



Introduction

This document describes the physical connections between an ST microcontroller and the SN260. Recommendations are provided concerning the practices and procedures for designing, developing, and testing the SPI protocol on a ST microcontroller that is interfaced to the SN260 (for allowing the ST core to communicate with the SN260 ZigBee® network processor). The descriptions and recommendations are also designed to illustrate a work flow that should help produce a solid SPI protocol module for an ST microcontroller.

This document assumes that the reader is familiar with the *SN260 Datasheet* and, in particular, has gotten an overall understanding of the SN260 SPI protocol, the terminology used, the basic sequences described, and the transaction examples.

Contents

- 1 Physical interface (how to connect ST microcontrollers to the SN260) 3**
 - 1.1 ST microcontrollers to SN260 pin connections for the SPI communication 3
- 2 Verifying the SPI protocol 4**
 - 2.1 First transaction: SPI protocol version 4
 - 2.1.1 SPI protocol version transaction with an SN260 reset 4
 - 2.1.2 SPI protocol version transaction without an SN260 reset 4
 - 2.2 Second transaction: SPI status 5
 - 2.2.1 SPI status transaction 5
 - 2.2.2 Performing a hard reset 5
 - 2.3 Third transaction 6
 - 2.3.1 EZSP Version command transaction 6
 - 2.3.2 EZSP NOP command transaction 7
- 3 SPI protocol considerations 8**
 - 3.1 The wait section 8
 - 3.2 Clock the SPI (polling for data) 8
 - 3.3 Interrupt on the falling edge of nHOST_INT 9
 - 3.4 Inter-command spacing 9
 - 3.5 Asynchronous signaling 9
 - 3.6 Waking the SN260 10
 - 3.7 Error conditions 10
 - 3.8 Error bytes 10
 - 3.9 Timeouts 11
 - 3.10 Interfacing the EZSP and the SPI protocol 11
- 4 Revision history 12**

1 Physical interface (how to connect ST microcontrollers to the SN260)

This section describes the connections between the ST microcontroller and the SN260 micro and how to verify them. These connections allow the two micros to communicate through the standard serial interface (SPI).

1.1 ST microcontrollers to SN260 pin connections for the SPI communication

Table 1 describes an example of ST microcontroller connections to the SN260 used for the ZigBee REva kit.

Table 1. Summary of connections

Pin name	STR71x pins	ST7Lite3 pins	STR750 pins	STR912F pins	STM32F103x pins
MOSI	P0.5	PB.3	P0.07	P5.5	PA7
MISO	P0.4	PB.2	P0.06	P5.6	PA6
SCLK	P0.6	PB.1	P0.05	P5.4	PA5
nSSEL	P1.6	PA.4	P2.14	P6.4	PC10
HOST_INT	P0.1	PA.1	P1.10	P6.1	PC7
WAKE	P0.0	PA.0	P2.10	P6.0	PC6
RSTB	P1.5	PA3	P2.13	P6.3	PC9

2 Verifying the SPI protocol

Some very basic transactions can be performed to verify the SPI communication between the ST microcontroller and the SN260 micro. This section describes some of these basic transactions.

2.1 First transaction: SPI protocol version

Obtaining the SPI protocol version is a compact, simplified, and special transaction that illustrates a full transaction. Being able to properly obtain the SPI protocol version not only verifies five of the seven interface pins (MOSI, MISO, SCLK, nSSEL, and nHOST_INT), but it is also useful as a test for verifying that the SN260 is active and that the SPI protocol code being implemented on the host is compatible with the firmware on the SN260. Use this transaction to verify the connection with the SN260 during the host's boot sequence.

Before worrying about creating a generic SPI protocol module or handling error cases, it is best to get a feel for a transaction in the simplest way possible. The following steps should result in your first transaction and can be explicitly coded in a test function for reference.

2.1.1 SPI protocol version transaction with an SN260 reset

The following steps begin by resetting the SN260 to guarantee that it is in a known state. The SN260 resetting is an error condition described in the SN260 Datasheet, and results in the first transaction performed after a reset returning the reset error. These steps describe receiving this reset error instead of the SPI protocol version.

