AN711: SPI Host Interfacing Guide for Zigbee

This document describes the EZSP-SPI Protocol that is used by a host microcontroller to communicate with a Silicon Labs Network Co-Processor, or NCP, running the EmberZNet PRO stack. It includes recommended procedures for developing and testing a driver for the EZSP-SPI Protocol on a new host microcontroller.

Note: Zigbee EmberZNet SDK 7.0 introduced a new component-based architecture, along with a Project Configurator and other tools to replace AppBuilder and plugin configuration. In general, the new software components are comparable to the plugins. When applicable, instructions for both version 7.0 and higher and 6.10.x and lower are provided. For more information, see *AN1301: Transitioning from Zigbee EmberZNet SDK 6.x to SDK 7.x*.

KEY POINTS

- Physical interface
- SPI Protocol transaction format and timing
- SPI transaction format and timing
- Waking the NCP from sleep
- Powering on, power cycling, and rebooting
- Bootloading the NCP
- Error conditions
- Transaction examples

1. EZSP-SPI Protocol

The SPI protocol refers to the unique framing behavior of the SPI implementation of EZSP as opposed to UART. The EZSP protocol refers to the framing of EZSP-related commands and responses, which are encapsulated by certain SPI frames in the SPI protocol. The EZSP-SPI Protocol uses a 4-wire SPI interface to communicate between the host processor and NCP, plus an additional pair of GPIOs for handshake signaling in each direction.

- To the Host the NCP looks like a hardware peripheral.
- The NCP is the slave device and all transactions are initiated by the Host.
- The SPI interface supports a reasonably high data rate.

This document describes the current protocol versions:

- SPI Protocol version 2 (defined by SPIP_VERSION in spi-protocol.h; unchanged for many years)
- EZSP protocol version 8 (defined by EZSP_PROTOCOL_VERSION in ezsp-protocol.h; as of EmberZNet 6.7)

2. Physical Interface

This section describes the EZSP-SPI Protocol pin connections and how to verify them. It also provides details on the NCP's physical interface configuration, and on low power operation.

2.1 EZSP-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).

2.2 Verifying EZSP-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 to reset the NCP. The required duration of the low nRESET for the EFR32 is at least 35 ns. 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 250 ms, 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 1,100 ms 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.

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 10.3 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. After nWAKE asserts, nHOST_INT will assert in response, indicating that the nWAKE signal is connected properly.

2.3 Physical Interface Configuration

The NCP supports both SPI Slave Mode 0 (clock is idle low, sample on rising edge) and SPI Slave Mode 3 (clock is idle high, sample on rising edge) as illustrated in the following figure. The maximum SPI clock rate is documented in the NCP's reference manual. The convention for the waveforms in this document is to show Mode 0.

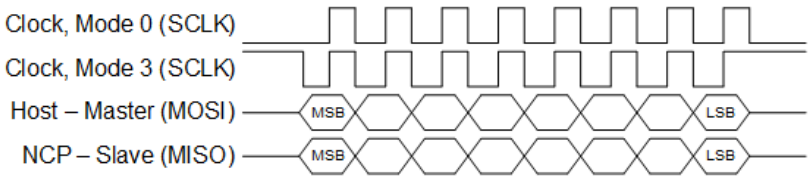

Figure 2.1. SPI Transfer Format, Mode 0 and Mode 3

The nHOST_INT signal and the nWAKE signal are both active low. The Host must supply a pull-up resistor on the nHOST_INT signal to prevent errant interruptions during undefined events such as the NCP resetting. The NCP supplies an internal pull-up on the nWAKE signal to prevent errant interruptions during undefined events such as the Host resetting.

2.4 Low-Power Operation and Signal Configurations

To minimize current consumption, the host should use matching pin configurations. While the NCP supports both Mode 0 and Mode 3, the NCP uses Mode 0. This means that when the NCP is awake, the idle state of the clock is low. When the NCP is sleeping, the host needs to use a configuration that does not conflict with the NCP's configuration to achieve the lowest power. (See also section 6. Waking the NCP from Sleep.) The following table describes the NCP's signal configuration in sleep.

Table 2.1. NCP Signal Configuration in Sleep

Signal	Configuration
MOSI	input, pullup
MISO	input, pullup
SCLK	input, pullup
nSSEL	input, pullup
nHOST_INT	Input, pullup
nWAKE	Input, pullup

3. SPI Protocol Transaction

3.1 Transaction Initiated by the Host

The basic EZSP-SPI transaction is half-duplex to ensure proper framing and to give the NCP adequate response time. The basic transaction, as shown in the following figure, is composed of three sections: Command, Wait, and Response. The transaction can be considered analogous to a function call. The Command section is the function call, and the Response section is the return value.

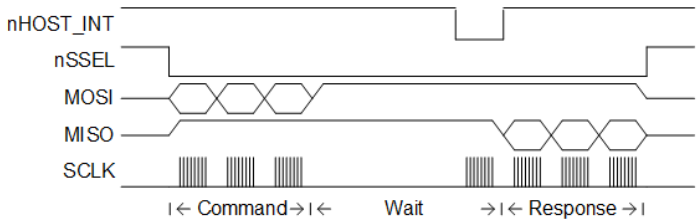

Figure 3.1. General Timing Diagram for a SPI Transaction

Note: Silicon Labs recommends controlling nSSEL from the host to ensure that nSSEL is asserted at the beginning of the transaction, and stays asserted until the end of the transaction, instead of using the auto chip select function.

3.1.1 Command Section

The Host begins the transaction by asserting the Slave Select and then sending a command to the NCP. This command can be of any length from 2 to 136 bytes and must not begin with 0xFF. During the Command section, the NCP will respond with only 0xFF. The Host should ignore data on MISO during the Command section. Once the Host has completed transmission of the entire message, the transaction moves to 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.

