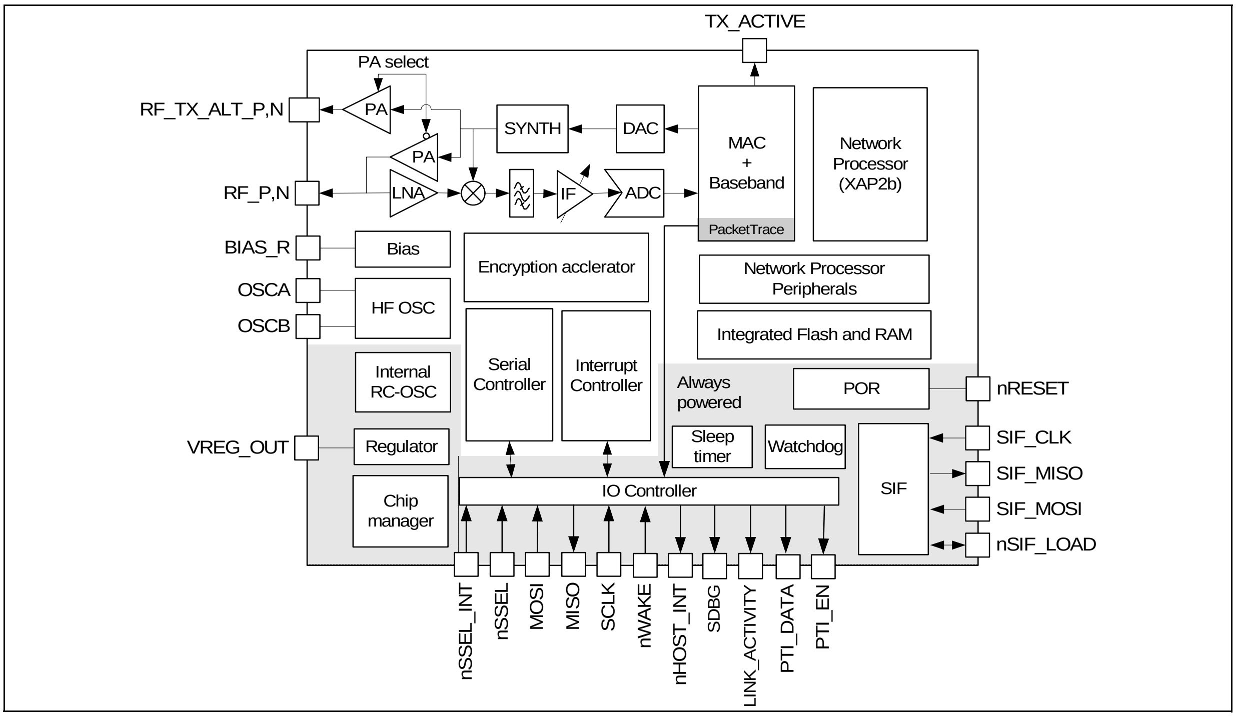
SN260

ZigBee® 802.15.4 network processor

Features

- Integrated 2.4GHz, IEEE 802.15.4-compliant transceiver:
  - Robust RX filtering allows co-existence with IEEE 802.11g and Bluetooth devices
  - - 97.5dBm RX sensitivity (1% PER, 20byte packet)
  - + 3dBm nominal output power
  - Increased radio performance mode (boost mode) gives –98.5dBm sensitivity and +5dBm transmit power
  - Integrated VCO and loop filter
  - Secondary TX-only RF port for applications requiring external PA.
- Integrated IEEE 802.15.4 PHY and MAC
- Dedicated peripherals and integrated memory
- EmberZNet™ ZigBee®-compliant stack running on the dedicated network processor
- Controlled by the Host using the EmberZNet™ Serial Protocol (EZSP)
  - Standard serial interface (allows for connection to a variety of Host micro controllers)
- Non-intrusive debug interface (SIF)
- Integrated hardware and software support for InSight Development Environment
- Provides integrated RC oscillator for low power operation
- Three sleep modes:
  - Processor idle (automatic)
  - Deep sleep—1.0µA
  - Power down—1.0µA
- Watchdog timer and power-on-reset circuitry
- Integrated AES encryption accelerator
- Integrated 1.8V voltage regulator
- Compatible with Ember EM250



Contents

1 General description

The SN260 integrates a 2.4GHz, IEEE 802.15.4-compliant transceiver with a 16-bit network processor (XAP2b core) to run EmberZNet, the ZigBee-compliant network stack. The SN260 exposes access to the EmberZNet API across a standard SPI module, allowing application development on a Host processor. This means that the SN260 can be viewed as a ZigBee peripheral connected over a SPI. The XAP2b microprocessor is a power-optimized core integrated in the SN260. It contains integrated Flash and RAM memory along with an optimized peripheral set to enhance the operation of the network stack.

The transceiver utilizes an efficient architecture that exceeds the dynamic range requirements imposed by the IEEE 802.15.4-2003 standard by over 15dB. The integrated receive channel filtering allows for co-existence with other communication standards in the 2.4GHz spectrum such as IEEE 802.11g and Bluetooth. The integrated regulator, VCO, loop filter, and power amplifier keep the external component count low. An optional high-performance radio mode (boost mode) is software selectable to boost dynamic range by a further 3dB.

The SN260 contains embedded Flash and integrated RAM for program and data storage. By employing an effective wear-leveling algorithm, the stack optimizes the lifetime of the embedded Flash, and affords the application the ability to configure stack and application tokens within the SN260.

To maintain the strict timing requirements imposed by ZigBee and the IEEE 802.15.4-2003 standard, the SN260 integrates a number of MAC functions into the hardware. The MAC hardware handles automatic ACK transmission and reception, automatic backoff delay, and clear channel assessment for transmission, as well as automatic filtering of received packets. In addition, the SN260 allows for true MAC level debugging by integrating the Packet Trace Interface.

An integrated voltage regulator, power-on-reset circuitry, sleep timer, and low-power sleep modes are available. The deep sleep and power down modes draws less than 1 µA, allowing products to achieve long battery life.

