Using SPI to Drive an MFRC522 RFID Reader

Spencer Leslie, Louie Lu, Jet Simon

April 2024

Contents

1 Introduction

This guide will walk you through how to use SPI to drive an MFRC522 (also sometimes labelled as RC522) RFID Reader. Section 2 gives a brief overview on the SPI protocol, and shows how to configure SPI on the BeagleBone. Section 3 explains the technologies behind RFID, and demonstrates the steps necessary to obtain the UID (unique ID) of an RFID tag. Please know that while this is a lengthy guide, we have designed the article to be beginner-friendly. Support files for SPI and RFID modules are included alongside this guide, but their implementations are only half-complete.

We wrote this two-in-one guide because we found that other student guides for SPI, while helpful, contain some crucial inaccuracies as of the time of writing. Section 2 of our guide can be considered an update to those documents. As for RFID, beginner resources are very scarce. Most Internet guides hand-wave away the technical details, simply encouraging direct use of popular premade libraries which are not immediately compatible with the BeagleBone. We hope that by reading this guide, you can not only get your RFID reader to work for this class's hardware, but also gain an appreciation for what is going on under the hood.

2 SPI

2.1 Overview

To drive the MFRC522 RFID reader, we will be using a protocol we have not yet seen in this class - SPI (Serial Peripheral Interface). While it might be possible to use I2C or UART to drive the MFRC522, we will not be covering them here. Based on our findings, the MFRC522 is primarily intended to be used with SPI.

It is useful to frame our introduction of SPI in terms of an already familiar protocol: I2C. Like I2C, SPI employs a synchronous, master-slave architecture. The master is the BeagleBone, and the slave is an SPI device, i.e. our RFID reader. The main difference between the two protocols is that I2C uses just one signal line to exchange data - SDA (Serial Data), while SPI uses two - MOSI (Master Out Slave In) and MISO (Master In Slave Out).

Practically speaking, there is just one important quirk you must know about SPI: there is a **one-byte delay** between the two signal lines, MOSI and MISO. This one-byte delay is responsible for some slightly unintuitive code, which we will see in a later section. For now, this concludes the information we need to know about the protocol itself. The next section shows you how to set up SPI on the BeagleBone.

2.2 Configuring SPI on the BeagleBone

There are two SPI buses on the BeagleBone, labeled SPI0 and SPI1. Below is a table showing the two buses and their pin mappings. Note that you have two options for CS (Chip Select) when using SPI1.

Please note that if you elect to use SPI0, you will **cannibalize** the I2C1 bus, which controls the 14-seg display and the accelerometer. If you elect to use SPI1, you will cannibalize HDMI video and audio capabilities. Consult the P9 Header table for more information. For this guide, we will proceed with SPI1, CS0.

Table 1: SPI Buses on the BeagleBone

Bus	**Linux Device Filepath**	**Pins**
SPI0	/dev/spidev0.0	CS: P9_17 (SPI0_CS0) SCLK: P9_22 (SPI0_SCLK) MOSI: P9_18 (SPI0_D1) MISO: P9_22 (SPI0_D0)
SPI1, CS0	/dev/spidev1.0	CS: P9_28 (SPI1_CS0) SCLK: P9_31 (SPI1_SCLK) MOSI: P9_30 (SPI1_D1) MISO: P9_29 (SPI1_D0)
SPI1, CS1	/dev/spidev1.1	CS: P9_42 (SPI1_CS1)

Regardless of which bus you choose, SPI is disabled by default on the BeagleBone. We must take the following steps to enable SPI:

1. Explicitly load the custom SPI cape in order for the kernel to recognize that hardware:

   - Check that your SPI device tree overlay exists at /lib/firmware. They should be there by default.

     ```
     (bbg)$ ls | grep -r "SPIDEV" /lib/firmware
     ```

     grep: /lib/firmware/BB-SPIDEV1-00A0.dtbo: binary file matches # use for SPI1
     grep: /lib/firmware/BB-SPIDEV0-00A0.dtbo: binary file matches # use for SPI0

   - Enable the SPI device tree overlay through U-Boot.

     ```
     (bbg)$ sudo nano /boot/uEnv.txt
     ```

     Look for the header:

     ###Additional custom capes

     Then add the following line:

     uboot_overlay_addr5=/lib/firmware/BB-SPIDEV1-00A0.dtbo

   - Disable any capes that would compete with your selected SPI cape. For SPI1, that means disabling the HDMI video and audio capes.