The Wait section that occurs between the Command and Response sections is discussed further in section 3.1.2 Wait Section .

3.1.2 Wait Section

The Wait section is a period of time during which the NCP may be processing the command or performing other operations. This section can be any length of time up to 350 ms. Because of the variable size of the Wait section, an interrupt-driven or polling-driven method is suggested for clocking the SPI as opposed to a DMA method. Since the NCP can require up to 350 ms to respond, as long as the Host keeps Slave Select active, the Host can perform other tasks while waiting for a Response.

How the Wait Section is implemented and handled requires some careful consideration of the two techniques available:

- Clock the SPI until the NCP transmits a byte other than 0xFF (also known as polling on the SPI or polling for data) .
- Interrupt on the falling edge of nHOST_INT.

3.1.2.1 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, because transmitting 0xFF is considered an idle line. The NCP will also indicate that a Response is ready by asserting the nHOST_INT signal. The falling edge of nHOST_INT is the indication that a Response is ready. Once the nHOST_INT signal asserts, nHOST_INT will return to idle after the Host begins to clock data.

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. Because 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 PRO.

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.

Silicon Labs 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.

3.1.2.2 Interrupt on the Falling Edge of nHOST_INT

As detailed in section 3.1.3 Response Section, 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. The major disadvantage to interrupting on nHOST_INT is the potential for accidentally starting a new transaction before the previous transaction has completed. Remember, because a new transaction cannot begin until the previous transaction has completed, be careful not to accidentally overlap transactions.

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.

3.1.3 Response Section

When the NCP transmits a byte other than `0xFF`, the transaction has officially moved into the Response section. The NCP signals its readiness to enter the Response section by asserting nHOST_INT after a command from the host has been fully processed. The data format is the same format used in the Command section. The response can be of any length from 2 to 136 bytes and must not begin with `0xFF`. Depending on the actual response, the length of the response is known from the first or second byte. This length should be used by the Host to clock out exactly the correct number of bytes.

Once all bytes have been clocked, the Host is allowed to deassert chip select. Since the Host is in control of clocking the SPI, no ACKs or similar signals are needed back from the Host because the NCP assumes the Host could accept the bytes being clocked on the SPI. After every transaction, the Host must hold the Slave Select high for a minimum of 1 ms. This timing requirement is called the inter-command spacing and is necessary to allow the NCP to process a command and become ready to accept a new command.

3.2 Transaction Initiated by the NCP

When the NCP has data to send to the Host outside of a command/response transaction (Slave Select is idle), it will assert the nHOST_INT signal to indicate asynchronous data waiting in the NCP for the host. The nHOST_INT signal is designed to be an edge-triggered signal as opposed to a level-triggered signal; therefore, the falling edge of nHOST_INT is the true indicator of data availability. The Host then has the responsibility to initiate a transaction to ask the NCP for its output. The Host should initiate this transaction as soon as possible to prevent possible backup of data in the NCP. The NCP will deassert the nHOST_INT signal after receiving a byte on the SPI. Due to inherent latency in the NCP, the timing of when the nHOST_INT signal returns to idle can vary between transactions. nHOST_INT will always return to idle for a minimum of 25 µs before asserting again. If the NCP has more output available after the transaction has completed, the nHOST_INT signal will assert again after Slave Select is deasserted and the Host must make another request.

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.

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.

3.3 Unexpected Resets

The NCP is designed to protect itself against undefined behavior due to unexpected resets. The protection is based on the state of Slave Select since the inter-command spacing mandates that Slave Select must return to idle. The NCP's internal SPI Protocol uses Slave Select returning to idle as a trigger to reinitialize its SPI Protocol. By always reinitializing, the NCP is protected against the Host unexpectedly resetting or terminating a transaction. Additionally, if Slave Select is active when the NCP powers on, the NCP will ignore SPI data until Slave Select returns to idle. By ignoring SPI traffic until idle, the NCP will not begin receiving in the middle of a transaction.

If the Host resets, in most cases it should also reset the NCP so that both devices are once again in the same state: freshly booted.

If the NCP resets during a transaction, the Host can expect either a Wait Section timeout or a missing Frame Terminator indicating an invalid Response.

If the NCP resets outside of a transaction, the Host should proceed normally.

4. SPI Protocol Data Format

4.1 Data Format

The data format, also referred to as a *command*, is the same for both the Command section and the Response section. The data format of the SPI Protocol is straightforward, as illustrated in the following figure.

SPI Byte	Length or Error	Payload Frame (Variable Length)	Frame Terminator

Figure 4.1. EZSP-SPI Protocol Data Format

The total length of a command must not exceed 136 bytes.

All commands must begin with the **SPI Byte**. Some commands are only two bytes—that is, they contain the SPI Byte and Frame Terminator only.

The **Length Byte** is only included if there is information in the Payload Frame and the Length Byte defines the length of just the Payload Frame. Therefore, if a command includes a Payload Frame, the Length Byte can have a value from 3 through 133 and the overall command size will be from 6 through 136 bytes. The SPI Byte can be a specific value indicating if there is a Payload Frame or not. If there is a Payload Frame, then the Length Byte can be expected.

The **Error Byte** is used by the error responses to provide additional information about the error and appears in place of the length byte. This additional information is described in the following sections.

The **Payload Frame** contains the data needed for operating EmberZNet PRO. The EZSP Frame and its format are explained in *UG100: EZSP Reference Guide*. The Payload Frame may also contain the data needed for operating the bootloader, which is called a Bootloader Frame. Refer to *UG103.6: Bootloading Fundamentals*, for more information on the bootloader.

The **Frame Terminator** is a special control byte used to mark the end of a command. The Frame Terminator byte is defined as 0xA7 and is appended to all Commands and Responses immediately after the final data byte. The purpose of the Frame Terminator is to provide a known byte the SPI Protocol can use to detect a corrupt command. For example, if the NCP resets during the Response Section, the Host will still clock out the correct number of bytes. But when the host attempts to verify the value 0xA7 at the end of the Response, it will see either the value 0x00 or 0xFF and know that the NCP just reset and the corrupt Response should be discarded.