Finally, the SN260 utilizes the non-intrusive SIF module for powerful software debugging and programming of the network processor.

Target applications for the SN260 include:

- Building automation and control
- Home automation and control
- Home entertainment control
- Asset tracking

The SN260 can only be purchased with the EmberZNet stack. This technical datasheet details the SN260 features available to customers using it with the EmberZNet stack.

2 Pin assignment

Figure 1. SN260 pin assignment

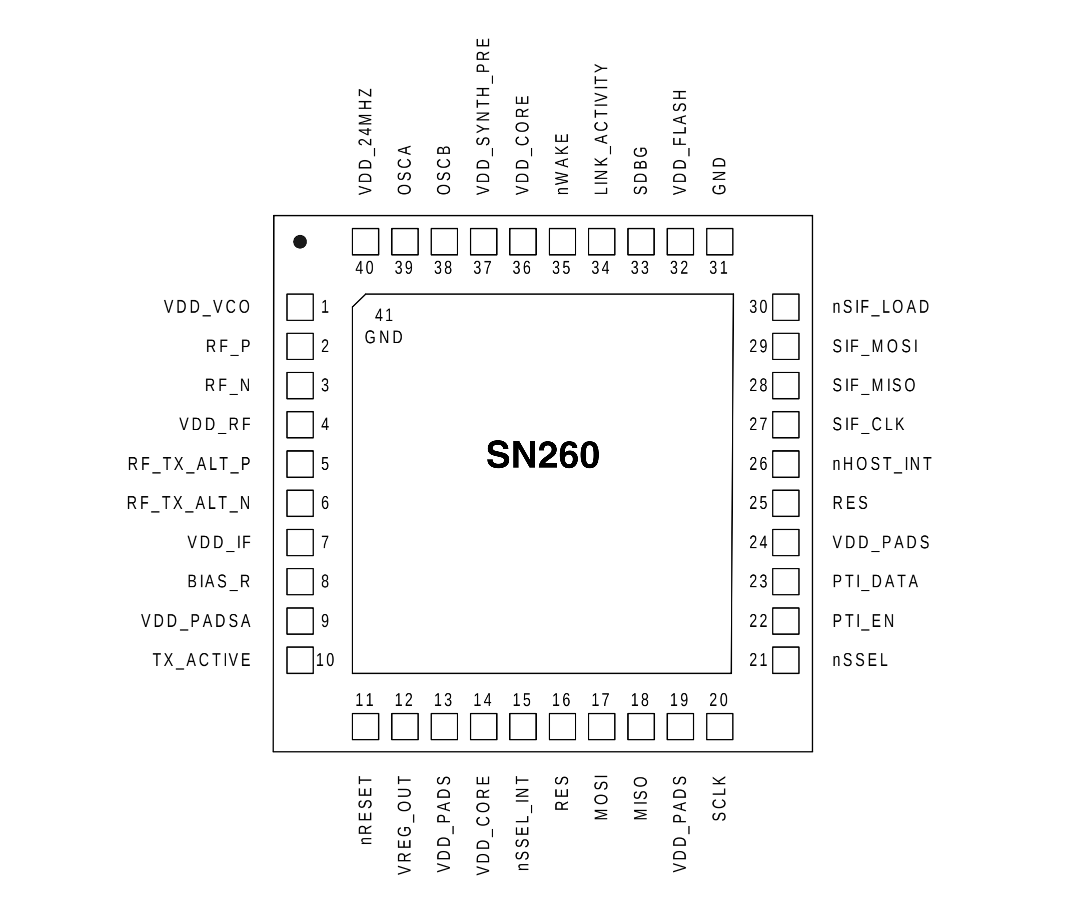


Table 1. Pin descriptions

Pin #	Signal	Direction	Description
1	VDD_VCO	Power	1.8V VCO supply
2	RF_P	I/O	Differential (with RF_N) receiver input/transmitter output
3	RF_N	I/O	Differential (with RF_P) receiver input/transmitter output
4	VDD_RF	Power	1.8V RF supply (LNA and PA)
5	RF_TX_ALT_P	O	Differential (with RF_TX_ALT_N) transmitter output (optional)
6	RF_TX_ALT_N	O	Differential (with RF_TX_ALT_P) transmitter output (optional)
7	VDD_IF	Power	1.8V IF supply (mixers and filters)
8	BIAS_R	I	Bias setting resistor
9	VDD_PADSA	Power	Analog pad supply (1.8V)
10	TX_ACTIVE	O	Logic-level control for external RX/TX switch The SN260 baseband controls TX_ACTIVE and drives it high (1.8V) when in TX mode. (Refer to *Table 6* and section TX_ACTIVE signal.)

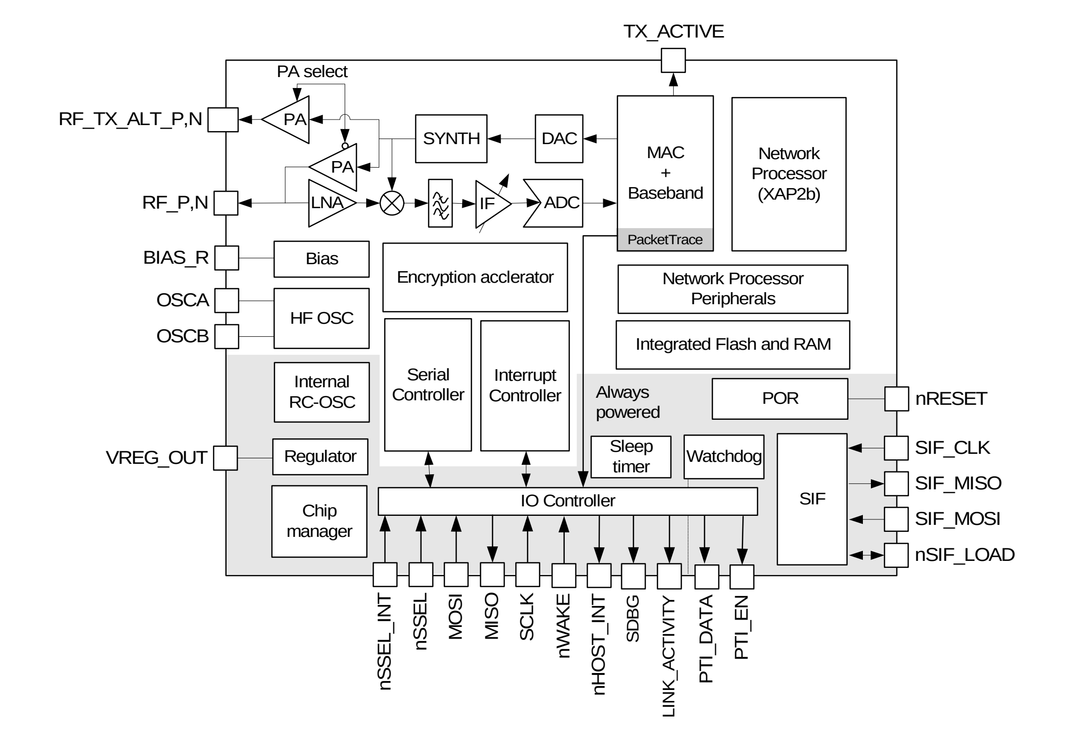
Table 1. Pin descriptions (continued)

