ember

Application Note 5067
23 September 2010
120-5067-000A

Bringing Up New Microcontrollers

For a Network Co-Processor and EmberZNet 4.x

This document describes recommended practices and procedures for designing, developing, and testing the SPI Protocol on a new microcontroller that is interfaced to a Network Co-Processor (NCP) and using EmberZNet 4.x stack software. The descriptions and recommendations illustrate a work flow that should help produce a solid SPI Protocol module for a new host microcontroller.

Caution: This application note applies only to EmberZNet 4.x.

This document assumes that you have already read the *EM260 Datasheet* (120-0260-000). You should have an overall understanding of the SPI Protocol, the terminology used, the basic sequences described, and the transaction examples.

Note: While the SPI Protocol is available on the EM260, EM351, and EM357 chips, the *EM260 Datasheet* is the primary source for SPI Protocol documentation. For detailed timing diagrams and waveforms, refer to the *EM260 Datasheet*.

For debugging, a logic analyzer—preferably with at least seven channels—is highly recommended for examining the state of and the transitions on the NCP interface.

Contents

Physical Interface 2
First Transaction: SPI Protocol Version 3
Second Transaction: SPI Status 4
Performing a Hard Reset 4
Third Transaction: EZSP Version Command 5
The Wait Section 6
Inter-Command Spacing 8
Asynchronous Signaling 8
Waking the NCP 8
Error Conditions 8
Interfacing the EZSP and the SPI Protocol 9

Ember Corporation
47 Farnsworth Street
Boston, MA 02210
+1 (617) 951-0200
www.ember.com

Physical Interface

This section describes the SPI Protocol pin connections and how to verify them.

SPI Protocol Pin Connections

The physical pin connections are straightforward, and there is only a special recommendation for the nHOST_INT pin. nHOST_INT can be connected to any input. For interrupt-based operation, nHOST_INT must be connected to an external interrupt that can generate an interrupt on a falling edge. Furthermore, if the host intends to sleep and to be woken up by the NCP, nHOST_INT should be connected to a pin that is capable of waking the host. nHOST_INT should have a pull-up applied to it so that nHOST_INT does not bounce in an unknown state if the NCP is reset. An internal pull-up on the pin that nHOST_INT is connected to is acceptable.

Connect the three SPI signals (MOSI, MISO, and SCLK) to the host's SPI. Connect nSSEL to any output from the host that can operate Slave Select. For many microcontrollers, nSSEL will simply be connected to a general-purpose output. Connect nWAKE and nRESET to any general-purpose output from the host (remember, the NCP supplies an internal pull-up on both the nWAKE and nRESET pins).

Warning: When interfacing the SPI Protocol to an EM260 chip, the nSSEL_INT signal *must* be tied to nSSEL. nSSEL_INT can be left unconnected when interfacing to an EM35x chip.

Verifying SPI Protocol Pin Connections

Once all of the signals are connected and a logic analyzer is attached, begin by pulling the nRESET signal low for a short period (a minimum of 8µs) to reset the NCP. nHOST_INT will return to idle (go high) almost immediately after reset (if it is not already). Note that nHOST_INT will not be driven high by a reset, but instead will default to an input. Therefore, if an external pull-up is not applied to nHOST_INT, it is possible for nHOST_INT to not go high immediately after reset but a short while later. During the startup sequence, the NCP will switch nHOST_INT to an output and actively drive it high. After approximately 250ms, the nHOST_INT signal will assert (go low) and stay asserted until the host initiates a transaction. The startup time of the NCP can vary widely, but 250ms is a good rule of thumb for when nHOST_INT will assert after reset. nHOST_INT asserting after pulling on the nRESET pin indicates that both the nRESET pin and nHOST_INT are connected and operating correctly. If nHOST_INT is tied to an external interrupt on the host, this is also a good time to test the interrupt generation by pulling on the nRESET pin to trigger nHOST_INT assertion.

The simplest method to test nSSEL (as well as nSSEL_INT; tied to nSSEL on EM260 designs) is to simply assert nSSEL (set low) and then deassert nSSEL. When nSSEL deasserts, nHOST_INT will return to idle and then approximately 70µs later will assert again and stay asserted.

Testing the three SPI signals (MOSI, MISO, and SCLK) is best done by formulating a complete transaction. Unfortunately, the nWAKE signal cannot be used or tested until a first, complete transaction has occurred (refer to the section First Transaction: SPI Protocol Version). This is because the nHOST_INT signal must deassert after reset for a proper Wake Handshake to be performed. Once a complete transaction has finished and

nHOST_INT has deasserted, nWAKE may be asserted. Approximately 140µs after nWAKE asserts, nHOST_INT will assert in response, indicating that the nWAKE signal is connected properly.

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 NCP is active and that the SPI Protocol code being implemented on the host is compatible with the firmware on the NCP. Use this transaction to verify the connection with the NCP 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.

SPI Protocol Version Transaction with an NCP Reset

Note: The following steps begin by resetting the NCP to guarantee that it is in a known state. The NCP resetting is an error condition described in the *EM260 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.

Note: The *EM260 Datasheet* discusses each data byte mentioned in the following steps.