1. Assert nRESET and release to reset the SN260 and guarantee it is in a known state
2. Wait for nHOST_INT to assert, which indicates that the SN260 is active
3. Assert nSSEL to begin a transaction
4. Transmit 0x0A
5. Transmit 0xA7
6. Continually transmit 0xFF until the byte received is not 0xFF. The first byte received that is not 0xFF will be 0x00
7. Transmit 0xFF while receiving 0x02
8. Transmit 0xFF while receiving 0xA7
9. De-assert nSSEL to finish the transaction.

2.1.2 SPI protocol version transaction without an SN260 reset

1. Assert nSSEL to begin a transaction
2. Transmit 0x0A
3. Transmit 0xA7
4. Continually transmit 0xFF until the byte received is not 0xFF. The first byte received that is not 0xFF will be 0x81
5. Transmit 0xFF while receiving 0xA7
6. De-assert nSSEL to finish the transaction

2.2 Second transaction: SPI status

The SPI Status transaction is very similar to the SPI protocol version transaction (detailed previously). Like the SPI protocol version transaction, this transaction provides a simple example of interaction with the SN260. This is recommended as a test transaction to verify the connection with the SN260 during the host's boot sequence.

2.2.1 SPI status transaction

1. Assert nSSEL to begin a transaction
2. Transmit 0x0B
3. Transmit 0xA7
4. Continually transmit 0xFF until the byte received is not 0xFF. The first byte received that is not 0xFF will be 0xC1
5. Transmit 0xFF while receiving 0xA7
6. De-assert nSSEL to finish the transaction.

2.2.2 Performing a hard reset

When the host resets, it is far simpler to reset the SN260 and begin from a known state as opposed to trying to recover and resync with the previous (unknown) state of the SN260. The recommended procedure when the host resets is to perform a Hard Reset of the SN260 during bootup. A Hard Reset is defined as the following sequence:

1. Toggle nRESET (active low) to reset the SN260
2. Wait for nHOST_INT to assert, which indicates that the SN260 is active
3. Perform an SPI protocol version transaction and verify that the Response from the SN260 is the SN260 Reset error condition
4. Perform an SPI protocol version transaction and verify that the SPI protocol version number is as expected
5. Perform an SPI status transaction and verify that the SN260 is "alive".

The purpose of performing this hard reset on bootup is threefold.

- By guaranteeing that the SN260 is freshly booted, just like the host, the host can proceed with standard node and network initialization instead of consuming extra code space just trying to determine what state the SN260 was left in.
- Since the SN260 generates the SN260 Reset error, which will override any legitimate transaction Response, the Hard Reset can acknowledge this planned and expected error condition so that the EZSP or full application does not have to implement special handling. Therefore, whenever an SN260 Reset error is experienced outside of a Hard Reset, it can be treated as a true unexpected error condition.
- The SPI protocol version and SPI Status transactions are specialized transactions not implemented or used by the normal EZSP. These transactions are intended to be utility devices that allow the host to perform a simple "handshake" with the SN260. This handshake not only verifies that the SN260 is alive and available to the EZSP, but also that the SPI protocol implemented in the SN260 is compatible with the SPI protocol implemented on the host.

2.3 Third transaction

Before implementing a generic SPI protocol on the host, it is explicitly recommended to code a third transaction to provide exposure to an EZSP Frame and the format of the data involved with an EZSP Frame.

2.3.1 EZSP Version command transaction

If using the EmberZNet™ 3.x stack the third transaction is the VERSION command. The VERSION command is required to be the first EZSP command issued to the SN260. It exercises the code path all the way through the EM260 firmware. Therefore, this command is useful not only for verifying that the EZSP is active, but also for illustrating the implementation of an EZSP transaction.

1. Assert nSSEL to begin a transaction
2. Transmit 0xFE
3. Transmit 0x04
4. Transmit 0x00
5. Transmit 0x00
6. Transmit 0x00
7. Transmit 0x02
8. Transmit 0xA7
9. Continually transmit 0xFF until the byte received is not 0xFF. The first byte received that is not 0xFF will be 0xFE
10. Transmit 0xFF while receiving 0x07
11. Transmit 0xFF while receiving 0x00
12. Transmit 0xFF while receiving 0x80
13. Transmit 0xFF while receiving 0x00
14. Transmit 0xFF while receiving 0x02
15. Transmit 0xFF while receiving 0x02
16. Transmit 0xFF while receiving 0x11 (Note that this value is the build number and may vary.)
17. Transmit 0xFF while receiving 0x30
18. Transmit 0xFF while receiving 0xA7
19. De-assert nSSEL to finish the transaction

2.3.2 EZSP NOP command transaction

If using the EmberZNet™ 2.5.x stack the third transaction is the NOP command. The NOP command does not perform any actual function but exercises the code path all the way through the SN260 firmware. Therefore, this command is useful not only for verifying that the EZSP is active, but for illustrating the implementation of an EZSP transaction.