Pin #	Signal	Direction	Description
11	nRESET	I	Active low chip reset (internal pull-up)
12	VREG_OUT	Power	Regulator output (1.8V)
13	VDD_PADS	Power	Pads supply (2.1 – 3.6V)
14	VDD_CORE	Power	1.8V digital core supply
15	nSSEL_INT	I	SPI Slave Select Interrupt (from Host to SN260) This signal must be connected to nSSEL (Pin 21)
16	RES		Reserved for future use, do not connect to any signal.
17	MOSI	I	SPI Data, Master Out / Slave In (from Host to SN260)
18	MISO	O	SPI Data, Master In / Slave Out (from SN260 to Host)
19	VDD_PADS	Power	Pads supply (2.1 – 3.6V)
20	SCLK	I	SPI Clock (from Host to SN260)
21	nSSEL	I	SPI Slave Select (from Host to SN260)
22	PTI_EN	O	Frame signal of Packet Trace Interface (PTI)
23	PTI_DATA	O	Data signal of Packet Trace Interface (PTI)
24	VDD_PADS	Power	Pads supply (2.1 – 3.6V)
25	RES		Reserved for future use, do not connect to any signal.
26	nHOST_INT	O	Host Interrupt signal (from SN260 to Host)
27	SIF_CLK	I	Serial Interface, Clock (internal pull down)
28	SIF_MISO	O	Serial Interface, Master In / Slave Out
29	SIF_MOSI	I	Serial Interface, Master Out / Slave In
30	nSIF_LOAD	I/O	Serial Interface, load strobe (open collector with internal pull up)
31	GND	Power	Ground Supply
32	VDD_FLASH	Power	1.8V Flash memory supply
33	SDBG	O	Spare Debug signal
34	LINK_ACTIVITY	O	Link and Activity signal
35	nWAKE	I	Wake Interrupt signal (from Host to SN260)
36	VDD_CORE	Power	1.8V digital core supply
37	VDD_SYNTH_PRE	Power	1.8V synthesizer and pre-scalar supply
38	OSCB	I/O	24MHz crystal oscillator or left open for when using an external clock input on OSCA
39	OSCA	I/O	24MHz crystal oscillator or external clock input
40	VDD_24MHZ	Power	1.8V high-frequency oscillator supply
41	GND	Ground	Ground supply pad in the bottom center of the package forms Pin 41 (see the *EM260 Reference Design* for PCB considerations)

3 Top-level functional description

Figure 2 shows a detailed block diagram of the SN260.

Figure 2. SN260 block diagram



The radio receiver is a low-IF, super-heterodyne receiver. It utilizes differential signal paths to minimize noise interference, and its architecture has been chosen to optimize co-existence with other devices within the 2.4GHz band (namely, IEEE 802.11g and Bluetooth). After amplification and mixing, the signal is filtered and combined prior to being sampled by an ADC.

The digital receiver implements a coherent demodulator to generate a chip stream for the hardware-based MAC. In addition, the digital receiver contains the analog radio calibration routines and control of the gain within the receiver path.

The radio transmitter utilizes an efficient architecture in which the data stream directly modulates the VCO. An integrated PA boosts the output power. The calibration of the TX path as well as the output power is controlled by digital logic. If the SN260 is to be used with an external PA, the TX_ACTIVE signal should be used to control the timing of the external switching logic.

The integrated 4.8GHz VCO and loop filter minimize off-chip circuitry. Only a 24MHz crystal with its loading capacitors is required to properly establish the PLL reference signal.

The MAC interfaces the data memory to the RX and TX baseband modules. The MAC provides hardware-based IEEE 802.15.4 packet-level filtering. It supplies an accurate symbol time base that minimizes the synchronization effort of the software stack and meets the protocol timing requirements. In addition, it provides timer and synchronization assistance for the IEEE 802.15.4 CSMA-CA algorithm.

The SN260 integrates hardware support for a Packet Trace module, which allows robust packet-based debug. This element is a critical component of InSight Desktop, the Ember software IDE, providing advanced network debug capability when coupled with the InSight Adapter.

The SN260 integrates a 16-bit XAP2b microprocessor developed by Cambridge Consultants Ltd. This power-efficient, industry-proven core provides the appropriate level of processing power to meet the needs of the EmberZNet Zigbee-compliant stack, EmberZNet. In addition, the SIF module provides a non-intrusive programming and debug interface allowing for real-time application debugging.

The SN260 exposes the EmberZNet Serial API over the SPI, which allows application development to occur on a Host micro controller of choice. In addition to the four SPI signals, two additional signals, nHOST_INT and nWAKE, provide an easy-to-use handshake mechanism between the Host and the SN260.

The integrated voltage regulator generates a regulated 1.8V reference voltage from an unregulated supply voltage. This voltage is decoupled and routed externally to supply the 1.8V to the core logic. In addition, an integrated POR module allows for the proper cold start of the SN260.