1. Assert nRESET and release to reset the NCP and guarantee it is in a known state.
2. Wait for nHOST_INT to assert, which indicates that the NCP 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. Deassert nSSEL to finish the transaction.

SPI Protocol Version Transaction without an NCP 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 0x82.
5. Transmit 0xFF while receiving 0xA7.
6. Deassert nSSEL to finish the transaction.

Second Transaction: SPI Status

The SPI Status transaction is very similar to the SPI Protocol Version transaction (detailed in the previous section). Like the SPI Protocol Version transaction, this transaction provides a simple example of interaction with the NCP. Ember recommends this as a test transaction to verify the connection with the NCP during the host's boot sequence.

Note: The *EM260 Datasheet* discusses each data byte mentioned in the following steps.

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. Deassert nSSEL to finish the transaction.

Performing a Hard Reset

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

1. Toggle nRESET (active low) to reset the NCP.
2. Wait for nHOST_INT to assert, which indicates that the NCP is active.
3. Perform a SPI Protocol Version transaction and verify that the Response from the NCP is the NCP Reset error condition.

4. Perform a SPI Protocol Version transaction and verify that the SPI Protocol Version number is as expected.

5. Perform a SPI Status transaction and verify that the NCP is "Alive."

The purpose of performing this Hard Reset on bootup is threefold.

- By guaranteeing that the NCP 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 NCP was left in.
- Since the NCP generates the NCP 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 a NCP 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 NCP. This handshake not only verifies that the NCP is alive and available to the EZSP, but also that the SPI Protocol implemented in the NCP is compatible with the SPI Protocol implemented on the host.

Third Transaction: EZSP Version Command

Before implementing a generic SPI Protocol on the host, Ember recommends explicitly coding a third transaction for providing exposure to an EZSP Frame and the format of the data involved with an EZSP Frame. The EZSP Frame used in this transaction is the VERSION command. The VERSION command is required to be the first EZSP command issued to the NCP. It exercises the code path all the way through the NCP 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.

Note: This section assumes you are using EmberZNet 4.x. Previous versions of EmberZNet used a different transaction sequence using a NOP command for this example.

Note: The *EM260 Datasheet* discusses each data byte mentioned in the following steps.

EZSP Version Command 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 0x04.
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 0x04.
15. Transmit 0xFF while receiving 0x02.
16. Transmit 0xFF while receiving 0x10. (Note that this value is the build number and may vary.)
17. Transmit 0xFF while receiving 0x45.
18. Transmit 0xFF while receiving 0xA7.
19. Deassert nSSEL to finish the transaction.

The Wait Section

Transmitting a command is a basic operation that simply requires asserting Slave Select and "dumping" the command bytes 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 deassert 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.

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 NCP 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 major 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 the blocking nature of polling. Since transactions must occur serially (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 dictating 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 NCP'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.

Ember recommends choosing the simplest solution possible in the context of the host's resources and other requirements. During development, starting with the simplest blocking `while()` loop is an easy solution that can be expanded and customized as development progresses.

Interrupt on the Falling Edge of nHOST_INT

As detailed and illustrated in the *EM260 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 major advantage to interrupting 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. The major disadvantage to interrupting on nHOST_INT 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 NCP 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.

Inter-Command Spacing

The inter-command spacing is a simple time requirement needed to guarantee that the NCP 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 deasserting Slave Select. Since burning CPU cycles for 1ms is often undesirable, Ember recommends using a simple timer. By setting or starting a timer when Slave Select is deasserted, 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.

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 NCP 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 *EM260 Datasheet* for sample waveforms showing these signals together.

Waking the NCP

Waking the NCP should be a straightforward implementation that only requires you to choose between a polling or an interrupt mechanism for knowing when the NCP 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 deassert nWAKE and continue operating the EZSP as desired.

The only major caveat, 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.

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 NCP 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 NCP.

There is one common exception to this rule: When the host *intentionally* resets the NCP (for example, as described in the section Performing a Hard Reset), the host must expect the NCP 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 NCP (e.g., bootloading). The recommended practice is for the host to have access to and control of the NCP's nRESET signal, and to toggle nRESET if an error condition occurs. When the NCP is being loaded with new firmware, it will not be capable of responding to the host; the host may think the NCP 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 NCP in this situation. Some options include:

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

Error Bytes

As described in the section Special Response Bytes in the *EM260 Datasheet*, there are four SPI Bytes that indicate error conditions. When implementing the code to receive a Response from the NCP, 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 deasserting Slave Select and processing the error. With the exception of an *intentional* NCP Reset error condition, the host should report, through a `printf` or other simple method, these four errors to the developer for debugging purposes, but should ultimately result in an assert or similar reset mechanism.

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 occurs, 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.

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), Ember recommends that the EZSP interface 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.

After Reading This Document

If you have questions or require assistance with the procedures described in this document, contact Ember Customer Support. The Ember Customer Support portal provides a wide array of hardware and software documentation such as FAQ's, reference designs, user guides, application notes, and the latest software available to download. To obtain support on all Ember products and to gain access to the Ember Customer Support portal, visit http://www.ember.com/support_index.html.

Ember Corporation
47 Farnsworth Street
Boston, MA 02210
+1 (617) 951-0200
www.ember.com