     Look for the header:

     ###Disable auto loading of virtual capes (emmc/video/wireless/adc)

     Then uncomment the following lines:

     disable_uboot_overlay_video=1
     disable_uboot_overlay_audio=1

2. (Optional) By default, the SPIDEV files are owned by root. Normally, you would have to use sudo to run a program that uses SPI. To bypass this, you can use this simple quick fix:

- Create a udev rule that assigns a user group to the SPIDEV file.

  ```
  (bbg)$ sudo nano /etc/udev/rules.d
  ```

  Change the file to contain exactly the following:

  KERNEL==“spidev*”, GROUP=“spiusr”, MODE=“0660”

- Add the spiusr group.

  ```
  (bbg)$ sudo groupadd spiusr
  ```

- Add the debian user to the spiusr group.

  ```
  (bbg)$ sudo gpasswd -a debian spiusr
  ```

3. Reboot for the changes to take effect, then SSH back in.

   ```
   (bbg)$ sudo reboot
   ```

4. Confirm the changes.

   - Verify that the spidev driver was loaded.

     ```
     (bbg)$ ls -al /dev/spidev1.0
     ```

     crw-rw---- 1 root spiusr 153, 2 Mar 11 23:43 /dev/spidev1.0

   - Confirm the pins have been configured for SPI. Note that we should not have to run any config-pins commands ourselves; loading the device tree overlay automatically configures them for us.

     ```
     (bbg)$ show-pins
     ```

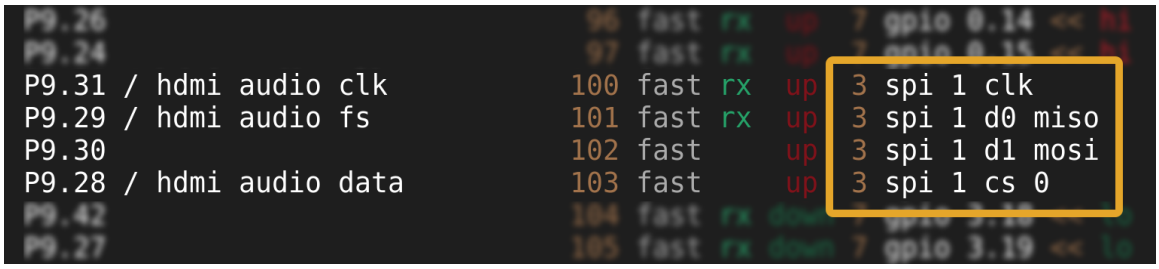

Figure 1: *show-pins* confirms that SPI1 is ready to go

2.3 Working with SPI in C

Linux has its own SPI module <`linux/spi/spidev.h`> which abstracts away many of the finer details for us. We will take full advantage of this module to drive our SPI device. Below, we show the code to initialize, and then communicate over, SPI. You may notice that this code is nearly identical to working with I2C.

Error checking has been omitted for brevity.

2.3.1 Initialization

To initialize a SPI bus, first open the bus via its filepath (see Table 1), and obtain its file descriptor.

```
#include <linux/spi/spidev.h>
#define SPI_DEV_BUS1_CS0 "/dev/spidev1.0"

int spiFileDesc = open(SPI_DEV_BUS1_CS0, O_RDWR);
```

Then, set the SPI mode to DEFAULT (0) using `ioctl`. ioctl is a Linux system call which handles device-specific file I/O. We use ioctl in conjunction with our file descriptor to perform actions on our SPI device.

```
#define SPI_MODE_DEFAULT 0

int spiMode = SPI_MODE_DEFAULT;
ioctl(spiFileDesc, SPI_IOC_WR_MODE, &spiMode);
```

Lastly, set the max speed of the SPI device. Ideally, this should match the SPI device you are working with. The below value worked for our RFID application, so we recommend sticking with it.

```
#define SPEED_HZ_DEFAULT 4000000

int speedHz = SPEED_HZ_DEFAULT;
ioctl(spiFileDesc, SPI_IOC_WR_MODE, &speedHz);
```

2.3.2 Reading and Writing Registers

Much like I2C, working with SPI is a matter of reading and writing registers of the slave device. To read a register, we must first create an SPI transfer struct, which is defined in the Linux SPI module. Ensure that this struct does not contain any unexpected values by immediately memsetting its values to 0.

```
struct spi_ioc_transfer spiTransaction;
memset(&spiTransaction, 0, sizeof(struct spi_ioc_transfer));
```

This struct has three fields we are concerned with: tx_buf, rx_buf, and len.

tx_buf is the transmit (i.e. send) buffer, and uses the MOSI (Master Out Slave In) line. We fill this buffer with what we want to communicate to the SPI device. At a minimum, tx_buf should contain the register address of the SPI device that you want to access. If you are writing a register, you should also fill tx_buf with the actual value you want to write to the register.

