



# **Mifare Reader Communication Protocol**

## **- SPV1**

*Release 1.0.0*

**SonMicro Elektronik**

**May 02, 2017**



# CONTENTS

<b>1</b>	<b>INTRODUCTION</b>	<b>1</b>
<b>2</b>	<b>HARDWARE INTERFACE PRECAUTIONS</b>	<b>3</b>
2.1	UART . . . . .	3
2.2	RS485 . . . . .	4
2.3	I2C . . . . .	4
<b>3</b>	<b>SPV1 UART PROTOCOL</b>	<b>7</b>
3.1	Example Command/Response Frame . . . . .	8
3.2	ARDUINO® UART Example . . . . .	9
<b>4</b>	<b>SPV1 I2C PROTOCOL</b>	<b>13</b>
4.1	I2C NOTES . . . . .	14
4.2	I2C SENDING COMMAND . . . . .	15
4.3	I2C RECEIVING RESPONSE . . . . .	18
4.4	I2C Example - Card UID Reading with CmdActivateAll . . . . .	23
<b>5</b>	<b>TRADEMARKS</b>	<b>35</b>
<b>6</b>	<b>DOCUMENT REVISION HISTORY</b>	<b>37</b>



## INTRODUCTION

This document explains the details of bottom level communication protocols (UART and I2C) used with SonMicro Mifare readers and modules. It explains the communication principles, structures and the base logic.

You may skip this section if a SDK, Software Development Kit, or code/API library that handles the protocol is provided for your development environment. SonMicro provides and keep developing easy to use SDKs, libraries and code samples for Python, Microsoft .NET , C, C++ for popular development environments including microcontrollers (i.e. ARDUINO®, ARM® mbed™, Raspberry) and operating systems Linux, Mac, Windows.

SonMicro has variety of modules, readers and boards with different settings, hardware and firmware versions. In this document, for easy reading, they will be called **SMR (SonMicro Mifare Modules and Readers)** in general. Please also check the target device's datasheet together with this document.

SMR basically supports two serial bottom level communication protocol; UART and I2C. The bit level protocol is same for UART, RS232, RS485 and USB Virtual Serial Port but the electrical characteristics are different. UART can be extended with a **hardware interface** to support RS232, RS485 and USB Virtual Com port.

UART and I2C communication shares same protocol and is based on **Spv1, Standard Serial Protocol Version 1**. Spv1 defines the rules of the communication protocol and transmitted/received (tx/rx) frame structure. There are only minor usage differences for the **Spv1** protocol for UART and I2C.

The Mifare command set or API which is explained in the related firmware manual document is written on top of **Spv1**.

---

**Note:** Command set and some features are based on the reader/module firmware version. There are different firmware versions for different type of applications or custom requirements. Base protocol, **Spv1** is same for all commands. Most demanded features are combined in single standard firmware version.

For firmware-specific functions and a list of supported command set, please also investigate the related firmware manual document.

---

**Attention:** UART is the primary communication interface of SMR and it is the only supported interface required for firmware upgrade, and pre-configuration (i.e. enabling I2C). Thus, it is strongly recommended to have UART pins if you are using Mifare module on your own board with a microcontroller, even if I2C is used. It is also a good practice to be able to bypass the external microcontroller UART TX pin(i.e. with a jumper), if exist any, to prevent output conflict with another UART TX (i.e. USB-UART converter to upgrade the module on board)

**UART is required:**

- For upgrading the module with a new firmware. Notice that; firmware upgrader is a PC software tool, thus USB-UART or RS232 interface (i.e. max232) is required to upgrade.
- For enabling the I2C, if were not enabled at the factory.

**Attention:** UART shares the same protocol with RS232, RS485 and USB Virtual Com Port readers. Upgrading over RS232 and USB Virtual Com Port is possible with proper hardware interfaces (i.e. USB-UART Converter). **However RS485 may not be upgraded.**

**RS485 Warning!**

- RS485 readers may not be upgraded depends on the integrated bootloader firmware version. Some bootloader program does not support DE signal management required for RS485 communication. Thus, communicating with the bootloader program is not possible. Also notice that the integrated bootloader program on the readers is read-only and cannot upgrade itself.
- RS485 readers, if supports upgrading (check bootloader firmware version), may not be upgraded if connected to the other readers on the bus. In such a case, they need to be upgraded one-to-one and isolated from other readers on the bus.

## HARDWARE INTERFACE PRECAUTIONS

**Attention:** Please first read the precautions explained in this section to protect your reader and the external controller.

Please also **make sure to read the hardware specifications** of the ultimate module/reader/board explained in target device **datasheet** or **hardware manual** before you make any hardware connections.

### 2.1 UART

SMR's UART I/O signal levels are CMOS/TTL level compliant and **5V tolerant**.

UART communication protocol (bit level) is exactly same for RS232, RS485 and USB Virtual Com Port readers but the hardware interfaces are different due to the different electrical characteristics.

SMR's **UART TX** output can be 3.3V or 5V depending on the module/reader type. Please check hardware specification of the target SMR for more details. For example, SM130 have 5V levels, SM5210/SM5211-SMD modules have 3.3V output levels.

SMR's **UART RX** input can be connected to any 3.3V or 5V controller's UART TX output pin, as they are 5V tolerant.

**Attention:** *Protect your external controller*

- **It is important that your controller UART RX pin must be 5V tolerant if SMR's UART TX level is 5V. Otherwise your controller is likely to be damaged.** For example; Raspberry Pi GPIO pins are not 5V tolerant. SMR with 5V UART interface (i.e. SM130) should be avoided to connect directly to any none-5V-tolerant controller. A level converter or a resistor divider is necessary for direct connection between 5V UART TX(SMR) and 3.3V UART RX (i.e. Raspberry or none-5V-tolerant controller)
- **It is also important that, even though some kind of SMR's UART TX output pin is 3.3V (i.e. SM5210/SM5211-SMD), they might be used on a reader, base board or evaluation kit that is connected to 5V RS232 interface internally, or a USB-UART Converter is connected with 5V level.** You should always watch the none-5V-tolerant controller (i.e. Raspberry Pi) never interacts with a 5V signal level.

**Attention:** *Protect your SMR (SonMicro Mifare Reader/Module)*

- **Never connect SMR's UART pins (or any other pin) to an RS232 device directly.** RS232 have +/- 12V signals, shares the same bit level protocol with the UART but they have different electrical characteristics. You should avoid connecting UART pins directly to the PC com port or any USB/RS232 converter. Otherwise the SMR will be damaged. For such a requirement, USB-UART Converter or UART-RS232 (i.e. max232) hardware interface is required between the Module-UART and RS232 (i.e. PC) sides.

## 2.2 RS485

Spv1, Standard Serial Protocol Version 1, supports RS485 interface by using DE (Data Enable) signal and node address byte in the spv1 protocol frame with the second-generation modules (i.e. SM5210/SM5211-SMD)

SM5210/SM5211-SMD modules can be connected to RS485 interface (i.e. ST485). There are also ready-to-use RS485 readers and boards available with bias, diode protection and simple filter circuits integrated with the second-generation modules.

**SM130 modules have no RS485 support.**

**Attention:**

- RS485 infrastructure (cabling, termination resistors etc.) is extremely important for reliable communication. It is assumed that you have understanding of RS485 interface and experience with the infrastructure. There is no any extended support is offered for RS485 caused problems. It is highly recommended to investigate RS485 networks. For example, a star or a random style connections should be avoided. Even the application confirmed to be working good, you may have problems in midterm.
- **Upgrading the module/reader firmware over RS485 is not supported.** See at [RS485 Warning](#)

## 2.3 I2C

SonMicro Mifare modules support I2C communication. Some of the ready-to-use Readers and Boards that are integrated with the SonMicro modules include the I2C connections as well. Please check our website for available boards.