The SN260 contains one high-frequency (24MHz) crystal oscillator and, for low-power operation, a second low-frequency internal 10 kHz oscillator.

The SN260 contains two power domains. The always-powered high voltage supply is used for powering the GPIO pads and critical chip functions. The rest of the chip is powered by a regulated Low Voltage Supply which can be disabled during deep sleep to reduce the power consumption.

4 Electrical characteristics

4.1 Absolute maximum ratings

Table 2 lists the absolute maximum ratings for the SN260.

Table 2. Absolute maximum ratings

Parameter	Test conditions	Min.	Max.	Unit
Regulator voltage (VDD_PADS)		- 0.3	3.6	V
Core voltage (VDD_24MHZ, VDD_VCO, VDD_RF, VDD_IF, VDD_PADSA, VDD_FLASH, VDD_SYNTH_PRE, VDD_CORE)		- 0.3	2.0	V
Voltage on RF_P,N; RF_TX_ALT_P,N		- 0.3	3.6	V
Voltage on nSSEL_INT, MOSI, MISO, SCLK, nSSEL, PTI_EN, PTI_DATA, nHOST_INT, SIF_CLK, SIF_MISO, SIF_MOSI, nSIF_LOAD, SDBG, LINK_ACTIVITY, nWAKE, nRESET, VREG_OUT		- 0.3	VDD_PADS+0.3	V
Voltage on TX_ACTIVE, BIAS_R, OSCA, OSCB		- 0.3	VDD_CORE+0.3	V
Storage temperature		- 40	+ 140	°C

4.2 Recommended operating conditions

Table 3 lists the rated operating conditions of the SN260.

Table 3. Operating conditions

Parameter	Test conditions	Min.	Typ.	Max.	Unit
Regulator input voltage (VDD_PADS)		2.1		3.6	V
Core input voltage (VDD_24MHZ, VDD_VCO, VDD_RF, VDD_IF, VDD_PADSA, VDD_FLASH, VDD_SYNTH_PRE, VDD_CORE)		1.7	1.8	1.9	V
Temperature range		- 40		+ 85	°C

4.3 Environmental characteristics

Table 4 lists the environmental characteristics of the SN260.

Table 4. Environmental characteristics

Parameter	Test Conditions	Min.	Typ.	Max.	Unit
ESD (human body model)	On any pin	- 2		+ 2	kV
ESD (charged device model)	Non-RF pins	- 400		+ 400	V
ESD (charged device model)	RF pins	- 225		+ 225	V
MSL (moisture sensitivity level)			TBD		

4.4 DC electrical characteristics

Table 5 lists the DC electrical characteristics of the SN260.

Table 5. DC characteristics

Parameter	Test Conditions	Min.	Typ.	Max.	Unit
Regulator input voltage (VDD_PADS)		2.1		3.6	V
Power supply range (VDD_CORE)	Regulator output or external input	1.7	1.8	1.9	V
Deep sleep current					
Quiescent current, including internal RC oscillator	At 25° C			1.0	µA
RX current					
Radio receiver, MAC, and baseband (boost mode)			29.0		mA
Radio receiver, MAC, and baseband			27.0		mA
CPU, RAM, and Flash memory	At 25° C and 1.8V core		8.5		mA
Total RX current $(= I_{\text{Radio receiver, MAC and baseband, CPU}} + I_{\text{RAM, and Flash memory}})$	At 25° C, VDD_PADS = 3.0V		35.5		mA
TX current					
Radio transmitter, MAC, and baseband (boost mode)	At max. TX power (+ 5dBm typical)		33.0		mA
Radio transmitter, MAC, and baseband	At max. TX power (+ 3dBm typical)		27.0		mA
	At 0dBm typical		24.3		mA
	At min. TX power (- 32dBm typical)		19.5		mA
CPU, RAM, and Flash memory	At 25° C, VDD_PADS = 3.0V		8.5		mA
Total TX current $(= I_{\text{Radio transmitter, MAC and baseband, CPU}} + I_{\text{RAM, and Flash memory}})$	At 25° C and 1.8V core; max. power out		35.5		mA

4.5 Digital I/O specifications

Table 6 contains the digital I/O specifications for the SN260. The digital I/O power (named VDD_PADS) comes from three dedicated pins (pins 13, 19, and 24). The voltage applied to these pins sets the I/O voltage.

Table 6. Digital I/O specifications

Parameter	Name	Min.	Typ.	Max.	Unit
Voltage supply	VDD_PADS	2.1		3.6	V
Input voltage for logic 0	V_{IL}	0		0.2 x VDD_PADS	V
Input voltage for logic 1	V_{IH}	0.8 x VDD_PADS		VDD_PADS	V
Input current for logic 0	I_{IL}			-0.5	A
Input current for logic 1	I_{IH}			0.5	A
Input pull-up resistor value	R_{IPU}		30		kΩ
Input pull-down resistor value	R_{IPD}		30		kΩ
Output voltage for logic 0	V_{OL}	0		0.18 x VDD_PADS	V
Output voltage for logic 1	V_{OH}	0.82 x VDD_PADS		VDD_PADS	V
Output source current (standard current pad)	I_{OHS}			4	mA
Output sink current (standard current pad)	I_{OLS}			4	mA
Output source current (high current pad: pins 33, 34, and 35)	I_{OHH}			8	mA
Output sink current (high current pad: pins 33, 34, and 35)	I_{OLH}			8	mA
Total output current (for I/O pads)	$I_{OH} + I_{OL}$			40	mA
Input voltage threshold for OSCA		0.2 x VDD_CORE		0.8 x VDD_PADS	V
Output voltage level (TX_ACTIVE)		0.18 x VDD_CORE		0.82 x VDD_CORE	V
Output source current (TX_ACTIVE)				1	mA

4.6 RF electrical characteristics

4.6.1 Receive

Table 7 lists the key parameters of the integrated IEEE 802.15.4 receiver on the SN260.

Note: Receive measurements were collected with Ember's EM260 reference design at 2440MHz. The Typical number indicates one standard deviation above the mean.