Conversely, rx_buf is the receive buffer, and uses the MISO (Master In Slave Out) line. After the SPI transaction is complete, rx_buf will contain the SPI device's response. This generally means that rx_buf will contain the register's content. Please note that you only need to fill rx_buf if you are reading a register.

The following sample code shows how to fill an SPI transfer struct if intending to read a register.

```
#define SPI_BUFF_SIZE 2

// Create buffers to pass in to the SPI struct.
myTxBuf[SPI_BUFF_SIZE] = { regAddrToRead, 0 };
myRxBuf[SPI_BUFF_SIZE] = { 0, 0 };                  // arbitrary init

// Set the 3 SPI struct fields, typecasting the buffers to match Linux's struct spec.
spiTransaction.tx_buf = (unsigned long)myTxBuf;
spiTransaction.rx_buf = (unsigned long)myRxBuf;
spiTransaction.len = SPI_BUFF_SIZE;
```

After filling the struct with our buffers, we call upon the trusty ioctl to perform the data transfer. We pass in the SPI file descriptor, the filled SPI transfer struct, as well as a macro defined in the Linux SPI module (`SPI_IOC_MESSAGE(1)`) that specifies to initiate the transaction. The 1 that we pass into the macro defines how many transfers to perform. While it is possible to perform multiple transfers in just one ioctl call, it is simpler to keep each register I/O operation as a separate call.

```
ioctl(spiFileDesc, SPI_IOC_MESSAGE(1), &spiTransaction);
```

Upon completion, rxBuf should contain the content of the register address we accessed. However, printing rxBuf[0] reveals a result of 0. What's going on? Did the transaction not occur?

Recall the one quirk we need to know about SPI: that MOSI and MISO share a one-byte delay. We stored `regAddrToRead` in myTxBuf[**0**], which uses the MOSI line. Therefore, myRxBuf, which uses the MISO line, stores the result in myRxBuf[**1**]. This also explains why we define our buffer size to be 2, instead of 1: we need that one extra space to store the result.

```
return rxBuf[0];     // returns a garbage value
return rxBuf[1];     // returns the register's content
```

2.4 Support Files

In the support files accompanying this guide, you will find a half-complete SPI implementation in `spi.h` and `spi.c`. What you have seen in this guide - initialization and reading a register - is filled in for you. However, the code to write a register is missing. We encourage you to pause this guide now, work through the module, and implement the missing function. A working SPI implementation will be required for the RFID section.

Hint: Writing a register is very similar to reading a register, but carefully consider which of the 3 SPI transfer struct fields you actually need to fill!

2.5 Troubleshooting

- If you try to run a program that uses SPI and you receive an error similar to:

  ```
  (bbg)$ Spi_init open:  No such file or directory
  ```

then this means that the SPI device tree overlay has not been loaded. First, run through the steps in Section 2.2 again carefully. If the same error persists, and you have included `BB-BONE-AUDI-02-00A0.dtbo` as a part of doing the Beat-Box assignment, you can try commenting that out and rebooting. We had conflicting reports on whether that particular audio overlay competes with SPI or not. Additionally, ensure that you have not tampered with other options in uEnv.txt. For example, enable_uboot_cape_universal=1 should be uncommented by default.

- If you have successfully loaded the SPI device tree overlay but show-pins is not listing the pins as configured for SPI, you can try manually configuring the pins:

```
(bbg)$ sudo config-pin p9.28 spi_cs
(bbg)$ sudo config-pin p9.29 spi
(bbg)$ sudo config-pin p9.30 spi
(bbg)$ sudo config-pin p9.31 spi_sclk
```

- If you are receiving permission errors, remember that running a program that uses SPI requires sudo by default. Follow the optional step in Section 2.2 if you wish to obviate this requirement.

