of its response. These functions build upon the previously introduced helper routines and represent the public API for interacting with the PN532 module.

1. `PN532_SPI_Init`: Waking Up the PN532 Module

```c
/*
 * Wake-up PN532 module
 * The Adafruit PN532 module uses power-saving modes
 *     and the chip starts in Low Power or HSU (High-Speed UART) mode
 * We're not getting any response until wake-up
 * A common trick is to reset SS for a few ms (and maybe also RST pin)
 */
void PN532_SPI_Init(void) {

    /*-- Wake Up PN532 --*/
    SS_LOW();
    HAL_Delay(5);
    SS_HIGH();
}
```

Listing 1.8: Waking Up the PN532 Module.

The `PN532_SPI_Init` function initiates communication by performing a wake-up sequence for the PN532, which may start in a low-power or High-Speed UART (HSU) mode. This step ensures the PN532 enters SPI mode and is ready to receive commands. The wake-up sequence typically involves briefly toggling the SPI Slave Select (SS) line. Without this initialization, further communication via SPI is not possible.

2. `PN532_getFirmwareVersion`: Reading PN532 Firmware Information

```c
 /*
 * PN532_getFirmwareVersion
 *  Retrieves the 4-byte firmware version from the PN532.
 *
 * @return: 32-bit firmware version number for PN532 or STATUS_532_ERROR for an error.
 *
 * Version:
 * 1st byte: Version of the IC. For PN532, the contain of this byte is 0x32
 * 2nd byte: Version of the firmware.
```

```
  * 3rd byte: Revision of the firmware.
  * 4th byte: 3 LS bits indicate which are the functionalities supported.
  *
  * In the case of the PN532: 0x32010607
  *
  */
uint32_t PN532_getFirmwareVersion(void) {
    packet_buffer[0] = PN532_COMMAND_GETFIRMWAREVERSION;
    StatusCode532_t status;


    status = sendCommand(packet_buffer, 1);
    if (status != STATUS_532_OK) return STATUS_532_ERROR;
    status = readResponseToCommand(PN532_COMMAND_GETFIRMWAREVERSION,
                                   packet_buffer, 12, PN532_WAITTIME);
    if (status < 0) return STATUS_532_ERROR;

    // store the result into an unsigned 32 bit integer
    uint32_t response;
    response = packet_buffer[0];
    response <<= 8;
    response |= packet_buffer[1];
    response <<= 8;
    response |= packet_buffer[2];
    response <<= 8;
    response |= packet_buffer[3];

    return response;
}
```

Listing 1.9: Reading PN532 Firmware Information.

The `PN532_getFirmwareVersion` function sends the `GETFIRMWAREVERSION` command to retrieve a 4-byte response from the PN532. This response includes the chip type, firmware version, firmware revision, and supported protocol features. It returns the data as a 32-bit integer or an error code if communication fails. This function is useful for identifying the chip and verifying communication.

3. `PN532_SAMConfiguration`: Setting Normal Operating Mode and Enabling RF

```
/*
  * PN532_SAMConfiguration
  *
  * The Security Access Module (SAM) Configuration command configures how the PN532 operates.
  * It sets the PN532 to normal operation mode and initializes the RF interface.
  * Without it, the chip may not behave as expected - it may not respond to
  * InListPassiveTarget properly,
  * or you might not receive tag UIDs.
  *
  * If SAMConfiguration succeed, PN532 returns respond to a command without any data
  *
  *
  * @return STATUS_532_OK if success, STATUS_532_ERROR if error.
  *
  *
  */
StatusCode532_t PN532_SAMConfiguration(void) {
    /*-- prepare command --*/
    packet_buffer[0] = PN532_COMMAND_SAMCONFIGURATION;
    packet_buffer[1] = 0x01; // normal operation mode
    packet_buffer[2] = 0x14; // timeout=1000 ms
    packet_buffer[3] = 0x01; // use IRQ pin!

    /*-- write command and read response --*/
    if (sendCommand(packet_buffer, 4) != STATUS_532_OK) return STATUS_532_ERROR;
    if (readResponseToCommand(PN532_COMMAND_SAMCONFIGURATION, packet_buffer,
                              sizeof(packet_buffer), PN532_WAITTIME) < 0) return STATUS_532_ERROR;

    return  STATUS_532_OK;
}
```

Listing 1.10: Setting Normal Operating Mode and Enabling RF.

The `PN532_SAMConfiguration` function sends the `SAMCONFIGURATION` command to configure the internal Security Access Module (SAM) settings of the PN532. It switches the chip into normal mode, sets a timeout for detecting passive targets, and enables the use of the IRQ pin. This configuration is essential for successful operation in tag detection and peer-to-peer communication scenarios.