Table 7. Receive characteristics

Parameter	Test conditions	Min.	Typ.	Max.	Unit
Frequency range		2400		2500	MHz
Sensitivity (boost mode)	1% PER, 20byte packet defined by IEEE 802.15.4	- 93	- 98.5		dBm
Sensitivity	1% PER, 20byte packet defined by IEEE 802.15.4	- 92	- 97.5		dBm
High-side adjacent channel rejection	IEEE 802.15.4 signal at -82dBm		35		dB
Low-side adjacent channel rejection	IEEE 802.15.4 signal at - 82dBm		35		dB
2nd high-side adjacent channel rejection	IEEE 802.15.4 signal at - 82dBm		40		dB
2nd low-side adjacent channel rejection	IEEE 802.15.4 signal at - 82dBm		40		dB
Channel rejection for all other channels	IEEE 802.15.4 signal at - 82dBm		40		dB
802.11g rejection centered at + 12MHz or - 13MHz	IEEE 802.15.4 signal at - 82dBm		40		dB
Maximum input signal level for correct operation (low gain)		0			dBm
Image suppression			30		dB
Co-channel rejection	IEEE 802.15.4 signal at - 82dBm		- 6		dBc
Relative frequency error (2 x 40 ppm required by IEEE 802.15.4)		- 120		+ 120	ppm
Relative timing error (2 x 40 ppm required by IEEE 802.15.4)		- 120		+ 120	ppm
Linear RSSI range			40		dB

4.6.2 Transmit

Table 8 lists the key parameters of the integrated IEEE 802.15.4 transmitter on the SN260.

Note: *Transmit measurements were collected with Ember's EM260 reference design at 2440MHz. The Typical number indicates one standard deviation below the mean.*

Table 8. Transmit characteristics

Parameter	Test conditions	Min.	Typ.	Max.	Unit
Maximum output power (boost mode)	At highest power setting		5		dBm
Maximum output power	At highest power setting	0	3		dBm
Minimum output power	At lowest power setting		- 32		dBm
Error vector magnitude	As defined by IEEE 802.15.4, which sets a 35% maximum		15	25	%
Carrier frequency error		- 40		+ 40	ppm
Load impedance			200		
PSD mask relative	3.5MHz away	- 20			dB
PSD mask absolute	3.5MHz away	- 30			dBm

4.6.3 Synthesizer

Table 9 lists the key parameters of the integrated synthesizer on the SN260.

Table 9. Synthesizer characteristics

Parameter	Test conditions	Min.	Typ.	Max.	Unit
Frequency range		2400		2500	MHz
Frequency resolution			11.7		kHz
Lock time	From off, with correct VCO DAC setting			100	s
Relock time	Channel change or RX/TX turnaround (IEEE 802.15.4 defines 192s turnaround time)			100	s
Phase noise at 100kHz			- 71		dBc/Hz
Phase noise at 1MHz			- 91		dBc/Hz
Phase noise at 4MHz			- 103		dBc/Hz
Phase noise at 10MHz			- 111		dBc/Hz

5 Functional description

The SN260 connects to the Host micro controller through a standard SPI interface. The EmberZNet Serial Protocol (EZSP) has been defined to allow an application to be written on a host micro controller of choice. Therefore, the SN260 comes with a license to EmberZNet, the EmberZNet ZigBee-compliant software stack. The following brief description of the hardware modules provides the necessary background on the operation of the SN260. For more information, contact your local ST sales representative.

5.1 Receive (RX) path

The SN260 RX path spans the analog and digital domains. The RX architecture is based on a low-IF, super-heterodyne receiver. It utilizes differential signal paths to minimize noise interference. The input RF signal is mixed down to the IF frequency of 4MHz by I and Q mixers. The output of the mixers is filtered and combined prior to being sampled by a 12Msps ADC. The RX filtering within the RX path has been designed to optimize the co-existence of the SN260 with other 2.4GHz transceivers, such as the IEEE 802.11g and Bluetooth.

5.1.1 RX baseband

The SN260 RX baseband (within the digital domain) implements a coherent demodulator for optimal performance. The baseband demodulates the O-QPSK signal at the chip level and synchronizes with the IEEE 802.15.4-2003 preamble. Once a packet preamble is detected, it de-spreads the demodulated data into 4-bit symbols. These symbols are buffered and passed to the hardware-based MAC module for filtering.

In addition, the RX baseband provides the calibration and control interface to the analog RX modules, including the LNA, RX Baseband Filter, and modulation modules. The EmberZNet software includes calibration algorithms which use this interface to reduce the effects of process and temperature variation.

5.1.2 RSSI and CCA

The SN260 calculates the RSSI over an 8-symbol period as well as at the end of a received packet. It utilizes the RX gain settings and the output level of the ADC within its algorithm.

The SN260 RX baseband provides support for the IEEE 802.15.4-2003 required CCA methods summarized in *Table 10*. Modes 1, 2, and 3 are defined by the 802.15.4-2003 standard; Mode 0 is a proprietary mode.

Table 10. CCA mode behavior

CCA mode	Mode behavior
0	Clear channel reports busy medium if either carrier sense OR RSSI exceeds their thresholds.
1	Clear channel reports busy medium if RSSI exceeds its threshold.
2	Clear channel reports busy medium if carrier sense exceeds its threshold.
3	Clear channel reports busy medium if both RSSI and carrier sense exceed their thresholds.

5.2 Transmit (TX) path

The SN260 transmitter utilizes both analog circuitry and digital logic to produce the O-QPSK modulated signal. The area-efficient TX architecture directly modulates the spread symbols prior to transmission. The differential signal paths increase noise immunity and provide a common interface for the external balun.

5.2.1 TX baseband

The SN260 TX baseband (within the digital domain) performs the spreading of the 4-bit symbol into its IEEE 802.15.4-2003-defined 32-chip I and Q sequence. In addition, it provides the interface for software to perform the calibration of the TX module in order to reduce process, temperature, and voltage variations.