3 RFID

3.1 Overview

RFID (radio-frequency identification) consists of two parts: a reader, and a tag. In official documentation, these parts are more commonly referred to as the PCD (proximity coupling device), and the PICC (proximity integrated circuit card). In this guide, we will proceed with the more familiar terms, reader and tag. The particular models we are working with are the MFRC522 reader, and the MF1S503x (also known as MIFARE CLASSIC 1K) tag. The standard kit comes with two tags: a white card, and a blue key fob. Despite their difference in shape and size, these are both MIFARE CLASSIC 1K tags.

The point of RFID is simple: to transmit data between the reader and the tag.

Data exchange between a reader and a tag occurs through radio waves. The reader initiates communication by broadcasting a radio wave which carries a **command**. If a tag is in range of the radio wave, the tag is powered up. Based on the command it receives, the tag then sends back its own radio wave containing some data. Most commonly, the reader commands a tag to return its **UID** (unique ID). The UID can be checked against a list of UIDs to permit or deny entrance to an apartment building, for example.

Noise is crucial to consider when working with RFID. There are many factors that can lead to an incorrect exchange of data. For example, holding a tag too briefly next to the reader will very often result in an incomplete transmission. External electromagnetic interference can corrupt the data. The clock timings of the devices could be mismatched. With so many things that could go wrong, we will need to implement ways to validate the data we receive. We stress that error checking with RFID is **not** an optional step.

In summary, as the programmer, our primary jobs when working with RFID are to:

(a) specify which commands the reader should send to the tag, and when to send them, and

(b) validate the responses from the tag.

Let us take a closer look at the two individual RFID parts.

3.2 MF1S503x, aka the MIFARE Classic 1K

3.2.1 The UID

The tag stores 1024 bytes of memory (hence "1K" in the name MIFARE Classic 1K), organized into blocks. It is possible to read and write any of these blocks with any data you wish, but doing so requires extra steps - including MIFARE Authentication - and is beyond the scope of this guide. We are just here for the UID. To obtain a tag's UID, we merely require the special read-only block known as the **Manufacturer Block**. This is the only block in memory that does not require elevated MIFARE Authentication to access.

The UID is 4 bytes long. It was "hard-coded" during production, and cannot be overwritten. While not *actually* guaranteed to be unique, this UID is, probabilistically speaking, *extremely* likely to be sufficient to identify tags from one another. Because of this technicality, you may see the UID described in official documentation as the NUID (non-unique ID). Do not be confused: UID and NUID are the same thing.

The byte that comes immediately after the UID in memory is the BCC (**Block Check Character**). This 5th byte was hard-coded during production to be equal to the exclusive OR over the UID's 4 bytes. Whenever

we send out a command to retrieve the UID, the BCC always comes along for the ride, so we receive 5 bytes in total. As part of our data validation policy, we will use the BCC to confirm whether or not a transmitted (read: potentially noisy!) UID matches the actual, intended UID.

3.2.2 Commands

As a reminder, the RFID reader broadcasts a command to initialize communication with a tag. If a tag is in range, it accepts the command and responds to the reader accordingly. To relinquish its UID, a tag must receive the following two commands from the reader, in order, one after the other:

Table 2: Required Commands to Obtain a UID

Reader sends...	**Command Code**	**Meaning**	**Tag responds with...**
1. REQA	0x26 (7-bit)*	search for a tag	ATQA (0x04 0x00)
2. ANTICOLLISION	0x93 0x20	retrieve UID of a found tag	UID + BCC

There are a few things worth pointing out from this table.

(a) REQA stands for Request A. This command's purpose is to probe for, and select, a single tag. The ANTICOLLISION command has a distinctly irrelevant name in relation to its function. Do not be confused: ANTICOLLISION means nothing more than "GIVE ME YOUR UID".

(b) The term "command" may have been a little abstract up until now, but we see now that a command is nothing more than a particular byte (or two). There are some peculiarities, however. Note that REQA's command code is mysteriously labelled 7-bit. The ANTICOLLISION command code has no such label, but is comprised of two bytes instead of one. Just be aware of these oddities for now; we will revisit how to deal with them when we start writing the code to send out these commands.

(c) ATQA stands for Answer to Request A. It is simply an arbitrary 2-byte code that the manufacturers decided the tag should return when receiving a REQA command. It should always consist of the two bytes 0x04 and 0x00. Just like with the UID and BCC, we should check, as part of our data validation policy, that we receive this precise sequence of bytes after sending out a REQA command.