Note: The Length Byte only specifies the length of the Payload Frame. It does not include the Frame Terminator.

4.2 SPI Bytes

There are four primary SPI Bytes: SPI Protocol Version, SPI Status, Bootloader Frame, and EZSP Frame.

- **SPI Protocol Version [0x0A]:** Sending this command requests the SPI Protocol Version number from the SPI Interface. The response will always have bit 7 set and bit 6 cleared. In this current version, the response will be 0x82, because the version number corresponding to this set of Command-Response values is version number 2. The version number can be a value from 1 to 63 (0x81–0xBF).
- **SPI Status [0x0B]:** Sending this command asks for the NCP status. The response status byte will always have the upper 2 bits set. In this current version, the status byte only has one status bit [0], which is set if the NCP is alive and ready for commands.
- **Bootloader Frame [0xFD]:** This byte indicates that the current transaction is a Bootloader transaction and there is more data to follow. This SPI Byte will cause the transaction to look like the full data format illustrated in Figure 4.1 EZSP-SPI Protocol Data Format on page 8. The byte immediately after this SPI Byte will be a Length Byte, and it is used to identify the length of the Bootloader Frame. Refer to *UG103.6: Bootloading Fundamentals*, for more information on the bootloader. If the SPI Byte is 0xFD, the minimum transaction size is 4 bytes.
- **EZSP Frame [0xFE]:** This byte indicates that the current transaction is an EZSP transaction and there is more data to follow. This SPI Byte will cause the transaction to look like the full data format illustrated in Figure 4.1 EZSP-SPI Protocol Data Format on page 8. The byte immediately after this SPI Byte will be a Length Byte, and it is used to identify the length of the EZSP Frame. The EZSP Frame is defined in document *UG100: EZSP Reference Guide*. If the SPI Byte is 0xFE, the minimum transaction size is six bytes.

Only five SPI Byte values, 0x00–0x04, are ever used as error codes (see the following table). When the error condition occurs, any command sent to the NCP will be ignored and responded to with one of these codes. These special SPI Bytes must be trapped and dealt with. In addition, for each error condition, the Error Byte (instead of the Length Byte) is also sent with the SPI Byte. See section 9.1 Error Bytes for more information.

The following table summarizes the possible commands and their responses in the SPI Byte.

Table 4.1. SPI Commands and Responses

Command Value	Command	Response Value	Response
Any	Any	0x00	NCP reset occurred. See section 7. Powering On, Power Cycling, and Rebooting. This is never used in another Response; it always indicates an NCP Reset. Error byte: The reset type. For an enumeration of these reset causes, refer to platform/service/legacy_hal/inc/em2xx-reset-defs.h.
Any	Any	0x01	Oversized Payload Frame received. The command contained an EZSP frame with a Length Byte greater than 133. The NCP was forced to drop the entire command. This is never used in another Response; it always indicates an overflow occurred. Error byte: reserved.
Any	Any	0x02	Aborted Transaction occurred. The transaction was not completed properly and the NCP was forced to abort the transaction. This is never used in another Response; it always indicates an aborted transaction occurred. Error byte: reserved
Any	Any	0x03	Missing Frame Terminator. The command was missing the Frame Terminator. The NCP was forced to drop the entire command. This is never used in another Response; it always indicates a Missing Frame Terminator in the Command. Error byte: reserved
Any	Any	0x04	Unsupported SPI Command. The command contained an unsupported SPI Byte. The NCP was forced to drop the entire command. This is never used in another Response; it always indicates an unsupported SPI Byte in the Command. Error byte: reserved.
0x00 – 0x09	Reserved	[none]	[none]
0x0A	SPI Protocol Version	0x81 – 0xBF	bit[7] is always set. bit[6] is always cleared. bit[5:0] is a number from 1 to 63.
0x0B	SPI Status	0xC0 – 0xC1	bit[7] is always set. bit[6] is always set. bit[0]—Set if alive and ready for commands.
0x0C – 0xFC	Reserved	[none]	[none]
0xFD	Bootloader Frame	0xFD	Bootloader Frame

Command Value	Command	Response Value	Response
0xFE	EZSP Frame	0xFE	EZSP Frame
0xFF	Invalid	0xFF	Invalid

5. SPI Protocol Timing

The following figure illustrates all critical timing parameters in the SPI Protocol. These timing parameters are a result of the NCP's internal operation and both constrain Host behavior and characterize NCP operation. The parameters shown are discussed elsewhere in this document. Note that this figure is not drawn to scale, but is instead drawn only to illustrate where the parameters are measured.