4. `InListPassiveTarget`: Detecting Tags and Retrieving UID

```
/*
  * InListPassiveTarget
  * Sends a command to detect ISO14443A targets. If found, returns the card's UID and its length.
  *
  * @args:
  *    uid            Pointer to the array that will be populated with the card's UID (up to 7 bytes).
  *    uidLength      Pointer to the variable that will hold the length of the card's UID.
  *
  * @return STATUS_532_OK if success, STATUS_532_ERROR for an error.
  *
  */
StatusCode532_t InListPassiveTarget (uint8_t *uid, uint8_t *uid_length) {
    /*-- prepare command --*/
    packet_buffer[0] = PN532_COMMAND_INLISTPASSIVETARGET;
    packet_buffer[1] = 0x01; // (MaxTg) - max 1 card
    packet_buffer[2] = 0x00; // (BrTy) - 0x00: 106 kbps type A (ISO/IEC14443 Type A),

    /*-- write command and read response --*/
    if (sendCommand(packet_buffer, 3) != STATUS_532_OK) return STATUS_532_ERROR;
    if (readResponseToCommand(PN532_COMMAND_INLISTPASSIVETARGET,packet_buffer,
                              sizeof(packet_buffer), PN532_WAITTIME) < 0) return STATUS_532_ERROR;

    /*
     *          ISO14443A Card Response Format
     *  -------------------------------------
     *   byte          |   Description
     *  -------------  |  -------------------
     *   b0            |   NbTg: Number of targets found (usually 1)
     *   b1            |   Tg
     *   b2..3         |   SENS_RES
     *   b4            |   SEL_RES
     *   b5            |   NFCID Length
     *   b6..b9        |   NFCID
     */

    /*-- authenticate and save data --*/
    /* byte 0 */
    if (packet_buffer[0] != 1)
      return STATUS_532_NOTAG;  // no tags found

    /* byte 5 */
    *uid_length = packet_buffer[5]; // save uid length

    /* UID */
    for (uint8_t i = 0; i < packet_buffer[5]; i++)
        uid[i] = packet_buffer[6 + i];  // save uid bytes

    return STATUS_532_OK;
}
```

Listing 1.11: Detecting Tags and Retrieving UID.

The `InListPassiveTarget` function issues a command to detect ISO14443A-compliant passive targets (such as NFC or MIFARE cards) and reads their unique identifier (UID). The function interprets the structured response to confirm the presence of a target and then extracts the UID and its length. On success, the UID is copied to a user-provided buffer, and a success status is returned; otherwise, an error or "no tag found" status is indicated.

1.2.6 PN532 Header File: `PN532.h`

To support communication with the PN532 NFC controller, a dedicated header file `PN532.h` is provided. This header includes all necessary constant definitions, type declarations, and function prototypes required to interact with the PN532 using the exported API.

```c
#ifndef SRC_PN532_H_
#define SRC_PN532_H_

#include "stm32h7xx_hal.h"

extern SPI_HandleTypeDef hspi2;
#define SPI_HANDLER hspi2
#define PN532_SS_GPIO_Port  GPIOB
#define PN532_SS_Pin    GPIO_PIN_4

#define MAX_FRAME_LENGTH          (255)

/* Wait Time */
#define PN532_WAITTIME                        (1000)
#define PN532_ACK_WAIT_TIME                   (200)

/*********************************************************
 * Define names for constant frame bytes (e.g. preamble etc.)
 *********************************************************/
#define PN532_PREAMBLE                        (0x00)
#define PN532_STARTCODE1                      (0x00)
#define PN532_STARTCODE2                      (0xFF)
#define PN532_POSTAMBLE                       (0x00)
#define PN532_HOSTTOPN532                     (0xD4)
#define PN532_PN532TOHOST                     (0xD5)

/*********************************************************
 * Define SPI operations used in PN532
 *********************************************************/
#define PN532_SPI_STATREAD                    (0x02)
#define PN532_SPI_DATAWRITE                   (0x01)
#define PN532_SPI_DATAREAD                    (0x03)

/*********************************************************
 * Define used PN532 commands
 *********************************************************/

#define PN532_COMMAND_GETFIRMWAREVERSION    (0x02)
#define PN532_COMMAND_SAMCONFIGURATION      (0x14)
#define PN532_COMMAND_INLISTPASSIVETARGET   (0x4A)

/* Status ready: */
#define PN532_SPI_READY                       (0x01)

/*********************************************************
 * Define a type for status returned from PN532 functions
 * Pa3cio, UL FRI, 2025
 *********************************************************/
typedef enum StatusCode532 {
  STATUS_532_OK                 =  0,
  STATUS_532_ERROR              = -1,
  STATUS_532_INVALID_ACK        = -2,
  STATUS_532_TIMEOUT            = -3,
  STATUS_532_INVALID_FRAME      = -4,
  STATUS_532_INVALID_LCS        = -5,
  STATUS_532_NO_SPACE           = -6,
  STATUS_532_INVALID_DCS        = -7,
  STATUS_532_NOTREADY           = -8,
  STATUS_532_NODATA             = -9,
  STATUS_532_NOTAG              = -10
} StatusCode532_t;

/*********************************************************
 * Exported PN532 functions
 *********************************************************/
void PN532_SPI_Init(void);
uint32_t PN532_getFirmwareVersion(void);
StatusCode532_t InListPassiveTarget (uint8_t *uid, uint8_t *uid_length);
StatusCode532_t PN532_SAMConfiguration(void);

#endif /* SRC_PN532_H_ */
```