3.3 MFRC522

From the previous section, we learned *which* commands to send to a tag, and *when* to send them, in order to obtain its UID. Now we will see *how* to send these commands from the MFRC522 reader to the tag.

3.3.1 The FIFO Buffer

The MFRC522 has a 64-byte, bi-directional buffer known as the FIFO Buffer, which is used to communicate between the reader and a tag. The purpose of this buffer is to *transceive*, which is a portmanteau of *trans*mit (i.e. send), and re*ceive*. The FIFO Buffer first transmits a command code, then receives a tag's response. The FIFO Buffer does not constantly transceive. It only transceives a single time when we instruct it to.

Note that we never interface with the tag directly. Anything we, the programmer, wish to do with the tag, we must do through the reader's FIFO Buffer. The procedure we follow is:

(a) lodge a command code into the FIFO Buffer,

(b) demand the reader to transceive (which sends our command code and receives a response),

(c) retrieve (and validate!) the tag's response from the FIFO Buffer.

We can accomplish all of these tasks by reading and writing specific registers on the MFRC522. We will explore these registers very soon. But first: how do we instruct the reader to transceive?

3.3.2 Commands, Part II

The MFRC522 has its *own* set of command codes, separate from the commands we have thus far been familiarized with. REQA and ANTICOLLISION are commands to be lodged into the FIFO Buffer, then sent from the reader, and finally accepted by the tag.

On the other hand, a TRANSCEIVE command is meant to be sent from *us*, the programmer, and accepted by the *reader*. The table below shows two commands that we, the programmer, need to send to the reader. The tag makes no contact at all with these commands.

Table 3: Relevant MFRC522 Commands

We execute...	**Command Code**	**Meaning**
IDLE	0x00	cancel the current (transceive) command
TRANSCEIVE	0x0C	transmit and receive from the FIFO Buffer

The purpose of the TRANSCEIVE command should be self-evident. But to understand the IDLE command, we must reiterate that one TRANSCEIVE command equals just one radio wave being sent out - not a constant stream of waves. The purpose of the IDLE command is to reset the reader's register states in preparation for another TRANSCEIVE command.

We mentioned before that commands should be lodged into the FIFO Buffer. But remember - those are commands intended for the tag, not the reader. The TRANSCEIVE command has a different home: a particular register on the MFRC522 known as the CommandReg register.

3.3.3 Registers

If you have worked with the Zen Cape's accelerometer (Assignment 3), then you have some experience with combing through a device datasheet and parsing its registers. The MFRC522 is no different. It has its own datasheet, and Section 9.2 of the datasheet gives an overview of all its available registers. Each register is one byte (8 bits) long, and each bit - or group of bits - represents some function, as described in the datasheet.

Below is a table of the bare minimum registers that you must interface with to obtain a UID. The accompanying support files reveal what values we personally chose to write to each of these registers, and briefly explain our chosen values through comments. If you are still confused by any value, we encourage you to pull up the datasheet alongside your favourite hex-to-binary converter, and decode the meanings bit by bit.

Table 4: Relevant MFRC522 registers, sorted by function

Address (hex)	Register Name	Function
01h	CommandReg	starts and stops command execution
0Dh	BitFramingReg	adjusts bit orientation for commands
04h	ComIrqReg	detects the reception of data from a tag
06h	ErrorReg	detects errors from the last command
0Ch	ControlReg	detects invalid data bits
09h	FIFODataReg	stores contents of the FIFO Buffer
0Ah	FIFOLevelReg	number of bytes stored in the FIFO Buffer
011h	ModeReg	(one-time) sets mode for transmitting
014h	TxControlReg	(one-time) turns RFID antenna on
015h	TxASKReg	(one-time) turns RFID 100% ASK on

From the first row of the table, we find the CommandReg that we mentioned in the previous section. So to execute a TRANSCEIVE command, we would write the TRANSCEIVE command code (0x0C) to the CommandReg register (0x01). Right? Well... not quite.

3.3.4 Formatting a Register Address

The MFRC522 would not accept 0x01 (0000 0001) as a valid register address. From the datasheet, we see that the MFRC522 demands that register addresses be formatted a certain way when requesting access:

Table 8. Address byte 0 register; address MOSI

7 (MSB)	6	5	4	3	2	1	0 (LSB)
1 = read 0 = write	address						0

Figure 2: Section 8.1.2.3 of MFRC522 datasheet

To be explicit:

- the MSB must be 1 if you intend to read, and 0 if you intend to write, a register.
- the actual register address you are trying to access should be condensed into the middle 6 bits.
- the LSB must always be 0 (which 0000 0001 violates).