SM130(Old Generation) I2C support is limited and requires different firmware version to be upgraded, I2C 2.8.

SM5210/SM5211-SMD, second-generation modules or second-generation based readers support the I2C with full command set. However please be aware that **I2C feature needs to be enabled by relevant configuration command.**

Modules' I2C signal levels are CMOS/TTL level compliant and **5V tolerant**. Depends on the Module/Reader and the pull up resistor circuit, the output signal level can be 3.3V or 5V. Please check the hardware specification of the target module/reader for more details.

**Attention:** *Protect your external controller*

- **It is important that your controller I2C pins must be 5V tolerant if SMR's I2C level is 5V. Otherwise your controller is likely to be damaged.** For example; Raspberry Pi GPIO pins are not 5V tolerant. SMR with 5V I2C interface (i.e. SM130) should be avoided to connect directly to any none-5V-tolerant controller. I2C level converter (5V - 3.3V) circuit may be necessary in this case.
- **It is also important that, even though some of SMR I2C output signal level is 3.3V (i.e. SM5210/SM5211-SMD), they might be used on a mother board or an evaluation kit (i.e. SM5210-EK) that pull up resistors are connected to 5V internally instead of 3.3V.** In such a case, you should not connect I2C pins to the none-5V-tolerant controller. You need to check the hardware specification of the ultimate board.



## SPV1 UART PROTOCOL

The communication between the host and the SMR can take place at 9600bps, 19200bps, 38400bps, 57600bps or 115200bps N, 8, 1.

SMR communicates at 19200bps, N,8,1 as default. Baud rate can be changed with relevant configuration command.

---

**Note:** This document does not cover firmware behaviors such as sending card UID as ASCII output mode.

If you configure SMR to read card UID with ASCII output (auto mode with ASCII option), and if this is the only behavior you want, then the spv1 protocol details explained in this topic is not necessary to know. In this case, you can just listen for the incoming ascii data for card UID and process them in your hardware or software platform.

---

The host (i.e. external controller or P.C) first sends the command, then SMR process the command and replies with a response. The host can analyze the reply to check if the operation was successful or if any error occurred during the operation.

There are also some modes (i.e. auto read mode) or commands (i.e. Seek For Tag) that SMR sends response asynchronously without expecting any command from the host (Ex: whenever a card is detected). One of the other possibility is; firmware version is sent from SMR on power up if the SendFirmwareVersion on startup configuration option is enabled.

Following is the UART frame structure for the commands sent by the host and response received from SMR:

<b>Header</b>	<b>Node Address</b>	<b>Length</b>	<b>Cmd/Response</b>	<b>Data</b>	<b>Checksum</b>
1 byte(0xFF)	1 byte	1 byte	1 byte	N bytes	1 byte

*Table 3 Spv1, Standard Protocol Version 1, Command/Response byte array frame structure for UART/RS232/RS485 and Virtual Com port.*

1. **Header:** This is a single byte that indicates the beginning of the frame. This byte is fixed to 0xFF.
2. **Node Address:** This is used as node address to support RS485. Use 0x00 if RS485 is not used. 0x00 and 0x01 node addresses are special, if commands sent to these addresses, readers will accept commands whatever their node address is. Thus, node address 0x00 and 0x01 should not be used in RS485 systems. Otherwise all nodes will reply to the command and they will cause bit collision on the RS485 bus.
3. **Length:** This byte is used to indicate the length of the payload data. This includes command byte plus the number of data bytes. Data length can be found as *Length* minus 1
4. **Command/Response ID:** This byte is used to instruct the module which operation to perform. For the response packets, this indicates which command's response
5. **Data:** Data bytes are the parameters of a command or response.
6. **Checksum:** This is the checksum byte. This byte is used on the host as well as the module to check the validity of the packet and to trap any data corruption. This is calculated by adding all the bytes in the packet except the header byte

**Note:** You can observe the sent(TX) and received (RX) packages with the software tools provided for Mifare applications (i.e. SMRFID Mifare, Mifare Quick)

**Please notice that** it is not possible to send command and receive response from UART/RS232 terminal with ASCII communication (i.e. HyperTerminal). Spv1 protocol consists of hex byte arrays, some of the hex values cannot be represented as ASCII characters. If you would like to send commands from PC manually then please check for RS232 terminal that supports raw hex byte communication. For example **Hterm** is a useful RS232 terminal that supports raw byte communication.

---

**Hint:** If you are not using any provided SDK or code sample and would like to implement your own spv1 communication interface you can apply the following procedure to receive the protocol data.

- First listen 4 bytes. (Header, node address, Length, Command/Response ID)
- Extract the Length and listen for remaining expected bytes.
- Verify the checksum byte.
- Process the response by Command/Response ID byte.

Another approach would be to listen for received bytes one by one (thru IRQ channel) and use a state for each condition and clear or skip to the next state. This is “non-blocking” way to handle the protocol. Example states:

- #define SPV1SOFState 0
  - #define SPV1AddressState 1
  - #define SPV1DataLengthState 2
  - #define SPV1CommandState 3
  - #define SPV1DataState 4
  - #define SPV1ChecksumState 5
- 

### 3.1 Example Command/Response Frame

Read Firmware version example

**Command frame to be sent:**

TX>> FF 00 01 81 82

- 0xFF (Header)
- 0x00 (Node Address)
- 0x01 (Length)
- 0x81 (Read Firmware Version command)
- 0x82 (Checksum)

*This command has no parameters.*

**Response frame received:**

RX<< FF 00 0C 81 55 4D 20 31 2E 33 52 56 31 31 34 1F

- 0xFF (Header) Always fixed.

- 0x00 (Node Address) Data is received from 0x00 node address.
- 0x0C (Length) We have 12 minus 1 = 11 bytes data parameters , that is the firmware version string
- 0x81 (Response ID) Read Firmware Version command/response ID
- [0x55..0x34] (Data, 11 bytes) Ascii representation of 'UM 1.3RV114'
- 0x1F (Checksum) Sum of bytes from node address to checksum. [0x00 .. 0x34] , Header byte 0xFF, and final checksum byte is excluded.

## 3.2 ARDUINO® UART Example

Please check for the support page for code examples and source files.

### Card UID Reading with CmdActivateAll

Following code is prepared for the Arduino and same logic can be applied to all UART platforms. It reads the card UID by sending CmdActivateAll command and receive the response.

```

/*
 * When programming the Arduino, you may disconnect the UART_TX
 * pin of the module(UART RX) pin of the Arduino.
 * Otherwise Arduino cannot be programmed.
 *
 * If Serial Monitor is going to be used, open it with the same
 * baudrate of mifare module.
 * The default baudrate is 19200bps for the mifare modules
 *
 * This example project only illustrates basics of the UART
 * communication with SM130 and SM5210 explained in
 * SPV1 UART PROTOCOL section.
 * It provides an example to read card UID using
 * CmdActivateAll(SELECT_TAG) command. It does not provide a full
 * command set library or example. Check if there is a complete
 * library is written for your environment for faster integration.
 *
 */

// You can use Arduino UART pins to be connected to the Mifare Module
// At the same time you can use the Serial Monitor. However this can
// cause dirty logs as Mifare module responses will shown as well.
// Mifare responses shown as unknown characters because
// Arduino Serial Monitor shows ascii characters only.
// Notice that the serial.print used for log purposes will also be
// transmitted to the mifare module. But it wont cause any problem
// for Mifare module functionality.

// Use debug and serial.print only when it is needed and consider the
// above comments.

#define DEBUG

/* Command ID for ACTIVATE_ALL(SELECT_TAG) */
#define CMD_ACTIVATE_ALL 0x83

void setup() {