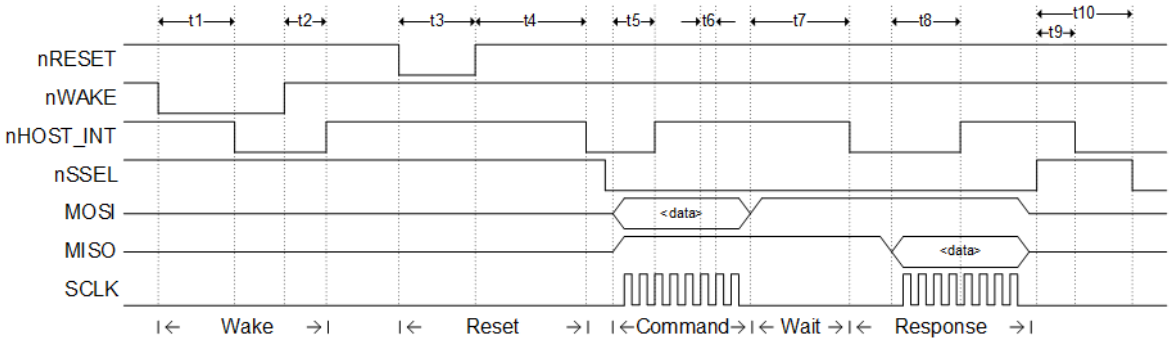

Figure 5.1. SPI Protocol Timing Waveform

5.1 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. To ensure that the NCP is always able to deal with incoming commands, a minimum inter-command spacing is defined at 1 ms between the rising edge of Slave Select (ending transaction) and the falling edge of Slave Select (starting transaction). After every transaction, the Host must hold the Slave Select high for a minimum of 1 ms. The Host must respect the inter-command spacing requirement, or the NCP will not have time to operate on the command; additional commands could result in error conditions or undesired behavior. If the nHOST_INT signal is not already asserted, the Host is allowed to use the Wake handshake instead of the inter-command spacing to determine if the NCP is ready to accept a command.

If the host is capable of blocking for 1 ms, the simplest solution is to simply burn CPU cycles for 1 ms after deasserting Slave Select. Since burning CPU cycles for 1 ms is often undesirable, Silicon Labs 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.

5.2 SPI Protocol Timing Measurements

Be aware that timing quoted in this document is for general guidance and should not be used for precise calculations. Timing will vary depending on factors like different hardware, different code paths, different radio traffic, and a variety of sequences that cannot be predicted.

Parameter	Description	Min.	Typ.	Max.	Unit
t1(a)	Wake handshake, while NCP is awake	-	Sleepy SPI NCPs not yet supported.	-	-
t1(b)	Wake handshake, while NCP is asleep	-	Sleepy SPI NCPs not yet supported.	-	-
t2	Wake handshake, finish	-	Sleepy SPI NCPs not yet supported.	-	-
t3	Reset pulse width	-	Refer to device's datasheet.	-	-
t4(a)	Startup time, entering application	-	1.1	-	ms
t4(b)	Startup time, entering bootloader	-	330	-	µs
t5	nHOST_INT deasserting after command	1.0	1.5	4.3	ms
t6	Clock period	-	Refer to SPI slave max timing in device's datasheet.	-	-
t7	Wait section	-	-	350	ms
t8	nHOST_INT deasserting after response	1.0	N/A	4.3	-
t9	nHOST_INT asserting after transaction	223	-	-	µs
t10	Inter-command spacing	1	-	-	ms

The timings shown in the table were measured against an EFR32MG12 on a BRD4170A (which has an EFR32MG12P433F1024GM68 using a 38.4MHz high frequency crystal). All Series 1 devices will have similar timing. Ultimately, the timing depends on the EFR's clock speed for how quickly it can respond to commands.

The parameter t4(a) "Startup time, entering application" is measured from startup/power on/pin reset to falling edge of nHOST_INT alerting the host that the NCP is ready.

The parameter t4(b) "Startup time, entering bootloader" is measured from startup/power on/pin reset with nWAKE assert to falling edge of nHOST_INT alerting the host that the NCP is ready.

The parameter t5 "nHOST_INT deasserting after command" is measured from the rising edge of the hosts transmit/command (MOSI) to rising edge of nHOST_INT.

The parameter t6 "Clock period" has to do with the speed of the SPI clock. Every EFR device has its own maximum speed for SPI Slave functionality. Refer to an NCP's datasheet regarding maximum SPI Slave speed since the exact clock source might be different. For the device mentioned here, its datasheet lists (6 * tHFPERCLK) for minimum clock period as a SPI Slave.

The parameter t8 "nHOST_INT deasserting after response" is measured from the rising edge of NCP transmit/response (MISO) to rising edge of nHOST_INT. t8 is a very rare event since nHOST_INT is typically already deasserted and left deasserted after t5.

The parameter t9 "nHOST_INT asserting after transaction" is measured from the rising edge of nSSEL to the falling edge of nHOST_INT when the transaction is oversized. Oversized transactions are the cleanest to invoke and therefore measure. There is only a minimum value of how fast the NCP responds since it's considered an asynchronous event with no upper limit.

5.2.1 SPI Protocol Timing Measurements on Different Series 1 Devices

The preceding section details timing for an EFR32MG12 on a BRD4170A (which has an EFR32MG12P433F1024GM68). The following table shows some examples of timing across different Series 1 devices.

Parameter	Description	XG1	XG12	XG13	XG14
t4(a)	Startup time, entering applications	1.15 s	1.11 s	1.13 s	1.11 s
t5	nHOST_INT deasserting after command	1.5 ms	1 ms	1 ms	1 ms
t9	nHOST_INT asserting after transaction	254 µs	223 µs	235 µs	235 µs

These three parameters are the only ones that would vary in a meaningful way. The other parameters are not quoted across devices because:

- t1 and t2 are for waking up and N/A until sleepy SPI NCPs are supported.
- t3 is described by the device's datasheet and does not relate to SPI timing.
- t6 is described by the device's datasheet and does not relate to SPI timing.
- t8 is a very rare event since nHOST_INT is typically already deasserted and left deasserted after t5.
- t7 and t10 are generous parameters allowing the NCP to complete its functionality and do not relate to SPI timing.

5.2.2 SPI Protocol Timing Measurements Between Series 1 and Series 2 Devices

There is a lot of timing variability between all the devices, especially when it comes to configuration, app activity, stack activity, and all other behaviors that occur with a Host and NCP.

For general comparison of timing, the parameter t5 is used because it is the most common and fundamental behavior of the NCP reacting to a host command. Remember that t5 is nHOST_INT deasserting after command.

The following table shows some statistics of t5 timing, in microseconds, after the Host boots up and resets the NCP.

Device	Min.	Max.	Median	Std Deviation
EFR32xG12	411	4674	1303	622
EFR32xG21	264	4008	1032	539
EFR32xG24	289	4002	955	529

This table shows that Series 2 is faster than Series 1, but there is no major variation.