To avoid confusion:

- only **register addresses** must be formatted in this manner. Values that you wish to write to the register, such as command codes, should not be changed.

To format a register address, we use bitwise operations. For example, if we want to read the VersionReg register (0x37), which tells us the firmware version of the MFRC522 model we have, we can use a combination of bit shifting (<<) and an OR bitmask (|).

Table 5: Bitwise operations on VersionReg 0x37

Binary	**Operation**	**Explanation**
0011 0111	-	starting value 0x37
0110 1110	<< 1	shifts all bits left one position
1110 1110	| 0x80	changes MSB to 1, while preserving all other bits

Notice that the final result conforms with the demands of Figure 2. The MSB is 1 because we are intending to read. The middle bits represent the actual address, 0x37. Lastly, the LSB is 0. Practically speaking, it is useful to create a helper function that formats the addresses for us. You will see this function, `Rfid_formatRegAddr`, in the support files.

3.3.5 Wiring to the BeagleBone

We have now covered all the background knowledge required to work with our MFRC522 reader. It's time to wire it up and start coding! The input voltage of the MFRC522 is 3.3V, which is a perfect match for the BeagleBone. We do not need any resistors, but we do need quite a few wires - six, to be exact. We can omit IRQ (interrupt request) and RST (reset). The following table shows the pin mappings for SPI1, CS0.

Table 6: SPI1 MFRC522 Pin Mappings

MFRC522 Pin	**BeagleBone Pin**
SDA	P9.28
SCK	P9.31
MOSI	P9.30
MISO	P9.29
GND	P9.1/P9.2
3V3	P9.3/P9.4

3.4 Working with RFID in C

In this section, we walk through sample code to lodge a REQA command into the FIFO Buffer, execute a TRANSCEIVE command, and receive an ATQA from a tag. This is a high-level overview, so some details have been omitted. You may find the comprehensive version in the accompanying support files.

You will see in this section that "working with RFID" is truly nothing more than reading and writing the appropriate registers. Most of the code itself should look almost trivially simple. Cheer up! You have already taken care of the hard part by understanding the inner workings of the reader and the tag.

3.4.1 Writing a Command to the FIFO Buffer

To lodge the REQA command code (0x26) into the FIFO Buffer, we must write the value 0x26 to the FIFODataReg register (0x09), which provides access to the FIFO Buffer. Assume we have a working function `Rfid_writeReg` that first properly formats, and then writes a value to, an MFRC522 register over SPI:

```
typedef uint8_t byte;

Rfid_writeReg(byte regAddr, byte value) {
    byte formattedRegAddr = Rfid_formatRegAddr(regAddr);
    Spi_writeReg(formattedRegAddr, value);
}
```

In theory, the code to lodge the REQA command into the FIFO Buffer should be as simple as:

```
#define FIFO_DATA_REG 0x09
#define REQA_CMD 0x26

Rfid_writeReg(FIFO_DATA_REG, REQA_CMD);
```

However, recall from Table 2 that REQA is a (7-bit)* command. We must first notify the reader to process only 7 bits of the full byte we feed it. We do this by setting the BitFramingReg register (0x0D) to 7.

```
#define BIT_FRAMING_REG 0x0D

Rfid_writeReg(BIT_FRAMING_REG, 0x07);
Rfid_writeReg(FIFO_DATA_REG, REQA_CMD);
```

And just like that, the FIFO Buffer is now ready for transceiving.

3.4.2 Executing a Transceive

Once we have lodged the REQA command into the FIFO Buffer, we want to make the reader transceive. Recall that we can achieve this by writing the TRANSCEIVE command to the CommandReg register (0x01).

```
#define COMMAND_REG 0x01
#define TRANSCEIVE_CMD 0x0C

Rfid_writeReg(COMMAND_REG, TRANSCEIVE_CMD);
```

However, this is not quite enough. It turns out the TRANSCEIVE command has a unique property. It requires a write to another, separate register to **actually** begin transceiving. We are already familiar with this register from the previous step - the BitFramingReg register. We used its lower bits to define the bit orientation as 7. Its MSB, however, has a completely different function; when turned to 1, it triggers the true execution of the TRANSCEIVE command.

```
#define COMMAND_REG 0x01
#define TRANSCEIVE_CMD 0x0C

Rfid_writeReg(COMMAND_REG, TRANSCEIVE_CMD);
Rfid_setBitmask(BIT_FRAMING_REG, 0x80);        // begin transceiving for real!
```

Do not worry about the implementation of `Rfid_setBitmask`, as that is done for you in the support files. Just understand its function: it sets the MSB of BitFramingReg to 1 with the bitmask 0x80 (1000 0000), while keeping all other bits the same. This is important, because we still want the bit orientation to be 7.