```

```
// put your setup code here, to run once:
pinMode(LED_BUILTIN, OUTPUT);
Serial.begin(19200);
}
```

```
byte ReadResponse(byte *rxBuffer)
{
    int count;
    byte spv1framelength = 0;
    byte checksum = 0;

    // Read 4 bytes first.
    // We always have minimum 4 bytes response for spv1
    // Extract the length byte and read remaining data.
    count = Serial.readBytes(rxBuffer, 4);
    if (count==4)
    {
        // 4 byte read success.
        // Extract the length indicator byte.
        spv1framelength = rxBuffer[2];
        // Read remaining bytes.
        count = Serial.readBytes(&rxBuffer[4], spv1framelength);
        if (count==spv1framelength)
        {
            // Read remaining byte success.
            // Checksum control
            checksum =0;
            // Do not include header byte in checksum calculation.
            for(int i=1;i<spv1framelength+3;i++)
                checksum += rxBuffer[i];

            if (checksum!=rxBuffer[spv1framelength+3])
            {
                // Checksum Error
                return 0;
            }
            else
            {
                return spv1framelength;
            }
        }
        else
        {
            // Possibly timeout
            return 0;
        }
    }
    else
    {
        // Possibly timeout
        return 0;
    }
}
```

```

void loop() {
  // put your main code here, to run repeatedly:
  byte txBuffer[32];
  byte rxBuffer[32];
  byte spv1framelength = 0;
  byte commandID;

  digitalWrite(LED_BUILTIN, HIGH);
  delay(500);

  // cmdActivateAll(SelectTag command)
  txBuffer[0] = 0xFF;
  txBuffer[1] = 0x00;
  txBuffer[2] = 0x01;
  txBuffer[3] = 0x83;
  txBuffer[4] = 0x84;

  Serial.write(txBuffer, 5);
  spv1framelength = ReadResponse(rxBuffer);

  if (spv1framelength>0)
  {
    commandID = rxBuffer[3];
    if (commandID==CMD_ACTIVATE_ALL)
    {
      if (spv1framelength==2)
      {
        #if defined DEBUG
          Serial.print("Response Error Code:");
          Serial.print(rxBuffer[4],HEX);
          Serial.print("\r\n");
          if (rxBuffer[4]=='N')
          {
            Serial.print("No Tag \r\n");
          }
        #endif
      }
      else if ((spv1framelength==6) || (spv1framelength==9))
      {
        int uidlength = spv1framelength - 2;
        #if defined DEBUG
          Serial.print("UID Length:");
          Serial.print(uidlength);
          Serial.print(" , UID:");
        #endif

        // Card UID is reported reverse. (LSB first)
        for (int i = 0; i<uidlength; i++)
        {
          Serial.print(rxBuffer[4 + uidlength - i],HEX);
          Serial.print("-");
        }
        Serial.print("\r\n");
        #endif
      }
      else
      {

```

```
        #if defined DEBUG
        Serial.print("Unknown response \r\n");
        #endif
    }
}
else
{
    // Response is not for CmdActivateAll.
    #if defined DEBUG
    Serial.print("Response does not belong CmdActiveAll \r\n");
    #endif
}
}
else
{
    #if defined DEBUG
    Serial.print("Cant get any spv1 response");
    #endif
}

digitalWrite(LED_BUILTIN, LOW);
delay(500);
}
```

## SPV1 I2C PROTOCOL

SonMicro Modules/Readers supports I2C communication as an alternative to UART. Both communication interface can be used at the same time.

SMR functions as an I2C slave. See<sup>1</sup>. Default address of the SMR I2C slave is 0x42. 7-bit addressing is supported. I2C address can be changed by relevant configuration command.

The slave can support clock rates of 50 kHz, 100 kHz and 400 kHz.

SPV1 I2C protocol explained here is valid both for first-generation module(SM130) and the second-generation modules (i.e. SM5210/SM5211-SMD). Communication structure and principle is almost same for both modules. However, there are small differences exist both in hardware and usage. Usage differences are explained in this document with supported examples. Please reference module datasheet for hardware specs. (i.e. 3.3v - 5V CMOS/TTL differences)

---

### Note:

- For **SM130(First-generation module)** a different firmware version(i2c 2.8) is required to be upgraded for I2C operation. I2C command set and the behavior is also limited on SM130. Please also notice that **SM130 have 5V signal levels** unlike SM5210/SM5211-SMD (3.3V level). Please contact us to request I2C firmware for SM130. Please also consider migrating to SM130-M2, second-generation module integrated with SM5211-SMD module with SM130 pinout.
- For **SM5210/SM5211-SMD (Second-generation module)** firmware versions stdMfr 1.0.0, UM1.3R 1.1.4 and the greater versions supports the I2C communication fully and needs to be enabled by relevant configuration command. Please also notice that **SM5210/SM5211-SMD have 3.3V signal levels and is 5V tolerant**.

---

SMR I2C frame is similar to UART but there is no any header and node address byte in the frame.

Length	Cmd/Response	Data	Checksum
1 byte	1 byte	N bytes	1 byte

Table 4 Spv1, Standard Protocol Version 1, Command/Response byte array frame structure for I2C.

1. **Length:** This byte is used to indicate the length of the payload data. This includes the command byte and the number of data bytes. Data length can be found as Length minus 1
2. **Command/Response ID:** This byte is used to instruct the module which operation to perform. For the response packets, this indicates which command's response
3. **Data:** Data bytes are the parameters of a command or response.
4. **Checksum:** This is the checksum byte. This byte is used on the host as well as the module to check the validity of the packet and to trap any data corruption. This is calculated by adding all the bytes in the packet except the header byte

---

<sup>1</sup> SM5210/SM5211-SMD can act as a master I2C and can control i.e. EEPROM. However this is an application specific feature and not related with the Spv1 I2C protocol.

## 4.1 I2C NOTES

Default slave address is 0x42 but there may be a possibility to have different address if an upgrade operation overwrites to the configuration block of the SMR. Before you start developing with I2C, please check your firmware version, I2C address and if it is enabled with the software tools provided.

### 4.1.1 7-Bit Addressing

I2C protocol in general have 7-bit or 8-bit versions of addresses. SMR uses 7-bit addressing version. The LSB bit of address byte determines if the operation is read or write.

If the master I2C device is using 8-bit addressing then a left bit shift operation is necessary when communicating with the SMR. The last bit is set or cleared for read or write operation, respectively.

If the master I2C device supports 7bit addressing (i.e. Arduino Uno) then I2C address may be used without any bitwise operations. If you don't know the addressing version of master I2C controller then you can try both addressing versions.

---

**Note: For I2C master device supporting 8-bit addressing:**

- Use 0x84 when sending a command to the 0x42 slave address
- Use 0x85 when reading from 0x42 slave address

**Write operation**

The I2C slave address is shifted 1 bit to the left and the last bit is cleared when writing to the I2C slave device.

$0x42 \ll 1 = 0x84$

**Read operation**

The I2C slave address is shifted 1 bit to the left and last bit is set to 1 (bitwise OR)

$0x42 \ll 1 = 0x84 \mid 0x01 = 0x85$

**For I2C master device supporting 7-bit addressing:**

- Use 0x42 when writing/reading
  - For example, Arduino Uno supports 7-bit addressing with the *Wire Library* and I2C address can be used without any bitwise operations.
- 

### 4.1.2 Reading final byte with Nack

Reading the last byte just before the **i2c stop condition** needs to be done by NACK'ed read. This is to indicate master is done reading data from slave. For most MCU compilers, this has a representation as `i2c_read(0)`, `i2c_read(NACK)` or `i2c_read(false)`. Also it is possible that the provided API or hardware feature can make a dummy read with NACK automatically when using `i2c_stop()`. This dummy read if exist will have no negative effect on the communication.