1. Assert nSSEL to begin a transaction
2. Transmit 0xFE
3. Transmit 0x02
4. Transmit 0x00
5. Transmit 0x05
6. Transmit 0xA7
7. Continually transmit 0xFF until the byte received is not 0xFF. The first byte received that is not 0xFF will be 0xFE
8. Transmit 0xFF while receiving 0x02
9. Transmit 0xFF while receiving 0x80
10. Transmit 0xFF while receiving 0x05
11. Transmit 0xFF while receiving 0xA7
12. Deassert nSSEL to finish the transaction.

3 SPI protocol considerations

3.1 The wait section

Transmitting a command is a basic operation that requires the Slave Select to be asserted and the command bytes to be *dumped* on the SPI in the most convenient method available (such as using a `for()` loop over a manual write, an interrupt-driven write, or a DMA). Once the first byte of the Response is received and the transaction has moved into the Response Section, receiving a Response is a basic, three-step operation: decode the first two bytes to determine the length of the Response, receive that precise number of bytes, and then de-assert Slave Select.

How the wait section is implemented and handled requires some careful consideration of the two techniques available: clock the SPI (also known as polling on the SPI or polling for data) and interrupt on the falling edge of `nHOST_INT`.

3.2 Clock the SPI (polling for data)

The simplest and most straightforward method for determining when a Response is ready is to continually clock the SPI until the SN260 transmits a byte other than `0xFF`. When the host “clocks the SPI,” the host should simply transmit `0xFF`, since transmitting `0xFF` is considered an idle line.

The main advantage of polling for data is that the simplicity of polling requires very little code space, and in most cases this can be implemented using either a `while()` or a `do{}while` loop.

The disadvantage of polling for data is that it can block other communication. Since transactions must occur in serial (meaning a transaction must complete before another transaction can begin), the blocking nature of polling for data is usually only an issue if the host needs to perform tasks not related to EmberZNet.

For example, if the host captures a button press and must send a message over the network in response to the button press, blocking in a polling loop is not a critical issue because of the serial nature of the transaction. Conversely, if the host must periodically take an ADC measurement and perform calculations based on the measurement, then blocking in a polling loop might not be desirable.

Because the host is the SPI Master, there are essentially no timing requirements that dictate when or how often the host should clock the SPI (the most important requirement is to keep transactions moving quickly so that messages do not back up in the SN260's buffers). Therefore, the host can clock the SPI at its convenience, which means that a developer can choose to implement the simplest solution possible and sit in a `while()` loop waiting for a response. The developer can also choose a more advanced solution: for the host to poll periodically for a response while allowing other tasks to execute on the host. Knowing that the wait sections of many transactions can be milliseconds long, the developer may decide to clock the SPI and check for a response only once every millisecond.

It is recommended to choose the simplest solution possible in the context of the host's resources and other requirements. During development, a simple blocking `while()` loop is a good starting point that can be expanded and customized as development progresses.

3.3 Interrupt on the falling edge of nHOST_INT

As detailed and illustrated in the SN260 Datasheet, the falling edge of the signal nHOST_INT indicates that a Response is ready when the falling edge occurs while Slave Select is asserted. Instead of clocking the SPI (either by completely blocking or periodically polling) and waiting for a response, the host can be configured to interrupt on the falling edge of nHOST_INT. Once the host sees a falling edge on nHOST_INT, it must still clock the SPI until data other than 0xFF is received.

The main advantage to interrupts on nHOST_INT is the ability of the host to perform other tasks while waiting for a response. Remember, since a new transaction cannot begin until the previous transaction has completed, be careful not to accidentally overlap transactions.