Listing 1.12: PN532 Header File.

The header begins by including the HAL library to access STM32 hardware abstraction features. It defines symbolic names for the SPI peripheral and GPIO pins used for PN532 communication, as well as several useful constants such as wait times, frame formatting bytes, and SPI operation codes. It also lists all command codes used with the PN532 controller. An enumeration type `StatusCode532_t` is defined to standardize the interpretation of return values from PN532-related functions. These status codes help indicate successful execution or various error conditions during communication. Finally, the header declares all functions provided by the PN532 module, including initialization, configuration, and basic tag reading functions. These API calls are implemented in the corresponding `PN532.c` source file and are intended to be called from user applications such as `main.c`.

1.2.7 Using the PN532 Exported API Functions

This subsection demonstrates how to use the exported PN532 API functions in a typical embedded application. The following outline shows the essential steps for initializing the PN532 module, configuring it for normal operation, and continuously checking for the presence of RFID targets (tags). Peripheral initialization routines such as SPI, GPIO, and system clocks are assumed to be handled earlier in the program. Listing 1.13 shows only a relevant portion of the `main.c` file, omitting the initialization of peripherals such as SPI, GPIO, and system clocks.

```c
uint8_t uid[] = { 0, 0, 0, 0, 0, 0, 0 };  // Buffer to store the returned UID
uint8_t uidLength;                        // Length of the UID
uint32_t firmwareVersion = 0;

int main(void)
{

  /*******************************************************************************************
   * 1. Wake-up PN532 and check if it is available by getting its firmware version
   *******************************************************************************************/

  PN532_SPI_Init();
  while(1) {
    firmwareVersion = PN532_getFirmwareVersion();
    if (firmwareVersion != STATUS_532_ERROR) {
      break;
    }
    HAL_Delay(250);
  }

  /*******************************************************************************************
   * 2. Configure SAM: set normal operation mode and initialize the RF interface
   *******************************************************************************************/

  while (PN532_SAMConfiguration() != STATUS_532_OK){
    HAL_Delay(100);
  }

  /*******************************************************************************************
   * 3. Every second check if there is a RFID target present and read the UID
   *******************************************************************************************/

  while (1)
  {
    statusCode = InListPassiveTarget(uid, &uidLength);
    if (statusCode == STATUS_532_OK) {

      // do dsomething with UID
    }
    HAL_Delay(1000);
  }
}
```

Listing 1.13: Using the PN532 Exported API in `main.c` to Detect and Read RFID Tags.

1. **Wake-up and Firmware Version Check** (`PN532_SPI_Init`, `PN532_getFirmwareVersion`)
   Upon power-up, the PN532 module typically starts in a low-power or high-speed UART mode, and will not respond over SPI until it is awakened. The `PN532_SPI_Init()` function performs a simple wake-up sequence by toggling the Slave Select (SS) pin. The firmware version is then requested in a loop using `PN532_getFirmwareVersion()` until a valid response is received. This ensures that the PN532 is awake and communicating properly before proceeding.
2. **SAM Configuration** (`PN532_SAMConfiguration`)
   Before interacting with RFID cards, the PN532 must be placed into normal operating mode and the RF interface must be initialized. This is done using `PN532_SAMConfiguration()`, which sends the appropriate configuration command. A loop ensures that the configuration is repeated until it succeeds, accounting for any potential timing or communication issues.
3. **Reading RFID Tags** (`InListPassiveTarget`)
   After initialization, the program enters an infinite loop where it polls for nearby ISO14443A-compliant RFID targets (e.g., NFC cards or tags) once per second. The `InListPassiveTarget()` function performs this task and, if a card is detected, it populates the UID and its length into provided variables. At this point, application-specific logic (e.g., comparing UIDs or logging events) can be implemented to process the detected tag.

This simple usage scenario illustrates the practical integration of the PN532 driver functions in an embedded project. It provides a minimal but functional RFID polling application that can serve as a base for more advanced NFC/RFID features.