### 4.1.3 Reading from slave - Polling or IRQ

There are two ways to read data from slave. You can continuously read from slave to check if there is data available, or you can wait for slave to send signal, a logic high on the DE(DR) pin whenever a response is ready.

## Polling Method

Reading from slave device can be done continuously by polling(reading) a single byte from slave and check if it is none-zero. This is the length byte of the Spv1 frame buffer (see Table 4). When it is none-zero it means that there is a response ready and master can read available bytes indicated with the length byte.

**Note:** If polling mode is used then it is recommended to use delays (i.e. 10 milliseconds or more) between pollings (single byte response check queries). This is to prevent frequent disruption of SMR to do its duty without interruption.

## IRQ Method

Another approach is to use DR(DE) IRQ pin of the SMR. This pin goes high to inform I2C master controller that there is a response ready to be polled. In this case master only polls data whenever the DE(DR) IRQ pin of the SMR goes high. Using DE(DR) pin is optional.

### 4.1.4 Special Note for SM130

**Attention:** The expected response prepared by SM130 will be preserved for only 50 milliseconds and then the frame buffer will be cleared (Length byte will be zero). In other words, when a command is sent to the SM130 module, there is only 50 milliseconds time to read the response. Otherwise response will get cleared. This can be troublesome when debugging your application.

There is no such a limitation with the second-generation modules (i.e. SM5210/SM5211-SMD)

## 4.2 I2C SENDING COMMAND

To send a command, properly constructed I2C Spv1 frame(Table 4) for that command is transferred to the slave.

Phase 1	Phase 2	Phase 3	Phase 4
Start Condition	Address(write)	Spv1 Frame Buffer(write)	Stop Condition

Table 4.2 I2C bus sequence for sending a command

### 4.2.1 I2C Bus(SDA/SCL) Activity for CmdActivateAll

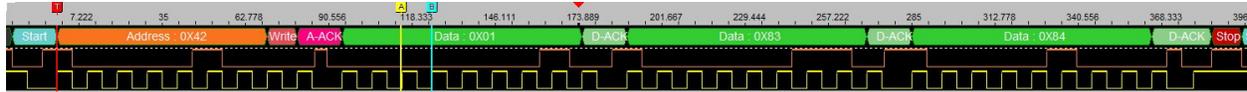
CmdActivateAll, previously named SELECT\_TAG, command frame for I2C is as follows:

Length	Cmd/Response	Data	Checksum
0x01	0x83	No Data	0x84

Table 4.2.1 CmdActivateAll(SELECT TAG) Command, I2C Spv1 Frame

This command has no data, and has 3 bytes needs to be transferred to the i2c slave device. [0x01,0x83,0x84]. Remember that the Length Byte(first byte) in the I2C Spv1 frame Table 4 indicates the number of the following bytes in the Spv1 frame, excluding the checksum byte.

Following is the logic analyzer output of sending 3 bytes for CmdActivateAll command. [0x01,0x83,0x84]



Start	Address	Write	A-ACK	Data	D-ACK	Data	D-ACK	Data	D-ACK	Stop
Start	42	Write	A-ACK	01	D-ACK	83	D-ACK	84	D-ACK	Stop

Figure 4.2.1 I2C bus SDA/SCL activity of sending CmdActivateAll Command

**Hint:** SM5210/SM5211-SMD module has an on-board LED as an indicator for different behaviors. One of the useful functions of this LED is, it will blink if a command is received (thru UART or I2C) successfully. Thus, you can test and observe this LED if the command is received by SM5210/SM5211-SMD module successfully.

## 4.2.2 Send Command Example - Generic

Examples provided here is for illustrating the base logic. Please check at our support page for full code examples or ready to use libraries for faster integration if provided any.

**Note:** In some compilers, when using provided I2C low level APIs, it may be required to check the **write completed flag** for the last written byte. For example; when using low level I2C APIs of ARM® mbed™, `i2c_stop()` does not check the write completed flag, thus causing the last byte not transferred to the slave. To prevent this, a dummy write, `i2c_write(whatever)` can be used just after the last byte `i2c.write(checksum)` and just before the `i2c_stop()`. Dummy write (is a trick) helps to check the previous byte(protocol last byte - checksum byte) is written. It is not expected to be transferred due to `i2c_stop()` will prevent it. Please notice that there is no need for this trick if you are using higher level I2C APIs or you are given API to check write completed flag.

**ARM® mbed™ provides also higher level I2C APIs which is recommended to use. Examples are given next.**

```
// NOTE:
// In some compilers(i.e. mbed) there may be no API
// to check if the write is completed. In this case;
// i2c_stop() prevents last byte to be written to the bus.
// Thus, a dummy write(that shall not actually write to the bus)
// or a delay(not recommended) can be inserted just after
// last written byte and just before the i2c_stop()
// Please first consider to use higher level APIs if provided
// or check if there is write completed API exist.

// use 0x42 or 0x84 for address when writing if master
// supports 7-bit address or 8-bit address.

// Phase 1 - Start Condition
i2c_start();

// Phase 2 - Address
i2c_write(0x84); // address. or i2c_write(0x42)

// Phase 3 - Spv1 Frame Buffer[0x01,0x83,0x84] for cmdActivateAll
i2c_write(0x01); // length. n data bytes + 1
```

```

i2c_write(0x83); // cmdID. ActivateAll(Select Tag)
i2c_write(0x84); // checksum
// Make sure the last byte(checksum) is written to the bus.
// Depends on the compiler you are using:
// If necessary just before stop condition
// check here if write is completed or use a dummy write that
// wont transferred to the slave.

// i2c_write(0) // Dummy write to make sure checksum is transferred
// or check write completed flag
// or insert delay to make sure the last byte is written.
// Delay is not recommended.

// Phase 4 - Stop Condition
i2c_stop();

```

### 4.2.3 Send Command Example - Arduino

Please note that `Wire.begin()`; needs to be included in setup method in Arduino

```

byte CmdActivateAll()
{
    byte status;
    Wire.beginTransaction(0x42); // address
    Wire.write(byte(0x01)); // length
    Wire.write(byte(0x83)); // command
    Wire.write(byte(0x84)); // checksum.
    status = Wire.endTransmission(); // stop transmitting
    return status;

    // 0:success
    // 1:data too long to fit in transmit buffer
    // 2:received NACK on transmit of address
    // 3:received NACK on transmit of data
    // 4:other error
}

```

### 4.2.4 Send Command Example - ARM® mbed™

Please note that global `I2C i2c(I2C_SDA, I2C_SCL)`; needs to be included in the cpp file.

```

I2C i2c(I2C_SDA, I2C_SCL);

// Notice 0x84(first argument in the i2c.write())
// It is the version of 1 bit left-shift of 0x42 slave address.
// Tested hardware environment is ST Nucleo-F302R8

int CmdActivateAll()
{
    int status;
    char spv1frame[3];
    spv1frame[0] = 0x01;
    spv1frame[1] = 0x83;
    spv1frame[2] = 0x84;
    status = i2c.write(0x84, spv1frame, 3);
}

```

```

return status;
// @returns
// 0 or non-zero - written number of bytes,
// negative - I2C_ERROR_XXX status
}

```

## 4.3 I2C RECEIVING RESPONSE

SMR will construct spv1 frame (see Table 4) whenever there is a response is ready. Otherwise the first byte (length indicator) of the spv1 frame is zero indicating that there is no response ready.