5.3 Interfacing EZSP to the EZSP-SPI Protocol

Due to the serial nature of the EZSP (that is, transactions must occur in sequence instead of overlapping), Silicon Labs 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 EZSP-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 could require up to 350 ms or a wake handshake could require up to 300 ms.

6. Waking the NCP from Sleep

Waking up the NCP involves a simple handshaking routine as illustrated in the following figure. This handshaking ensures that the Host will wait until the NCP is fully awake and ready to accept commands from the Host. If the NCP is already awake when the handshake is performed (such as when the Host resets and the NCP is already operating), the handshake will proceed as described below with no ill effects.

Figure 6.1. NCP Wake Sequence

Note: A wake handshake cannot be performed if nHOST_INT is already asserted.

nWAKE should not be asserted after the NCP has been reset until the NCP has fully booted, as indicated by the NCP asserting nHOST_INT. If nWAKE is asserted during this boot time, the NCP may enter bootloader mode. Refer to section 8. Bootloading 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 EZSP-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, is to make sure the proper piece of code gets triggered in response and to not perform further EZSP operations inside of interrupt context.

Waking the NCP involves the following steps:

1. Host asserts nWAKE.
2. NCP interrupts on nWAKE and exits sleep.
3. NCP performs all operations it needs to and will not respond until it is ready to accept commands.
4. NCP asserts nHOST_INT within 300 ms of nWAKE asserting.

   If the NCP does not assert nHOST_INT within 300 ms of nWAKE, it is valid for the Host to consider the NCP unresponsive and to reset the NCP.
5. Host detects nHOST_INT assertion. Because the assertion of nHOST_INT indicates the NCP can accept SPI transactions, the Host does not need to hold Slave Select high for the normally required minimum 1 ms of inter-command spacing.
6. Host deasserts nWAKE after detecting nHOST_INT assertion.
7. NCP will deassert nHOST_INT within 25 µs of nWAKE deasserting.
8. After 25 µs, any change on nHOST_INT will be an indication of a normal asynchronous (callback) event.

7. Powering On, Power Cycling, and Rebooting

When the Host powers on (or reboots), it cannot guarantee that the NCP is awake and ready to receive commands. Therefore, the Host should always perform the Wake NCP handshake to guarantee that the NCP is awake. If the NCP resets, it needs to inform the Host so that the Host can reconfigure the stack if needed.

When the NCP resets, it will assert the nHOST_INT signal, telling the Host that it has data. The Host should request data from the NCP as usual. The NCP will ignore whatever command is sent to it and respond only with two bytes. The first byte will always be `0x00` and the second byte will be the reset type as defined by `platform/service/legacy_hal/inc/em2xx-reset-defs.h`. This specialty SPI Byte is never used in another Response SPI Byte. If the Host sees `0x00` from the NCP, it knows that the NCP has been reset. The NCP will deassert the nHOST_INT signal shortly after receiving a byte on the SPI and process all further commands in the usual manner. In addition to the Host having control of the reset line of the NCP, the EmberZNet Serial Protocol also provides a mechanism for a software reboot.

When the host resets, it is far simpler to reset the NCP and begin from a known state than to try 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 an EZSP-SPI Protocol Version transaction and verify that the Response from the NCP is the NCP Reset error condition.
4. Perform an EZSP-SPI Protocol Version transaction and verify that the EZSP-SPI Protocol Version number is as expected.
5. Perform an EZSP-SPI Status transaction and verify that the NCP is “Alive” and ready to accept commands.

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.
- Because 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 an NCP Reset error is experienced outside of a Hard Reset, it can be treated as a true unexpected error condition.
- The EZSP-SPI Protocol Version and EZSP-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 EZSP-SPI Protocol implemented in the NCP is compatible with the EZSP-SPI Protocol implemented on the host.

8. Bootloading the NCP

The SPI Protocol supports a Payload Frame called the Bootloader Frame for communicating with the NCP when the NCP is in bootloader mode. The NCP can enter bootload mode through either an EZSP command or a recovery pin low while the NCP exits reset. In the Gecko Bootloader, a recovery pin can be defined in the board configuration to initiate bootloader mode. While nWAKE is the default for that function, it can be defined to be any valid GPIO pin. The pin definition method for the Gecko Bootloader is through BTL_GPIO_ACTIVATION_PORT/PIN/POLARITY set through the Gecko Bootloader's "EZSP GPIO Activation" component/plugin.

Assert nRESET to hold the NCP in reset. While nRESET is asserted, assert (active low) nWAKE and then deassert nRESET to boot the NCP. Do not deassert nWAKE until the NCP asserts nHOST_INT, indicating that the NCP has fully booted and is ready to accept data over the SPI Protocol. Once nHOST_INT is asserted, nWAKE may be deasserted. Refer to *UG100: EZSP Reference Manual* for more information on the bootloader and the format of the Bootloader Frame.

9. 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 development and debugging aids. If the host experiences an error condition, chances are the host and the NCP are 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 section 7. Powering On, Power Cycling, and Rebooting), 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 (for example, 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.

If two or more different error conditions occur back to back, only the first error condition will be reported to the Host (if it is possible to report the error). The following are error conditions that might occur with the NCP.

Unsupported SPI Command: If the SPI Byte of the command is unsupported, the NCP will drop the incoming command and respond with the Unsupported SPI Command Error Response. This error means the SPI Byte is unsupported by the NCP's current Mode. Bootloader Frames can only be used with the bootloader, and EZSP Frames can only be used with the EZSP.