After the MSB of BitFramingReg is set to 1, the reader will transmit the REQA command from the FIFO Buffer through its antenna as a physical radio wave. If a tag picks up the signal, the tag will send back an ATQA. The ATQA is then picked up through the reader's receiver, and stored in the FIFO Buffer.

3.4.3 Checking for Errors

Recall that RFID is sensitive to data transmission errors, and one of our most important jobs as the programmer is to perform data validation to ensure that the ATQA has been transmitted successfully.

We can detect RFID errors at different layers. The first layer detects errors before you even read the tag's response from the FIFO Buffer. The MFRC522 internally records some error reports in certain registers: ErrorReg, and ControlReg. Immediately following a TRANSCEIVE command, we should read those registers and check if any errors were detected.

```
#define ERROR_REG 0x06
#define CONTROL_REG 0x0C

byte errorRegValue = Rfid_readReg(ERROR_REG);
byte controlRegValue = Rfid_readReg(CONTROL_REG);
```

Using the datasheet to understand what each bit of the error registers represents, we devise a bitmask that screens for those pertinent bits. We perform the AND bitwise operation on the register value and the bitmask. If we receive a non-zero value, that means an error has occurred. We return an appropriate enum error code for debugging purposes.

```
#define ERROR_REG_BITMASK 0x13        // screens for miscellaneous errors
#define CONTROL_REG_BITMASK 0x07      // screens for "invalid bit" errors

if (errorRegValue & ERROR_REG_BITMASK) {
    return RFID_MISC_ERR;
}
if (controlRegValue & CONTROL_REG_BITMASK) {
    return RFID_INVALID_BYTE_ERR;
}
```

The second layer of error checking occurs after reading the tag's response from the FIFO Buffer. This layer's job is to confirm that the tag's response is indeed the exact sequence of bytes we are expecting. Literally speaking, after you read the ATQA from the FIFO buffer, you should explicitly:

(a) check the length of the response (reject if not precisely 2 bytes (16 bits)),

(b) check the content of the response (reject if not precisely (0x04, 0x00)).

3.4.4 Reading a Tag's Response from the FIFO Buffer

To read from the FIFO Buffer, we use the same register we used to write to the FIFO Buffer: FIFODataReg.

Internally, the FIFO Buffer keeps track of a pointer. Each time one byte is read from the FIFO Buffer, the pointer automatically moves to the next byte. Therefore, we do not have to worry about which address location of the FIFO Buffer to read from. We simply read the register n times, where n is equal to the number of bytes we have received. Between each byte being read, the pointer updates accordingly for us.

We can determine how many times to read from the FIFO Buffer by reading the FIFOLevelReg register (0x0A), which tells us how many bytes the FIFO Buffer is currently holding.

```
#define FIFO_LEVEL_REG 0x0A
#define FIFO_DATA_REG 0x09

byte numFIFOBytes = Rfid_readReg(FIFO_LEVEL_REG);     // an ATQA would return 2 bytes

for (int i = 0; i < numFIFOBytes; i++) {
    tagResponse[i] = Rfid_readReg(FIFO_DATA_REG);
}

// tagResponse contains:  [0x04, 0x00]
```

Congratulations! You have successfully sent a REQA command to a tag, and received an ATQA.

3.5 Obtaining a UID

To recap: in order to obtain a tag's UID, the following steps must occur. (Error checking has been omitted for clarity, but should be performed wherever possible.)