To read the whole spv1 frame from slave (SMR) conditional reading is required depends on the length indicator byte.

### 4.3.1 Standard Method to Receive Response

Both SM130(First-generation) and SM5210/SM5211-SMD(Second-generation) modules support standard method to receive response as explained in this topic. However, there are alternative methods to receive response if the standard method cannot be used in your development environment (i.e. Arduino) Please check for the next topics for alternative methods to receive response if you are not provided low level I2C APIs to implement the standard receive method.

Spv1 Response Frame					
Phase 1	Phase 2	Phase 3	Phase4	Phase5	Phase 6
Start Condition	Address(read)	Read Spv1 Frame First Byte (length indicator)	Read remaining if exist	Read Last Byte with NACK	Stop Condition

Table 4.3.1 I2C bus sequence for receiving Spv1 response frame from SM130(First Gen) and SM5210(Second Gen)

#### Check if there is a response ready (Phase3)

- Read single byte first (Phase3), that is the first byte (length indicator) of the Spv1 response frame. It indicates if there are more data to read or not.
- If the length is none-zero, move Phase 4 and read length + 1 bytes from the i2c slave. Notice that +1 is for the checksum (last byte) that length does not count it.
- If the length is zero then it means there is no response ready. In this case skip to Phase5 from Phase 3 and make a dummy read with NACK before stop condition.

#### Retrieve Spv1 frame response if exists (Phase 4)

- If length indicator is none-zero in phase 3 then read the length + 1 bytes from the slave to retrieve the whole spv1 frame.
- Move to the Phase 5 and read the last byte, in this case (checksum byte) with NACK.

#### Read last byte with Nack (Phase 5)

- Always read the last byte with NACK before the stop condition. This is to indicate that master is done reading data with the slave.
- If length is none-zero this is the checksum byte or an additional dummy read.
- If length is zero this a dummy read with NACK.

### I2C Command/Response Packet View

Following is the logic analyzer output of sending 3 bytes for CmdActivateAll command [0x01,0x83,0x84], and the received response package.

Start	Address	Write	A-ACK	Data	D-ACK	Data	D-ACK	Data	D-ACK	Stop	Dummy Read by MCU can be observed				
Start	42	Write	A-ACK	01	D-ACK	83	D-ACK	84	D-ACK	Stop					
Start	Address	Read	A-ACK	Data	D-ACK	Data	D-ACK	Data	D-ACK	Data	D-ACK	Data	D-NACK	Stop	Description
Start	42	Read	A-ACK	02	D-ACK	83	D-ACK	4E	D-ACK	D3	D-ACK	00	D-NACK	Stop	DATA NACK

Figure 4.3.1 *CmdActivateAll(SELECT\_TAG) I2C command and response packets.*

- 0x02 is the Length indicator byte of the Spv1 response frame
- 0x83 is the CmdActivateAll Command/Response ID
- 0x4E('N') is the response error code, indicating there is "No Tag"
- 0xD3 is the checksum byte, sum of the bytes starting from length to last data byte before checksum. (0x02 + 0x83 + 0x4E = 0xD3)
- 0x00 The last byte after the checksum, may or not exist depends on your environment. Some of the micro-controllers can make a dummy read when using `i2c.stop()` automatically. This byte should not be used for processing response and it can be any arbitrary value.

### I2C Packet View for No Response

Following is the logic analyzer view for the zero length response(No response)

Start	Address	Read	A-ACK	Data	D-ACK	Data	D-NACK	Stop	Description
Start	42	Read	A-ACK	00	D-ACK	00	D-NACK	Stop	DATA NACK

Figure 4.3.1 *I2C response when there is no spv1 response ready.*

- 0x00 First byte, the length indicator of the response frame.
- 0x00 Second byte is dummy read with NACK to inform master is done reading. A new sequence, [Table 4.3.1](#), is required to check if there is a ready response.

**Note:** Please see *I2C Code Example* for an implementation of the described method.

## 4.3.2 Alternative Method to Receive Response from SM5210/SM5211-SMD

The requirement for an alternative method arises because of some platforms does not provide low level I2C APIs to implement *the standard receive response method*, explained in previous topic. For example Arduino Wire Library does not provide low level APIs and it is not possible to manage `i2c_start`, `i2c_stop`, `i2c_read` and `Nack` manually. Thus, there is no way to implement the *the standard receive response method* with Arduino or similar platforms that does not provide low level I2C API.

The method introduced in this topic is to receive the response partially with two sequences (consist of two start and stop sequence) as illustrated in the tables below. This method is only valid for SM5210/SM5211-SMD, second-generation module. SM5210/SM5211-SMD supports response to be read with more than one start-stop sequence which is fine with any platform including the Arduino. However, unlike SM5210/SM5211-SMD, SM130, first-generation module, does not support partial/fragmented reading with multiple start and stop conditions. For SM130, the Spv1 response

frame needs to be read within a single start - stop sequence. Another method is introduced in the next section for SM130 modules and Arduino.

The method explained here for SM5210/SM5211-SMD is not limited only for Arduino platform (Wire library) and can be used with any platforms alternative to the *the standard receive response method*.

In this method, receiving the Spv1 response frame is done with two start-stop read sequence:

**Read Sequence - 1**

Phase 1	Phase 2	Phase 3	Phase 4
Start Condition	Address(read)	Read Spv1 Frame First Byte (length indicator) with NACK	Stop Condition

Table 4.3.2 Read Sequence 1 - Receiving length indicator byte from SM5210

**Check if there is a response ready (Phase3)**

- Read a single byte with Nack (Phase3), that is the first byte (length indicator) of the Spv1 response frame. It indicates if there are more data to read or not.
- If the length is none-zero, generate a stop condition and move to the **Read Sequence 2**.
- If the length is zero then it means there is no response ready. Generate a stop condition. You can repeat **Read Sequence 1** to check for an available response again.

**Read Sequence - 2**

		Spv1 Response frame second part without length byte		
Phase 1	Phase 2	Phase 3	Phase4	Phase5
Start Condition	Address(read)	Read remaining if exist indicated with the previous operation	Read Last Byte with NACK	Stop Condition

Table 4.3.2 Read Sequence 2 - Receiving remaining spv1 frame from SM5210

**Read remaining data (Phase3)**

- If the length indicator was none-zero on Read Sequence 1, then remaining data can be read from the slave with length + 1 byte. Notice that +1 is for the checksum(last byte) that length does not count it.
- The last byte read (checksum byte) is done with Nack and a stop condition is generated.

**I2C Command/Response Packet View**

Following is the logic analyzer output of sending 3 bytes for CmdActivateAll command [0x01,0x83,0x84], and the received response package done with two read sequences.

Start	Address	Write	A-ACK	Data	D-ACK	Data	D-ACK	Data	D-ACK	Stop		
Start	42	Write	A-ACK	01	D-ACK	83	D-ACK	84	D-ACK	Stop	<b>CmdActivateAll Command</b>	
Start	Address	Read	A-ACK	Data	D-NACK	Stop	Description					
Start	42	Read	A-ACK	02	D-NACK	Stop	DATA NACK	<b>Read Sequence 1</b>				
Start	Address	Read	A-ACK	Data	D-ACK	Data	D-ACK	Data	D-NACK	Stop	Description	
Start	42	Read	A-ACK	83	D-ACK	4E	D-ACK	D3	D-NACK	Stop	DATA NACK	<b>Read Sequence 2</b>

Figure 4.3.2 CmdActivateAll (SELECT\_TAG) I2C command and response packets with two read sequences.

## Read Sequence 1

- 0x02 is the Length indicator byte of the Spv1 response frame in Read Sequence 1. Because it is non-zero, Length + 1 (3 bytes) needs to be read in Read Sequence 2.