A disadvantage of interrupts on nHOST_INT is that there is the potential for accidentally starting a new transaction before the previous transaction has completed.

Note: The host should not poll on the level of the nHOST_INT signal. Despite nHOST_INT remaining low until the host performs an action, only the falling edge of nHOST_INT can be trusted to properly indicate data. The SN260 will carefully schedule the falling edge of nHOST_INT, but due to latency it cannot guarantee exactly when the nHOST_INT signal will return to idle after the host performs an action.

3.4 Inter-command spacing

The inter-command spacing is a simple time requirement needed to guarantee that the SN260 has finished processing a transaction and is ready to accept a new transaction. The inter-command spacing is currently defined as at least 1ms between the rising edge of Slave Select (ending transaction) and the falling edge of Slave Select (starting transaction). If the host is capable of blocking for 1ms, the simplest solution is to simply burn CPU cycles for 1ms after de-asserting Slave Select. Since burning CPU cycles for 1ms is often undesirable, it is recommended using a simple timer. By setting or starting a timer when Slave Select is de-asserted, the host can perform other tasks during the inter-command spacing. If a timer is used, the host must guarantee that any and all attempts at starting a new transaction are either blocked or stalled until the timer has expired. Once the timer has expired, the host may assert Slave Select and begin a new transaction.

3.5 Asynchronous signaling

As noted in the section Interrupt on the falling edge of nHOST_INT, this falling edge is used to indicate data availability. When nHOST_INT is asserted outside of a transaction (Slave Select is idle), the assertion means there is asynchronous data waiting in the SN260 for the host. Remember, the host should not poll on the level of the nHOST_INT signal. Instead, the host should assign an interrupt to nHOST_INT and use the falling edge (the interrupt) to set a flag or some similar marker. This way, the EZSP implementation on the host can regularly poll on the flag outside of the interrupt context and trigger the EZSP Callback command.

For more advanced functionality, you can connect nHOST_INT to a pin that is capable of waking the host from sleep, and therefore enter a low power mode, while waiting for any incoming data, like a normal asynchronous callback.

Note: Care must be taken when enabling an interrupt on nHOST_INT so that the proper piece of code is executed. nHOST_INT is capable of indicating three different situations (wake, callback, and response), and these situations are best indicated by the current state of the nSSEL and nWAKE pins. Refer to the SN260 Datasheet for sample waveforms showing these signals together.

3.6 Waking the SN260

Waking the SN260 should be a straightforward implementation that only requires you to choose between a polling or an interrupt mechanism for knowing when the SN260 is ready (much like the rest of the SPI protocol). After asserting the nWAKE signal, the host should either poll for a falling edge of nHOST_INT or set up for an interrupt on the falling edge. As soon as the edge is seen, the host should de-assert nWAKE and continue operating the EZSP as desired.

The only word of caution here though, as mentioned earlier about interrupting on nHOST_INT, is to make sure the proper piece of code gets triggered in response and to not perform further EZSP operations inside of interrupt context.

3.7 Error conditions

The error conditions encountered by the host are exactly that: errors. These errors are not meant to be encountered in a mature product and are primarily used as a development and debugging aid. If the host experiences an error condition, chances are the host and the SN260 have become out of sync, and the code needed to recover would be exceptionally error prone. Therefore, it is reasonable for the host to treat all error conditions or timeouts in the same way as asserts, and simply reset both the host and the SN260.

There is one common exception to this rule: When the host intentionally resets the SN260 (for example, as described in the section performing a Hard Reset), the host must expect the SN260 Reset error condition to occur on the next transaction. This error condition should be observed and discarded as expected and normal.

Note: The application must be careful not to interfere with any operation that loads firmware onto the SN260 (e.g., bootloading). The recommended practice is for the host to have access to and control of the SN260's nRESET signal, and to toggle nRESET if an error condition occurs. When the SN260 is being loaded with new firmware, it will not be capable of responding to the host; the host may think the SN260 is unresponsive and attempt to reset it, which will disrupt the loading of new firmware. You should consider the best method to avoid resetting the SN260 in this situation. Some options include:

- Putting the application in some mode where it leaves the SN260 alone.*
- Holding the host in reset, bootloader, or some other innocuous mode.*
- Disabling the host's access to the nRESET line on the SN260.*
- Physically disconnecting nRESET.*

3.8 Error bytes

As described in the section Special Response Bytes in the SN260 Datasheet, there are four SPI Bytes that indicate error conditions. When implementing the code to receive a Response from the SN260, the host must be capable of parsing the SPI Byte as soon as possible for any of these error conditions. The host must continue to receive the entire error before de-asserting Slave Select and processing the error. With the exception of an intentional SN260 Reset error condition, the host should report, through a print or other simple method, these four errors to the developer for debugging purposes, but should ultimately result in an assert or similar reset mechanism.