Oversized Payload Frame: If the transaction includes a Payload Frame, the Length Byte cannot be a value greater than 133. If the NCP detects a length byte greater than 133, it will drop the incoming Command and abort the entire transaction. The NCP will then assert nHOST_INT after Slave Select returns to Idle to inform the Host through an error code in the Response section what has happened. The NCP not only drops the Command in the problematic transaction, but the next Command is also dropped because it is responded to with the Oversized Payload Frame Error Response.

Aborted Transaction: An aborted transaction is any transaction where Slave Select returns to Idle prematurely and the SPI Protocol dropped the transaction. The most common reason for Slave Select returning to Idle prematurely is the Host unexpectedly resetting. If a transaction is aborted, the NCP will assert nHOST_INT to inform the Host through an error code in the Response section what has happened. When a transaction is aborted, the NCP not only drops the Command in the problematic transaction, but the next Command also gets dropped because it is responded to with the Aborted Transaction Error Response.

Missing Frame Terminator: Every Command and Response must be terminated with the Frame Terminator byte. The NCP will drop any Command that is missing the Frame Terminator. The NCP will then immediately provide the Missing Frame Terminator Error Response.

Long Transaction: A Long Transaction error occurs when the Host clocks too many bytes. As long as the inter-command spacing requirement is met, this error condition should not cause a problem because the NCP will send only `0xFF` outside of the Response section and ignore incoming bytes outside of the Command section.

Unresponsive: Unresponsive can mean the NCP is not powered, not fully booted yet, incorrectly connected to the Host, or busy performing other tasks. The Host must wait the maximum length of the Wait section before it can consider the NCP unresponsive to the Command section. This maximum length is 350 ms, measured from the end of the last byte sent in the Command section. If the NCP ever fails to respond during the Wait section, it is valid for the Host to consider the NCP unresponsive and to reset the NCP. Additionally, if nHOST_INT does not assert within 300 ms of nWAKE asserting during the wake handshake, the Host can consider the NCP unresponsive and reset the NCP.

9.1 Error Bytes

As described in section 4.2 SPI Bytes, five SPI Bytes 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.

9.2 Timeouts

The host can experience only two timeouts: 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.

10. Transaction Examples

This section contains the following transaction examples:

- SPI Protocol Version
- EmberZNet Serial Protocol Frame—Version Command
- NCP Reset
- EZSP-SPI status
- Three-Part Transaction: Wake, Get Version, Stack Status Callback

Note: The specific bytes of the EZSP portion of the frame examples below may vary among stack releases and especially among EZSP protocol versions. To confirm the EZSP frame format expected by your NCP firmware, please refer to the "Protocol Format" section of *UG100: EZSP Reference Guide* from the specific EmberZNet release corresponding to your NCP's firmware build.

10.1 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 and results in the first transaction performed after a reset returning the reset error. These steps describe receiving this reset error instead of the EZSP-SPI Protocol Version. The EZSP-SPI protocol version transaction is described in section 10.3 SPI Protocol Version.

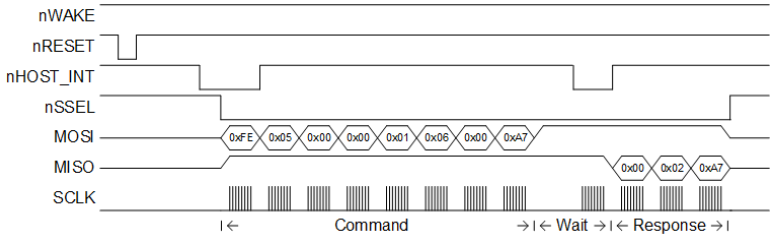

Figure 10.1. NCP Reset Example

1. Assert nRESET and release, which toggles active low to reset the NCP and guarantee it is in a known status..
2. nWAKE stays idle high between nRESET and nHOST_INT indicating the NCP should continue with normal booting (do not enter the bootloader).
3. nHOST_INT asserts, indicating that the NCP is active.
4. Assert nSSEL to begin a transaction.
5. Transmit the command:
   - 0xFE: SPI Byte indicating an EZSP Frame
   - 0x05: Length Byte showing the EZSP Frame is 5 bytes long
   - 0x00: EZSP Sequence Byte (Note that this value should vary based upon previous sequence bytes)
   - 0x00: EZSP Frame Control Low Byte indicating a command with no sleeping
   - 0x01: EZSP Frame Control High Byte indicating the frame version is 1 and not using an encrypted EZSP connection
   - 0x0600: EZSP Frame ID (2 bytes) indicating the callback command (sending the low byte first, so the EZSP Frame ID is 0x0006)
   - 0xA7: Frame Terminator
6. Wait for nHOST_INT to assert.
7. Transmit and receive 0xFF until a byte other than 0xFF is received.
8. Receive response 0x00 (a byte other than 0xFF).
9. Receive the Error Byte and decode (0x02 is enumerated as RESET_POWERON).
10. Receive the Frame Terminator (0xA7).
11. Response 0x00 indicates the NCP has reset and the Host should respond appropriately.
12. Deassert nSSEL to finish the transaction.
13. Since nHOST_INT does not assert again, there is no more data for the Host.

10.2 EZSP-SPI Status

The EZSP-SPI Status transaction is very similar to the EZSP-SPI Protocol Version transaction. Like the EZSP-SPI Protocol Version transaction, this transaction provides a simple example of interaction with the NCP. Silicon Labs recommends this as a test transaction to verify the connection with the NCP during the host's boot sequence.

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.

10.3 SPI Protocol Version

Obtaining the EZSP-SPI Protocol Version is a compact, simplified, and special transaction that illustrates a full transaction. Being able to properly obtain the EZSP-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 EZSP-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.