## Read Sequence 2

- 0x83 is the CmdActivateAll Command/Response ID
- 0x4E('N') is the response error code, indicating there is "No Tag"
- 0xD3 is the checksum byte, Sum of the bytes starting from length to last data byte before checksum. (0x02 + 0x83 + 0x4E = 0xD3)

## Example Implementation for ARM® mbed™

```

int ReadSingleResponseSM5210TwoSequence(int address7bit, unsigned char* receiveBuffer)
{
    int spv1framelength = 0;
    unsigned char checksum = 0;
    unsigned char address8bit;
    int temp = 0;

    address8bit = address7bit << 1;
    address8bit |= 0x01; // Set bit 0 - Read operation.
    i2c.read(address8bit, (char*)receiveBuffer, 1);
    spv1framelength = receiveBuffer[0];

    if (spv1framelength>0)
    {
        receiveBuffer[0] = spv1framelength;
        i2c.read(address8bit, (char*)&receiveBuffer[1], spv1framelength+1);

        checksum = receiveBuffer[0];

        for(int i=0;i<spv1framelength;i++)
        {
            checksum += receiveBuffer[1+i];
        }

        if (checksum!=receiveBuffer[spv1framelength+1])
        {
            spv1framelength = 0;
            #if defined DEBUG
            pc.printf("Checksum Error.\r\n");
            #endif
        }
    }

    #if defined DEBUG
    if (spv1framelength>0)
    {
        pc.printf("Spv1 Response Frame:");

        for(int i=0;i<spv1framelength+2;i++)
            pc.printf("%02x-", receiveBuffer[i]);
    }
    #endif
}

```

```

        pc.printf("\r\n");
    }
#endif

    return spv1framelength;
}

```

**Note:** Please see *I2C Code Example* for an implementation of the described method for Arduino.

### 4.3.3 Alternative Method to Receive Response from SM130 with Arduino

Unlike SM5210/SM5211-SMD, SM130 does not support partial/fragmented reading with multiple start and stop conditions. For SM130, the Spv1 response frame needs to be read within a single start - stop sequence.

The two methods introduced previously, *Standard Receive Response Method* and *Alternative Method to Receive Response from SM5210/SM5211-SMD* cannot be applied for SM130 and Arduino couple. This is because Arduino Wire library does not provide low level APIs, so the standard method cannot be used, additionally SM130 does not support partial/fragmented response reading so alternative method for SM5210 cannot be used for SM130.

For Arduino - SM130 couple, reading the response can be done with single start-stop sequence with the maximum amount of expected response length. Maximum expected response length depends on the command sent.

For example; CmdActivateAll(Select Tag) command, can return 3 bytes if there is 'No Tag' or can return with maximum of 12 bytes(depends on UID length of the tag). In this case, the maximum amount of expected length is 12 for CmdActivaAll command. Thus, for CmdActivateAll command response, master needs to read always 12 bytes from the slave and parse the response according to the length indicator byte of the spv1 frame.

**Note:** The method explained here is a dirty way, as it requires to know maximum amount of length to be received for each command. If a new version of Arduino Wire Library or similar I2C library supports manual handling of low level I2C APIs, such as, i2c\_start(), i2c\_stop() then please use the *Standard Receive Response Method*

		Spv1 Response frame with fixed(maximum expected) length			
Phase 1	Phase 2	Phase 3	Phase4	Phase5	
Start Condition	Address(read)	Read number of maximum expected bytes always.	Read Last Byte with NACK	Stop Condition	

Table 4.3.3 Receiving spv1 frame from SM130 in single sequence

#### Read data (Phase3 and Phase4)

- Read with a fixed length, that is the maximum expected amount of length for each command.
- The last byte read(checksum byte) is done with Nack and a stop condition is generated.
- Check the first byte in the received frame that is the length indicator, then parse the response according to length information. Omit the bytes received that is out of length indicator.

**Note:** Please see *I2C Code Example* for an implementation of the described method for Arduino and SM130 couple.

## 4.4 I2C Example - Card UID Reading with CmdActivateAll

Please check our support page if there are full code examples or libraries exist written for your platform. Code examples given here are just for illustrating the i2c communication basics explained in the previous topics. The principles and the alternate methods for different situations can be applied to any platform that supports I2C. Source code of the examples can be downloaded at our support page. Examples are given for ARM® mbed™ online compiler and Arduino platform.

### 4.4.1 Using Standard Receive Method - ARM® mbed™

Following code is tested with ARM® mbed™ online compiler and ST's NUCLEO F302R8 board. It implements *Standard Receive Response Method*. It reads the card UID by sending CmdActivateAll command and receive the response.

```
#include "mbed.h"

/*
 * This example is tested on mbed online compiler platform
 * with NUCLEO-F302R8 board
 */
//-----
// Hyperterminal configuration
// 9600 bauds, 8-bit data, no parity
//-----

#define DEBUG

/* Command ID for ACTIVATE_ALL(SELECT_TAG) */
#define CMD_ACTIVATE_ALL 0x83

Serial pc(SERIAL_TX, SERIAL_RX);
I2C i2c(I2C_SDA, I2C_SCL);

DigitalOut myled(LED1);

int CmdActivateAll()
{
    int status;
    char spv1frame[3];
    spv1frame[0] = 0x01;
    spv1frame[1] = 0x83;
    spv1frame[2] = 0x84;
    status = i2c.write(0x84, spv1frame, 3);
    return status;
    // @returns
    // 0 or non-zero - written number of bytes,
    // negative - I2C_ERROR_XXX status
}
```

Following part of the code, `int ReadSingleResponse(..)`, implements the *Standard Receive Response Method*.

```
int ReadSingleResponse(int address7bit, unsigned char* receiveBuffer)
{
    // Read the first byte of the spv1 frame.
```

```

// that is length indicator of the spv1 frame.
// if length is zero, send i2c_stop()
// if length is none-zero, receive the expected bytes

int spv1framelength = 0;
unsigned char checksum = 0;
unsigned char address8bit;

address8bit = address7bit << 1;
address8bit |= 0x01; // Set bit 0 - Read operation.
i2c.start();
i2c.write(address8bit); // 0x42->0x85 when reading from i2c slave

spv1framelength = i2c.read(true);
if (spv1framelength==0)
{
    // Done with query length byte. send i2c_stop()
    // mbed(with nucleo-F302R8 board) sends a read with
    // nack automatically when using i2c.stop();
    // Last read must be with NACK.
    // If your hardware system does not make a read with NACK when
    // using i2c_stop(), then you can do it manually by uncommenting
    // i2c.read(false); // Read last byte with nack.
    i2c.stop();
}
else if (spv1framelength!=0)
{
    // There is a spv1 response ready.
    receiveBuffer[0] = spv1framelength;
    checksum = receiveBuffer[0];

    for(int i=0;i<spv1framelength;i++)
    {
        receiveBuffer[1+i] = i2c.read(true);
        checksum += receiveBuffer[1+i];
    }

    // mbed(with nucleo-F302R8 board) automatically makes
    // a read with nack when i2c.stop() is used.
    // for consistency with other systems we will read
    // checksum byte(last byte) with NACK
    receiveBuffer[spv1framelength+1] = i2c.read(false);
    i2c.stop();

    if (checksum!=receiveBuffer[spv1framelength+1])
    {
        spv1framelength = 0;
        #if defined DEBUG
        pc.printf("Checksum Error.\r\n");
        #endif
    }
}

#if defined DEBUG
if (spv1framelength>0)
{
    pc.printf("Spv1 Response Frame:");
}
}