1. We, the programmer, lodge the REQA command into the reader's FIFO Buffer.
2. We, the programmer, execute the TRANSCEIVE command.
3. The reader transmits the REQA command, probing for tags within range.
4. A tag receives the command. It sends an ATQA back to the reader, signalling that it is ready to receive a UID request.
5. We, the programmer, lodge the ANTICOLLISION command into the FIFO Buffer.
6. We, the programmer, execute the TRANSCEIVE command.
7. The reader transmits the ANTICOLLISION command to the selected tag.
8. The selected tag receives the command. It sends its 4-byte UID, and its BCC, back to the reader.
9. We, the programmer, retrieve the UID and BCC from the FIFO Buffer.

3.6 Support Files

In the support files accompanying this guide, you will find a half-complete RFID implementation in `main.c`, `rfid.h`, and `rfid.c`. These files build off of the SPI code from Section 2, so you must complete that module first, if you have not already. You should read these files in the order: `rfid.h`, then `main.c`, then `rfid.c`.

Currently, the RFID module is able to send a REQA command, and receive an ATQA through the function `Rfid_searchForTag`. However, the code to send an ANTICOLLISION command, and therefore obtain a UID, is missing. We encourage you to use the pseudocode above - as well as the knowledge you have gained from reading this guide - to implement the missing function.

Hint: Sending an ANTICOLLISION command follows the exact same principles as a REQA command, so the structure will be very similar to the provided `Rfid_searchForTag` function!

Hint: Recall that ANTICOLLISION's command code (0x93 0x20) is 2 bytes long. To handle this, you need to write to the FIFODataReg register two times: 0x93 first, followed by 0x20.

Hint: ANTICOLLISION is a full 8-bit command, so you also need to alter the value you write to the BitFramingReg register. What value should you write? The datasheet should come in handy.

Hint: The internal function `Rfid_performChecksum` has been provided for you, and error checks a UID against its BCC: it takes the XOR over a UID's 4 bytes, and checks if that result is equal to its BCC.

3.7 Troubleshooting

- If your RFID reader does not seem to work at all:
  - If `Rfid_printFirmwareVersion` fails, then your RFID reader itself may be broken. Try the program on another MFRC522 unit.

  - Ensure that you have not removed or altered any code from the original support files outside of the assigned TODOs, especially the functions `Rfid_init` and `Rfid_transceive`.

- If your RFID reader can read the firmware version correctly, but subsequent operations seem to fail or return strange results:
  - You may have implemented `Spi_writeReg` or `Rfid_writeReg` incorrectly, so your writes are not performing the expected roles. You can test this theory by writing a value like 0x07 to the BitFramingReg register, and then calling `Rfid_readReg`, which is already implemented correctly, on that register. Check if the read value matches the value you wrote.

- If you are occasionally receiving an inaccurate or different UID from a single tag:
  - You may not have implemented the error checking properly, leading to a poorly transmitted UID slipping through. Follow the structure of the (3) error checks seen in `Rfid_searchForTag`.

- If your RFID reader reads a tag's UID successfully at first, but subsequently does not seem to read any tag until you restart the program:
  - In altering the module for your own needs, you may have changed the structure of main RFID reading loop to be slightly different than the original support files. Our reading loop was constructed very carefully to avoid any infinite loop problems. Try changing it back to the exact original. If that solves the problem, but you still need to change the loop, then ensure that you do not change the general *structure* of the loop.
  - You may be holding multiple tags within the reader's RF field, which has been documented to cause undesired behaviours. You should plan to only read one tag at a time.

- If your application occasionally returns a `RFID_TIMEOUT_ERR` even when a tag is permanently sitting on the reader, and this intermittent inaccuracy is unacceptable for your application:
  - This happened to us as well, and we were unable to determine the cause. For our use case, employing the "N samples past the threshold" strategy described in lecture to essentially ignore an errant timeout error was sufficient.

4 Appendix

Below are a list of resources referenced in the creation of this guide. Most of these are official documentation, which go into much greater detail on their respective technologies. If you wish to do anything deeper than what has been explored in this guide (such as reading and writing any block other than the Manufacturer Block on the MIFARE Classic 1K), these resources are compulsory.

If any of the following lead to dead links in the future, try pasting the link into the Wayback Machine to see if you can find a saved copy.

Thank you for reading this guide. Good luck, and happy coding!

MFRC522 Sample Implementation

miguelbalboa's Arduino MFRC522 Library

Datasheets

MFRC522 Datasheet

MF1S503x/MIFARE Classic 1K Datasheet

RFID Technology Specification

ISO/IEC 14443-2 (Part 2)

ISO/IEC 14443-3 (Part 3)

UID Specification

AN10927