3.9 Timeouts

There are only two timeouts the host can experience: wait section and wake handshake. Just like the Error Bytes, if either of these timeouts occur, the application should report them to the developer for debugging purposes, but should ultimately result in an assert or similar reset mechanism.

The timeouts are best measured using a timer, but if necessary the host can simply burn a known amount of CPU cycles while waiting for either normal operation to resume or the limit of allowable CPU cycles. For the wait section timeout, the time is measured from the end of the last byte transmitted in the Command to the start of the first byte received that is not 0xFF. For the wake handshake timeout, the time is measured from the falling edge of nWAKE to the falling edge of nHOST_INT.

3.10 Interfacing the EZSP and the SPI protocol

Due to the serial nature of the EZSP (that is, transactions must occur in sequence instead of overlapping), it is recommended that the EZSP interfaces into the SPI protocol through a polling driven mechanism. For example, after calling a function `sendCommand()`, the EZSP could continually call a function `pollForResponse()`. Otherwise, the EZSP implementation should be carefully coded to prevent the host from accidentally overlapping transactions.

If the host's SPI protocol is implemented using interrupts, the host should be careful to never perform a transaction inside of an interrupt context. This is especially important because a transaction or a wake handshake could require up to 200ms or 10ms respectively.

4 Revision history

Table 2. Document revision history

Date	Revision	Changes
13-Feb-2007	1	Initial release.
24-May-2007	2	Updated Section 2.3: Third transaction on page 6 .
4-Jun-2007	3	Minor editing on cover page.
3-Dec-2007	4	Added STM32F103x pins and reviewed comments for Table 1: Summary of connections on page 3 .

Please Read Carefully:

Information in this document is provided solely in connection with ST products. STMicroelectronics NV and its subsidiaries ("ST") reserve the right to make changes, corrections, modifications or improvements, to this document, and the products and services described herein at any time, without notice.

All ST products are sold pursuant to ST's terms and conditions of sale.

Purchasers are solely responsible for the choice, selection and use of the ST products and services described herein, and ST assumes no liability whatsoever relating to the choice, selection or use of the ST products and services described herein.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted under this document. If any part of this document refers to any third party products or services it shall not be deemed a license grant by ST for the use of such third party products or services, or any intellectual property contained therein or considered as a warranty covering the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

UNLESS OTHERWISE SET FORTH IN ST'S TERMS AND CONDITIONS OF SALE ST DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE USE AND/OR SALE OF ST PRODUCTS INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION), OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS EXPRESSLY APPROVED IN WRITING BY AN AUTHORIZED ST REPRESENTATIVE, ST PRODUCTS ARE NOT RECOMMENDED, AUTHORIZED OR WARRANTED FOR USE IN MILITARY, AIR CRAFT, SPACE, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS, NOR IN PRODUCTS OR SYSTEMS WHERE FAILURE OR MALFUNCTION MAY RESULT IN PERSONAL INJURY, DEATH, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE. ST PRODUCTS WHICH ARE NOT SPECIFIED AS "AUTOMOTIVE GRADE" MAY ONLY BE USED IN AUTOMOTIVE APPLICATIONS AT USER'S OWN RISK.

Resale of ST products with provisions different from the statements and/or technical features set forth in this document shall immediately void any warranty granted by ST for the ST product or service described herein and shall not create or extend in any manner whatsoever, any liability of ST.

ST and the ST logo are trademarks or registered trademarks of ST in various countries.

Information in this document supersedes and replaces all information previously supplied.

The ST logo is a registered trademark of STMicroelectronics. All other names are the property of their respective owners.

© 2007 STMicroelectronics - All rights reserved

STMicroelectronics group of companies

Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

www.st.com