```

```

        for(int i=0;i<spv1framelength+2;i++)
        pc.printf("%02x-",receiveBuffer[i]);

        pc.printf("\r\n");
    }
#endif

return spv1framelength;
}

```

```

int TryReadResponse(int address7bit,
                   unsigned char *receiveBuffer,
                   int read_attempt)
{
    unsigned char spv1framelength = 0;
    int i = 0;

    // Try number of given read_attempt to poll slave if there is
    // a spv1 frame ready.

    spv1framelength = 0;
    while(spv1framelength==0)
    {
        i++;

        #if defined DEBUG
        pc.printf("Polling Attempt: ");
        pc.printf("%d \r\n",i);
        #endif

        // Use delay for each polling to prevent frequent distruption
        // of SM130 operation. If you use a lower delay value,
        // you will see polling attempt number will increase.
        // Recommended minimum value is 20ms for SM130.
        wait(0.02);

        spv1framelength = ReadSingleResponse(address7bit, receiveBuffer);

        if (spv1framelength>0)
        {
            // We have spv1 response frame
        }

        if (i==read_attempt)
            break;
    }

    return spv1framelength;
}

```

```

int main()
{
    int status;
}

```

```

unsigned char receiveBuffer[32];
unsigned char spv1framelength;
int address7bit = 0x42;
unsigned char commandID;

// int i = 1;
// pc.printf("Hello World !\n");
while(1) {
    wait(0.5);
    myled = !myled;
    pc.printf("Sending CmdActivateAll \r\n");
    status = CmdActivateAll();
    spv1framelength = TryReadResponse(address7bit, receiveBuffer, 5);

    if (spv1framelength!=0)
    {

        // There is a valid spv1 response.
        // You can parse spv1 frame response here.
        // For CmdActivateAll, we have No Tag information or Card UID.

        commandID = receiveBuffer[1];
        if (commandID==CMD_ACTIVATE_ALL)
        {
            if (spv1framelength==2)
            {
                pc.printf("Response Error Code: %02x \r\n", receiveBuffer[2]);
                if (receiveBuffer[2]=='N')
                {
                    pc.printf("No Tag \r\n");
                }
            }
            else if ((spv1framelength==6) || (spv1framelength==9))
            {
                int uidlength = spv1framelength - 2;
                pc.printf("UID Length: %d", uidlength);
                pc.printf(" , UID:");

                // Card UID is reported reverse. (LSB first)
                for (int i = 0; i<uidlength; i++)
                {
                    pc.printf("%02x-", receiveBuffer[2 + uidlength - i]);
                }
                pc.printf("\r\n");

            }
            else
            {
                pc.printf("Unknown response \r\n");
            }
        }
    }
    else
    {
        // Response is not for CmdActivateAll.
        pc.printf("Response does not belong CmdActiveAll \r\n");
    }
}

```

```

else
{
pc.printf("spv1 response not ready");
}

}
}

```

#### 4.4.2 Using Alternative Methods for Reading Response - Arduino

Following example is written for Arduino, implementing I2C communication with SM5210/SM5211-SMD and SM130. It reads the card UID by sending CmdActivateAll command and receive the response.

It implements *Alternative Method to Receive Response from SM5210/SM5211-SMD* and *Alternative Method to Receive Response from SM130*

Please comment/uncomment #defines for SM130 and SM5210 implementation.

```

#include <Wire.h>

/*
 * This example project only illustrates basics of the I2C
 * communication with SM130 and SM5210 explained in SPV1
 * I2C COMMUNICATION document/chapter. It provides an
 * example to read card UID using CmdActivateAll(SELECT_TAG)
 * command. It does not provide a full command set library
 * or example. Check if there is a complete library is
 * written for your environment for faster integration.
 *
 * SM5210 and SM130 have slight differences in reading
 * the I2C response. SM5210 is backwards compatible
 * but alternate method used for SM5210 for reading the
 * response is more straight forward. Please read
 * SPV1 I2C COMMUNICATION document(or chapter) for more details
 *
 * Also notice that, logic for reading response can be
 * implemented differently in another system depends on the
 * provided APIs. For example Arduino Wire library does not
 * provide low level APIs for I2C. For SM130, it is the only way
 * to read i2c buffer with the maximum amount of data expected
 * for the command. This can be resolved differently with another
 * system if low level I2C APIs are provided such as
 * i2c_start, i2c_stop,i2c_write, i2c_read.
 *
 * If you are using SM5210 uncomment the SM130 and
 * comment the SM5210 defines.
 *
 */

#define DEBUG

/* comment/uncomment either SM130 or SM5210 */
// If SM130 is used, SM5210 will work as well.
// However SM5210 method to read response is more elegant.
// You dont need to determine maximum number of length for response

```

```
// for SM5210. Max expected response varies for each command.
// SM130 wont work if SM5210 is defined.

// #define SM130
#define SM5210

#if defined (SM130) && (defined SM5210)
#error "Define only one target!"
#endif

#if defined (SM130) || (defined SM5210)
/* Do nothing */
#else
#error "Target not defined"
#endif

/* Command ID for ACTIVATE_ALL(SELECT_TAG) */
#define CMD_ACTIVATE_ALL 0x83

void setup() {
  // put your setup code here, to run once:
  pinMode(LED_BUILTIN, OUTPUT);
  Serial.begin(9600);
  Wire.begin(); // join i2c bus (address optional for master)
}
```

```
byte CmdActivateAll()
{
  byte status;
  // Use 0x84 when writing to 0x42 I2C address.
  // Shift 0x42 to left 1 bit and clear last bit.
  Wire.beginTransmission(0x42);

  // Length of the frame to be sent including
  // command and databytes, excluding checksum.
  Wire.write(byte(0x01));

  // Command ID (0x83 -> Activate ALL(SELECT TAG))
  // (This command has no data)
  Wire.write(byte(CMD_ACTIVATE_ALL));

  // Checksum.
  Wire.write(byte(0x84));

  // stop transmitting
  status = Wire.endTransmission();
  return status;

  // 0:success
  // 1:data too long to fit in transmit buffer
  // 2:received NACK on transmit of address
  // 3:received NACK on transmit of data
  // 4:other error
}
```

Following part of the code, byte `ReadSingleResponseSM130(...)`, implements the *Alternative Method to*

*Receive Response from SM130.*

```

byte ReadSingleResponseSM130(int address,
                             int max_length_for_response,
                             byte *receiveBuffer)
{
    bool status = false;
    byte available_length;
    byte spv1framelength = 0;
    byte checksum = 0;

    // Notice that the following method for SM130 can also work
    // with SM5210. Thus we can say SM5210 I2C is back compatible
    // with SM130. However SM5210 solution is straight forward and
    // more elegant.
    // It is recommended to use this method only for SM130 and use
    // better and cleaner code for SM5210.

    // SM130(first generation module) provides Spv1 response frame
    // to be read only with one stop condition unlike the
    // SM5210(second generation modules).Thus it is required to
    // read the first byte of the spv1 frame(length indicator)
    // without stop condition.
    // However Arduino Wire Library does not provide low level API
    // for start and stop conditions.
    // Therefore we read spv1 frame from the slave with the maximum
    // expected length so we have the whole spv1 frame response with
    // a single stop condition, once at a time.
    // Then we will process the received spv1 frame response
    // according to the frame length information
    // (first byte of the spv1 frame response)

    // Read from slave with the maximum expected length of
    // command's response
    available_length = Wire.requestFrom(address, max_length_for_response);
    for(int i=0;i<available_length;i++)
    {
        if (Wire.available())
        {
            // Collect the spv1 response in the buffer.
            // First byte is the length indicator of the
            // spv1 response frame.
            receiveBuffer[i] = Wire.read();
        }
        else
        {
            // This should never happen
            // Serial.print("Unexpected condition \r\n");
            return 0;
        }
    }

    // We have whole frame read from the slave defined with
    // the max_length_for_response. However it does not indicate
    // that there is a spv1 response ready.
    // If there is a ready spv1 response, then
    // the first byte(length byte) of the frame needs to be none-zero,
    // indicating number of following bytes in the frame excluding the