5.2.2 TX_ACTIVE signal

Even though the SN260 provides an output power suitable for most ZigBee applications, some applications will require an external power amplifier (PA). Due to the timing requirements of IEEE 802.15.4-2003, the SN250 provides a signal, TX_ACTIVE, to be used for external PA power management and RF Switching logic. When in TX, the TX Baseband drives TX_ACTIVE high (as described in *Table 6*). When in RX, the TX_ACTIVE signal is low. If an external PA is not required, then the TX_ACTIVE signal should be connected to GND through a 100k Ohm resistor, as shown in the application circuit in *Figure 12*.

5.3 Integrated MAC module

The SN260 integrates critical portions of the IEEE 802.15.4-2003 MAC requirements in hardware. This allows the SN260 to provide greater bandwidth to application and network operations. In addition, the hardware acts as a first-line filter for non-intended packets. The SN260 MAC utilizes a DMA interface to RAM memory to further reduce the overall micro controller interaction when transmitting or receiving packets.

When a packet is ready for transmission, the software configures the TX MAC DMA by indicating the packet buffer RAM location. The MAC waits for the backoff period, then transitions the baseband to TX mode and performs channel assessment. When the channel is clear, the MAC reads data from the RAM buffer, calculates the CRC, and provides 4-bit symbols to the baseband. When the final byte has been read and sent to the baseband, the CRC remainder is read and transmitted.

The MAC resides in RX mode most of the time, and different format and address filters keep non-intended packets from using excessive RAM buffers, as well as preventing the SN260 CPU from being interrupted. When the reception of a packet begins, the MAC reads 4-bit symbols from the baseband and calculates the CRC. It assembles the received data for storage in a RAM buffer. A RX MAC DMA provides direct access to the RAM memory. Once the packet has been received, additional data is appended to the end of the packet in the RAM buffer space. The appended data provides statistical information on the packet for the software stack.

The primary features of the MAC are:

- CRC generation, appending, and checking
- Hardware timers and interrupts to achieve the MAC symbol timing
- Automatic preamble, and SFD pre-pended to a TX packet
- Address recognition and packet filtering on received packets
- Automatic acknowledgement transmission
- Automatic transmission of packets from memory
- Automatic transmission after backoff time if channel is clear (CCA)
- Automatic acknowledgement checking
- Time stamping of received and transmitted messages
- Attaching packet information to received packets (LQI, RSSI, gain, time stamp, and packet status)
- IEEE 802.15.4-2003 timing and slotted/unslotted timing

5.4 Packet trace interface (PTI)

The SN260 integrates a true PHY-level PTI for effective network-level debugging. This two-signal interface monitors all the PHY TX and RX packets (in a non-intrusive manner) between the MAC and baseband modules. It is an asynchronous 500kbps interface and cannot be used to inject packets into the PHY/MAC interface. The two signals from the SN260 are the frame signal (PTI_EN) and the data signal (PTI_DATA). The PTI is supported by InSight Desktop.

5.5 16-bit microprocessor

The SN260 integrates the XAP2b microprocessor developed by Cambridge Consultants Ltd., making it a true network processor solution. The XAP2b is a 16-bit Harvard architecture processor with separate program and data address spaces. The word width is 16 bits for both the program and data sides.

The standard XAP2 microprocessor and accompanying software tools have been enhanced to create the XAP2b microprocessor used in the SN260. The XAP2b adds data-side byte addressing support to the XAP2 allowing for more productive usage of RAM and optimized code.

The XAP2b clock speed is 12MHz. When used with the EmberZNet stack, firmware is loaded into Flash memory over the air or by a serial link using a built-in bootloader in a reserved area of the Flash. Alternatively, firmware may be loaded via the SIF interface with the assistance of RAM-based utility routines also loaded via SIF.

5.6 Embedded memory

The SN260 contains embedded Flash and RAM memory for firmware storage and execution. In addition it partitions a portion of the Flash for simulated EEPROM and token storage.

5.6.1 Simulated EEPROM

The protocol stack reserves a section of Flash memory to provide simulated EEPROM storage area for stack and customer tokens. The Flash cell has been qualified for a data retention time of >100 years at room temperature and is rated to have a guaranteed 1,000 write/erase cycles. Because the Flash cells are qualified for up to 1,000 write cycles, the simulated EEPROM implements an effective wear-leveling algorithm which effectively extends the number of write cycles for individual tokens.

The number of set-token operations is finite due to the write cycle limitation of the Flash. It is not possible to guarantee an exact number of set-token operations because the life of the simulated EEPROM depends on which tokens are written and how often.

The SN260 stores non-volatile information necessary for network operation as well as 8 tokens available to the Host (see section *Section 7.2.6: Tokens on page 37*). The majority of internal tokens are only written when the SN260 performs a network join or leave operation. As a simple estimate of possible set-token operations, consider an SN260 in a stable network (no joins or leaves) not sending any messages where the Host uses only one of the 8-byte tokens available to it. Under this scenario, a very rough estimate results in approximately 330,000 possible set-token operations. The number of possible set-token calls, though, depends on which tokens are being set, so the ratios of set-token calls for each token plays a large factor. A very rough estimate for the total number of times an App token can be set is approximately 320,000.

These estimates would typically increase if the SN260 is kept closer to room temperature, since the 1,000 guaranteed write cycles of the Flash is for across temperature.

5.6.2 Flash information area (FIA)

The SN260 also includes a separate 1024-byte FIA that can be used for storage of data during manufacturing, including serial numbers and calibration values. Programming of this special Flash page can only be enabled using the SIF interface to prevent accidental corruption or erasure. The EmberZNet stack reserves a small portion of this space for its own use and in addition makes eight manufacturing tokens available to the application. See *Section 7.2.6: Tokens on page 37*, for more information.

5.7 Encryption accelerator

The SN260 contains a hardware AES encryption engine that is attached to the CPU using a memory-mapped interface. NIST-based CCM, CCM*, CBC-MAC, and CTR modes are implemented in hardware. These modes are described in the IEEE 802.15.4-2003 specification, with the exception of CCM*, which is described in the ZigBee Security Services Specification 1.0. The EmberZNet stack implements a security API for applications that require security at the application level.