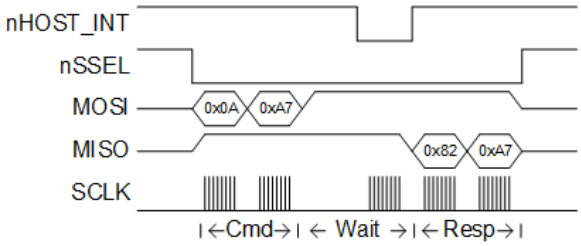

Figure 10.2. SPI Protocol Version Example

1. Activate Slave Select (nSSEL).
2. Transmit the command 0x0A - SPI Protocol Version Request.
3. Transmit the Frame Terminator, 0xA7.
4. Wait for nHOST_INT to assert.
5. Transmit and receive 0xFF until a byte other than 0xFF is received.
6. Receive response 0x82 (a byte other than 0xFF), then receive the Frame Terminator, 0xA7.
7. Bit 7 is always set and bit 6 is always cleared in the Version Response, so this is Version 2.
8. Deactivate Slave Select.

10.4 EmberZNet Serial Protocol Frame—Version Command

Before implementing a generic EZSP-SPI Protocol on the host, Silicon Labs recommends explicitly coding a 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 must 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.

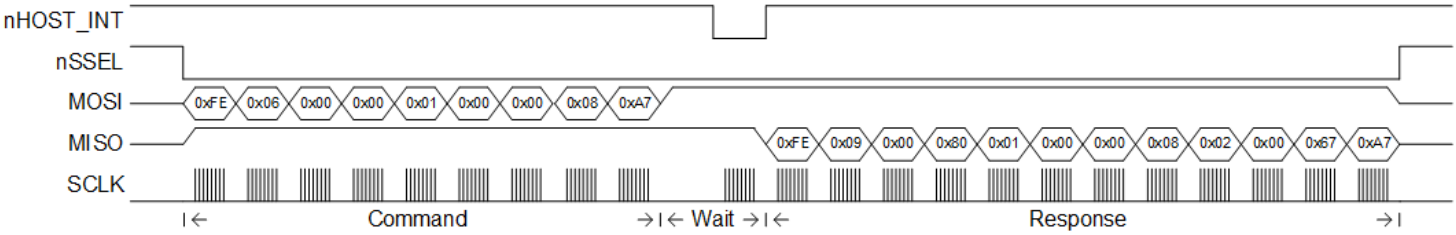

Figure 10.3. EmberZNet Serial Protocol Frame - Version Command Example

1. Assert nSSEL to begin a transaction.
2. Transmit the appropriate command:
   - 0xFE: SPI Byte indicating an EZSP Frame
   - 0x06: Length Byte showing the EZSP Frame is 6 bytes long
   - 0x00: EZSP Sequence Byte (Note that this value should vary based upon previous sequence bytes)
   - 0x00: EZSP Frame Control Low Byte indicating a command with no sleeping
   - 0x01: EZSP Frame Control High Byte indicating the frame version is 1 and not using an encrypted EZSP connection
   - 0x0000: EZSP Frame ID (2 bytes) indicating the Version command
   - 0x08: EZSP Parameter for this command (desiredProtocolVersion, note that this value may vary; your software may be newer than the version 8 shown here).
   - 0xA7: Frame Terminator
3. Wait for nHOST_INT to assert.
4. Transmit and receive 0xFF until a byte other than 0xFF is received.
5. Receive response 0xFE (a byte other than 0xFF) and read the next byte for a length.
6. Stop transmitting after the number of bytes (length) is received plus the Frame Terminator.
7. Decode the response:
   - 0xFE: SPI Byte indicating an EZSP Frame
   - 0x09: Length Byte showing the EZSP Frame is 9 bytes long
   - 0x00: EZSP Sequence Byte (Note that this value should vary based upon previous sequence bytes)
   - 0x80: EZSP Frame Control Low Byte indicating a response with no errors and with no pending callbacks
   - 0x01: EZSP Frame Control High Byte indicating the frame version is 1 and not using an encrypted EZSP connection
   - 0x0000: EZSP Frame ID (2 bytes) indicating the Version response
   - 0x08: EZSP Parameter for this response (protocolVersion, note that this value may vary; your software may be newer than the version 8 shown here)
   - 0x02: EZSP Parameter for this response (stackType)
   - 0x00: EZSP Parameter for this response (stackVersion, note that this value may vary)
   - 0x67: EZSP Parameter for this response (stackVersion: denotes the major version and minor version, such as 0x67 for EmberZNet 6.7).)
   - 0xA7: Frame Terminator
8. Deassert nSSEL to finish the transaction.