```

```

// checksum byte.

if ((available_length>0) && (receiveBuffer[0]!=0))
{
    spv1framelength = receiveBuffer[0] ;
    // Notice that Total frame length is +2 for the receiveBuffer
    // + 1(for length byte) + 1(for checksum)

    #if defined DEBUG
        Serial.print("Spv1 Response Frame:");
        for(int i=0;i<spv1framelength+2;i++)
        {
            Serial.print(receiveBuffer[i],HEX);
            Serial.print("-");
        }
        Serial.print("\r\n");
    #endif

    // checksum control
    checksum = 0;
    for(int i=0;i<spv1framelength+1;i++)
        checksum += receiveBuffer[i];

    if (checksum!=receiveBuffer[spv1framelength+1])
    {
        spv1framelength = 0;
        #if defined DEBUG
            Serial.print("Checksum error\r\n ");
        #endif
    }

}
else
    spv1framelength = 0; //There is no spv1 frame response ready.

return spv1framelength;
}

```

Following part of the code, byte `ReadSingleResponse(...)`, implements the *Alternative Method to Receive Response from SM5210*.

```

byte ReadSingleResponse(int address,byte *receiveBuffer)
{
    // This method works for SM5210(Second Generation Modules) only.
    // This method does not work for SM130(First Generation Modules)

    byte spv1framelength = 0;
    byte checksum = 0;

    // Read 1 byte from slave first within start-stop condition.
    // This is the spv1 frame response length indicator byte.
    // If length indicator byte is none-zero then
    // read available data with another start-stop read sequence
    // Unlike SM130, SM5210 supports reading remaining data after
    // a stop condition.

```

```

Wire.requestFrom(address, 1);

if (Wire.available())
{
    // First byte of the spv1 frame is length indicator.
    spv1framelength = Wire.read();
}
else
    spv1framelength =0;

// If spv1framelength is none-zero. We have spv1 frame ready.
// Read the rest of the data determined with the
// spv1framelength + 1
// +1 is to read the cheskum byte as well.
if (spv1framelength!=0)
{
    // We have spv1 response ready.
    receiveBuffer[0] = spv1framelength;
    Wire.requestFrom(address, spv1framelength+1);
    // +1 is for checksum.

    checksum = receiveBuffer[0];
    for(int i=0;i<spv1framelength+1;i++)
    {
        if (Wire.available())
        {
            receiveBuffer[i+1] = Wire.read();
            checksum += receiveBuffer[i+1];
        }
        else
        {
            // If there is no data available,
            // then we should
            // have the length information
            // wrongly. Possibly data was
            // not synchronized with
            // expected frame.
            // System will correct it self
            // with the next command or read.
            Serial.print("length mismatch \r\n");
            return false;
        }
    }

    #if defined DEBUG
    Serial.print("Spv1 Response Frame:");
    for(int i=0;i<spv1framelength+2;i++)
    {
        Serial.print(receiveBuffer[i],HEX);
        Serial.print("-");
    }
    Serial.print("\r\n");
    #endif

    // Checksum control
    // We substract last byte(checksum) from calculated sum.
    checksum = checksum - receiveBuffer[spv1framelength+1];

```

```

    if (checksum!=receiveBuffer[spv1framelength+1])
    {
        spv1framelength = 0;
        #if defined DEBUG
            Serial.print("Checksum error\r\n ");
        #endif
    }

}

return spv1framelength;
}

```

```

#if defined SM5210
byte TryReadResponse(int address,byte *receiveBuffer,byte read_attempt)
#else
byte TryReadResponseSM130(int address, int max_length_for_response,
                           byte *receiveBuffer,byte read_attempt)
#endif
{
    byte spv1framelength = 0;
    int i = 0;

    // Try number of given read_attempt to poll slave if there is a
    // spv1 frame ready.

    spv1framelength = 0;
    while(spv1framelength==0)
    {
        i++;

        #if defined DEBUG
            Serial.print("Polling Attempt: ");
            Serial.print(i);
            Serial.print("\r\n");
        #endif

        // Use delay for each polling to prevent frequent disruption
        // of SM130 operation.
        // If you use a lower delay value, you will see polling attempt
        // number will increase.
        // Recommended minimum value is 20ms for SM130.
        delay(20);

        #if defined SM5210
            spv1framelength =ReadSingleResponse(address,receiveBuffer);
        #else
            spv1framelength = ReadSingleResponseSM130(address,
                                                       max_length_for_response,
                                                       receiveBuffer);
        #endif

        if (spv1framelength>0)
        {
            // We have spv1 response frame
        }
    }
}

```

```

    if (i==read_attempt)
        break;
}

return spv1framelength;
}

```

```

void loop() {
    // put your main code here, to run repeatedly:
    byte status;
    byte address = 0x42;
    byte commandID;
    byte spv1framelength;
    byte receiveBuffer[32];
    digitalWrite(LED_BUILTIN, HIGH);
    delay(500);

    // Send CmdActivateAll (SELECT_TAG) command
    status = CmdActivateAll();
    // Try reading response.

    #if defined SM5210
        spv1framelength = TryReadResponse(address, receiveBuffer, 5);
    #else // SM130
        // Max expected response for CmdActivateAll is 12 bytes.
        // 1 Byte Length, 1 Byte Command, 1 Byte TagType,
        // 8 Byte UID(Max) ,1 Byte Checksum
        spv1framelength = TryReadResponseSM130(address, 12, receiveBuffer, 5);
    #endif

    if (spv1framelength!=0)
    {

        // There is a valid spv1 response.
        // You can parse spv1 frame response here.
        // For cmd ActivateAll, we have No Tag information or Card UID.

        commandID = receiveBuffer[1];
        if (commandID==CMD_ACTIVATE_ALL)
        {
            if (spv1framelength==2)
            {
                Serial.print("Response Error Code:");
                Serial.print(receiveBuffer[2], HEX);
                Serial.print("\r\n");
                if (receiveBuffer[2]=='N')
                {
                    Serial.print("No Tag \r\n");
                }
            }
            else if ((spv1framelength==6) || (spv1framelength==9))
            {
                int uidlength = spv1framelength - 2;
                Serial.print("UID Length:");
            }
        }
    }
}

```

```
Serial.print(uidlength);
Serial.print(" , UID:");

// Card UID is reported reverse. (LSB first)
for (int i = 0; i<uidlength; i++)
{
    Serial.print(receiveBuffer[2 + uidlength - i],HEX);
    Serial.print("-");
}
Serial.print("\r\n");

}
else
{
    Serial.print("Unknown response \r\n");
}
}
else
{
    // Response is not for CmdActivateAll.
    Serial.print("Response does not belong CmdActiveAll \r\n");
}
}
else
{
    Serial.print("Cant get any spv1 response");
}

digitalWrite(LED_BUILTIN, LOW);
delay(500);

}
```

## TRADEMARKS

- MIFARE® is a registered trademark of NXP B.V. and is used under license.
- NTAG® is a trademark of NXP B.V
- ARM® and Cortex are registered trademarks of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved.
- ARM® mbed™ is a registered trademarks of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved.
- ARDUINO® is a registered trademark of Arduino AG



## DOCUMENT REVISION HISTORY

**Version 1.0.0 (02 May 2017)**

Initial release.