5.8 Reset detection

The SN260 contains multiple reset sources. The reset event is logged into the reset source register, which lets the CPU determine the cause of the last reset. The following reset causes are detected:

- Power-on-reset
- Watchdog
- PC rollover
- Software reset
- Core power dip

5.9 Power-on-reset (POR)

Each voltage domain (1.8V digital core supply VDD_CORE and pads supply VDD_PADS) has a power-on-reset (POR) cell.

The VDD_PADS POR cell holds the always-powered high-voltage domain in reset until the following conditions have been met:

- The high-voltage pads supply VDD_PADS voltage rises above a threshold.
- The internal RC clock starts and generates three clock pulses.
- The 1.8V POR cell holds the main digital core in reset until the regulator output voltage rises above a threshold.

Additionally, the digital domain counts 1,024 clock edges on the 24MHz crystal before releasing the reset to the main digital core.

Table 11 lists the features of the SN260 POR circuitry.

Table 11. POR specifications

Parameter	Min.	Typ.	Max.	Unit
VDD_PADS POR release	1.0	1.2	1.4	V
VDD_PADS POR assert	0.5	0.6	0.7	V
1.8V POR release	1.35	1.5	1.65	V
1.8V POR hysteresis	0.08	0.1	0.12	V

5.10 Clock sources

The SN260 integrates two oscillators: a high-frequency 24MHz crystal oscillator and a low-frequency internal 10kHz RC oscillator.

5.10.1 High-frequency crystal oscillator

The integrated high-frequency crystal oscillator requires an external 24MHz crystal with an accuracy of ±40ppm. Based upon the application bill of materials and current consumption requirements, the external crystal can cover a range of ESR requirements. For a lower ESR, the cost of the crystal increases but the overall current consumption decreases. Likewise, for higher ESR, the cost decreases but the current consumption increases. Therefore, the designer can choose a crystal to fit the needs of the application.

Table 12 lists the specifications for the high-frequency crystal.

Table 12. High-frequency crystal specifications

Parameter	Test conditions	Min.	Typ.	Max.	Unit
Frequency			24		MHz
Duty cycle		40		60	%
Phase noise from 1kHz to 100kHz				- 120	dBc/Hz
Accuracy	Initial, temperature, and aging	- 40		+ 40	ppm
Crystal ESR	Load capacitance of 10pF			100	W
Crystal ESR	Load capacitance of 18pF			60	W
Start-up time to stable clock (max. bias)				1	ms
Start-up time to stable clock (optimum bias)				2	ms
Current consumption	Good crystal: 20Ω ESR, 10pF load		0.2	0.3	mA
Current consumption	Worst-case crystals (60Ω, 18pF or 100Ω, 10pF)			0.5	mA
Current consumption	At maximum bias			1	mA

5.10.2 Internal RC oscillator

The SN260 has a low-power, low-frequency RC oscillator that runs all the time. Its nominal frequency is 10kHz. It is divided down to 1kHz using a variable divider to allow software to accurately calibrate it. This calibrated clock is used by the sleep timer.

Time-keeping accuracy depends on temperature fluctuations the chip is exposed to, power supply impedance, and the calibration interval, but in general it will be better than 150ppm (including crystal error of 40ppm).

Table 13 lists the specifications of the RC oscillator.

Table 13. RC oscillator specifications

Parameter	Test conditions	Min.	Typ.	Max.	Unit
Frequency			10		kHz
Analog trim steps			1		kHz
Frequency variation with supply	For a voltage drop from 3.6V to 3.1V or 2.6V to 2.1V		0.75	1.5	%

5.11 Random number generator

The SN260 allows for the generation of random numbers by exposing a randomly generated bit from the RX ADC. Analog noise current is passed through the RX path, sampled by the receive ADC, and stored in a register. The value contained in this register could be used to seed a software-generated random number. The EmberZNet stack utilizes these random numbers to seed the random MAC backoff and encryption key generators.

5.12 Watchdog timer

The SN260 contains an internal watchdog timer clocked from the internal oscillator. If the timer reaches its time-out value of approximately 2 seconds, it will reset the SN260. This reset signal cannot be routed externally to the Host.

The SN260 firmware will periodically restart the watchdog timer while the firmware is running normally. The Host cannot effect or configure the watchdog timer.

5.13 Sleep timer

The 16-bit sleep timer is contained in the always-powered digital block. The clock source for the sleep timer is a calibrated 1kHz clock. The frequency is slowed down with a 2^N prescaler to generate a final timer resolution of 1ms. With a 1ms tick and a 16-bit timer, the timer wraps about every 65.5 seconds. The EmberZNet stack appropriately handles timer wraps allowing the Host to order a theoretical maximum sleep delay of 4 million seconds.

5.14 Power management

The SN260 supports four different power modes: active, idle, deep sleep, and power down.

Active mode is the normal, operating state of the SN260.

While in idle mode, code execution halts until any interrupt occurs. All modules of the SN260 including the radio continue to operate normally. The EmberZNet stack automatically invokes idle as appropriate.

Deep sleep mode and power down mode both power off most of the SN260, including the radio, and leave only the critical chip functions powered. The internal regulator is disabled and VREG_OUT is turned off. All output signals are maintained in a frozen state. Upon waking from deep sleep or power down mode, the internal regulator is re-enabled. Deep sleep and power down result in the same sleep current consumption. The two sleep modes differ as follows: the SN260 can wake on both an internal timer and an external signal from deep sleep mode; power down mode can only wake on an external signal.

6 SPI protocol

The SN260 low level protocol centers on the SPI interface for communication with a pair of GPIO for handshake signaling.

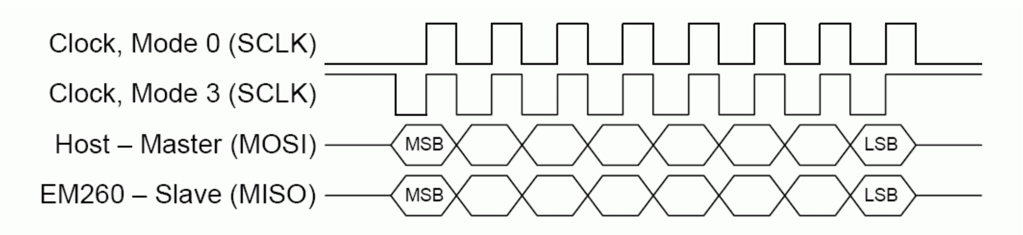
- The SN260 looks like a hardware peripheral.
- The SN260 is the slave device and all transactions are initiated by the Host (the master).
- The SN260 supports a reasonably high data rate.