10.5 Three-Part Transaction: Wake, Get Version, Stack Status Callback

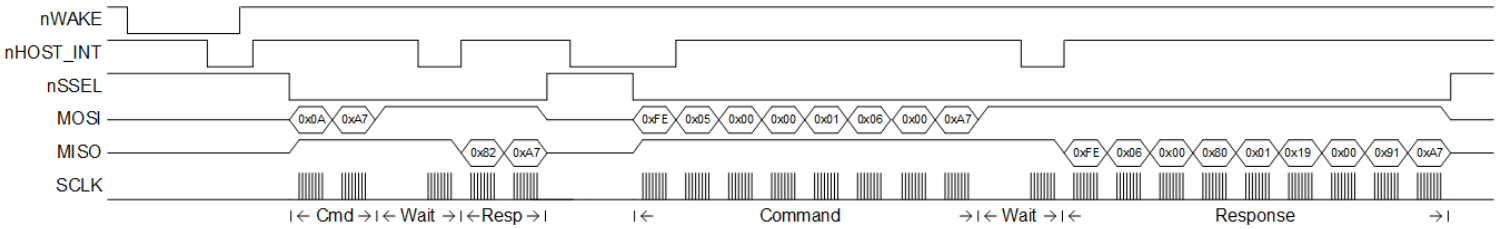

Figure 10.4. Timing Diagram of the Three-Part Transaction

1. Activate nWAKE and activate timeout timer.
2. NCP wakes up (if not already) and enables communication.
3. nHOST_INT asserts, indicating the NCP can accept commands.
4. Host sees nHOST_INT activation within 300 ms and deactivates nWAKE and timeout timer.
5. nHOST_INT deasserts immediately after nWAKE.
6. Assert nSSEL to begin a transaction.
7. Transmit the Command 0x0A - SPI Protocol Version Request.
8. Transmit the Frame Terminator, 0xA7.
9. Wait for nHOST_INT to assert.
10. Transmit and receive 0xFF until a byte other than 0xFF is received.
11. Receive response 0x82 (a byte other than 0xFF), then receive the Frame Terminator, 0xA7.
12. Bit 7 is always set and bit 6 is always cleared in the Version Response, so this is Version 2.
13. Deassert nSSEL to finish the transaction.
14. Host begins timing the inter-command spacing of 1 ms in preparation for sending the next command.
15. nHOST_INT asserts shortly after deactivating Slave Select, indicating a callback.
16. Host sees nHOST_INT, but waits for the 1 ms before responding.
17. Assert nSSEL to begin a transaction.
18. Transmit the command:
    - 0xFE: SPI Byte indicating an EZSP Frame
    - 0x05: Length Byte showing the EZSP Frame is 5 bytes long
    - 0x00: EZSP Sequence Byte (Note that this value should vary based upon previous sequence bytes)
    - 0x00: EZSP Frame Control Low Byte indicating a command with no sleeping
    - 0x01: EZSP Frame Control High Byte indicating the frame version is 1 and not using an encrypted EZSP connection
    - 0x0600: EZSP Frame ID (2 bytes) indicating the callback command (sending the low byte first, so the EZSP Frame ID is 0x0006)
    - 0xA7: Frame Terminator
19. Wait for nHOST_INT to assert.
20. Transmit and receive 0xFF until a byte other than 0xFF is received.
21. Receive response 0xFE (a byte other than 0xFF), read the next byte for a length.
22. Stop transmitting after the number of bytes (length) is received plus the Frame Terminator.
23. Decode the response:
    - 0xFE: SPI Byte indicating an EZSP Frame
    - 0x06: Length Byte showing the EZSP Frame is 6 bytes long
    - 0x00: EZSP Sequence Byte (Note that this value should vary based upon previous sequence bytes)
    - 0x80: EZSP Frame Control Low Byte indicating a response with no errors
    - 0x01: EZSP Frame Control High Byte indicating the frame version is 1 and not using an encrypted EZSP connection
    - 0x1900: EZSP Frame ID (2 bytes) indicating the stackStatusHandler command (sending the low byte first, so the EZSP Frame ID is 0x0019)
    - 0x91: EZSP Parameter for this response (EmberStatus EMBER_NETWORK_DOWN)
    - 0xA7: Frame Terminator
24. Deassert nSSEL to finish the transaction
25. Since nHOST_INT does not assert again, there is no more data for the Host.

11. Configuring a Linux Host

Configuring a Linux Host with an EFR32-based NCP is relatively straightforward with application framework support in the EmberZNet SDK. In this case, the application framework support takes care of the necessary configuration of a typical Linux-based host to work with an EFR32-based NCP, and developers do not have to go through every detail in the preceding sections in this document in order to start implementing a host application. We still recommend that you become familiar with the information here, especially the SPI protocol transaction concepts and examples and timing requirements.

By default, the EmberZNet SDK selects an EZSP UART interface to communicate with the NCP. In EmberZNet 6.x, to enable SPI, you must remove the EZSP UART plugin from your project and replace it with the EZSP SPI plugin. In EmberZNet 7.x, you must remove the NCP UART Hardware Flow Control component from your project and replace it with the SPI NCP component. You may do this from the Plugins tab in Simplicity Studio's Simplicity IDE. For more help on using Simplicity Studio and EmberZNet SDK, refer to *QSG106: Getting Started with EmberZNet PRO* if using EmberZNet 6.x, or *QSG180: Zigbee EmberZNet v7.x Quick-Start Guide for EmberZNet 7.x*.
In EmberZNet 6.x, save the project and generate its files. After generating the project, you must edit the generated board header file to configure the hardware settings for the project. The board header file is located in the generated project folder and is named <project-name>_board.h. The following macros can optionally be defined in this board header to specify the hardware configuration for the SPI driver. To preserve these changes, do not overwrite this file if you need to generate project files again. In EmberZNet 7.x, the configuration for the SPI driver must be done through the Driver > SPI component configuration.

Note: Any macro not defined in the board header file will revert to the defaults values as shown here.

Table 11.1. SPI Driver Macros and their Default Values

Macro	Default Value
#define NCP_SPI_DEVICE	"/dev/spidev0.0"
#define NCP_SPI_MODE	0
#define NCP_SPI_SPEED_HZ	(1024 * 1024) // 1 MHz
#define NCP_CHIP_SELECT_GPIO	"8"
#define NCP_HOST_INT_GPIO	"22"
#define NCP_RESET_GPIO	"23"
#define NCP_WAKE_GPIO	"24"
#define NCP_RESET_DELAY_US	26

12. Notes on How to Measure Timing

In general, gathering timing data requires a logic analyzer on all the SPI pins, nHOST_INT, nWAKE, and nRESET. The timing data for this document was gathered by using each radio board on a WSTK development kit connected to a Raspberry Pi running a Z3Gateway app. The radio boards were connected to the Raspberry Pi using BRD8016A. Each NCP device was programmed with a Bootloader SPI EZSP and NCP SPI as found in Simplicity Studio. The bootloader and NCP SPI available in Simplicity Studio needed to be generated, compiled, and then flashed down. Programming the NCP was accomplished using a separate WSTK in Debug Mode: OUT. There are no prebuilt binaries due to the variation in devices and build combinations.

IoT Portfolio
www.silabs.com/products

Quality
www.silabs.com/quality

Support & Community
www.silabs.com/community

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

www.silabs.com