6.1 Physical interface configuration

The SN260 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) at a maximum SPI clock rate of 5MHz, as illustrated in *Figure 3*.

Note: The convention for the waveforms in this document is to show Mode 0.

Figure 3. SPI transfer format, Mode 0 and Mode 3

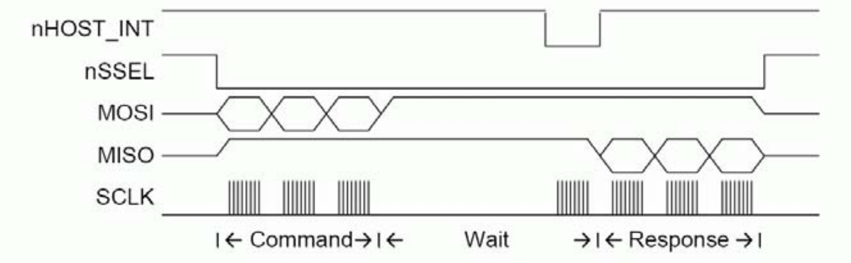


The nHOST_INT signal and the nWAKE signal are both active low. The Host must supply a pull-up resistor on the nHOST_INT signal to prevent errant interruptions during undefined events such as the SN260 resetting. The SN260 supplies an internal pull-up on the nWAKE signal to prevent errant interruptions during undefined events such as the Host resetting.

6.2 SPI transaction

The basic SN260 SPI transaction is half-duplex to ensure proper framing and to give the SN260 adequate response time. The basic transaction, as shown in *Figure 4*, 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 4. General timing diagram for a SPI transaction



6.2.1 Command section

The Host begins the transaction by asserting the Slave Select and then sending a command to the SN260. This command can be of any length from 2 to 128 bytes and must not begin with `0xFF`. During the Command section, the SN260 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.

6.2.2 Wait section

The Wait section is a period of time during which the SN260 may be processing the command or performing other operations. Note that this section can be any length of time up to 200 milliseconds. 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 SN260 can require up to 200 milliseconds to respond, as long as the Host keeps Slave Select active, the Host can perform other tasks while waiting for a Response.

To determine when a Response is ready, use one of two methods:

- Clock the SPI until the SN260 transmits a byte other than `0xFF`.
- Interrupt on the falling edge of nHOST_INT.

The first method, clocking the SPI, is recommended due to simplicity in implementing. During the Wait section, the SN260 will transmit only `0xFF` and will ignore all incoming data until the Response is ready. When the SN260 transmits a byte other than `0xFF`, the transaction has officially moved into the Response section. Therefore, the Host can poll for a Response by continuing to clock the SPI by transmitting `0xFF` and waiting for the SN260 to transmit a byte other than `0xFF`. The SN260 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.

6.2.3 Response section

When the SN260 transmits a byte other than `0xFF`, the transaction has officially moved into the Response section. The data format is the same format used in the Command section. The response can be of any length from 2 to 128 bytes and will not begin with `0xFF`. Depending on the actual response, the length of the response is known from the first or second byte and this length should be used by the Host to clock out exactly the correct number of bytes. Once all bytes have been clocked, it is allowable for the Host to de-assert chip select. Since the Host is in control of clocking the SPI, there are no ACKs or similar signals needed back from the Host because the SN260 will assume 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 1ms. This timing requirement is called the inter-command spacing and is necessary to allow the SN260 to process a command and become ready to accept a new command.

6.2.4 Asynchronous signaling

When the SN260 has data to send to the Host, it will assert the nHOST_INT signal. 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 SN260 for its output. The Host should initiate this transaction as soon as possible to prevent possible backup of data in the SN260. The SN260 will de-assert the nHOST_INT signal after receiving a byte on the SPI. Due to inherent latency in the SN260, 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 10µs before asserting again. If the SN260 has more output available after the transaction has completed, the nHOST_INT signal will assert again after Slave Select is de-asserted and the Host must make another request.

6.2.5 Spacing

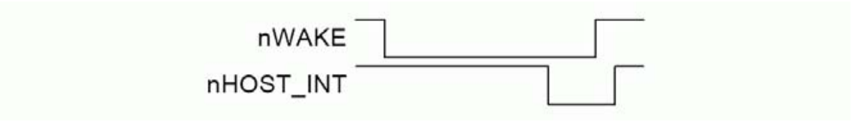
To ensure that the SN260 is always able to deal with incoming commands, a minimum inter-command spacing is defined at 1ms. After every transaction, the Host must hold the Slave Select high for a minimum of 1ms. The Host must respect the inter-command spacing requirement, or the SN260 will not have time 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 SN260 is ready to accept a command.

6.2.6 Waking the SN260 from sleep

Waking up the SN260 involves a simple handshaking routine as illustrated in *Figure 5*. This handshaking ensures that the Host will wait until the SN260 is fully awake and ready to accept commands from the Host. If the SN260 is already awake when the handshake is performed (such as when the Host resets and the SN260 is already operating), the handshake will proceed as described below with no ill effects.

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

Figure 5. SN260 wake sequence



Waking the SN260 involves the following steps:

1. Host asserts nWAKE.
2. SN260 interrupts on nWAKE and exits sleep.
3. SN260 performs all operations it needs to and will not respond until it is ready to accept commands.
4. SN260 asserts nHOST_INT within 10ms of nWAKE asserting. If the SN260 does not assert nHOST_INT within 10ms of nWAKE, it is valid for the Host to consider the SN260 unresponsive and to reset the SN260.
5. Host detects nHOST_INT assertion. Since the assertion of nHOST_INT indicates the SN260 can accept SPI transactions, the Host does not need to hold Slave Select high for the normally required minimum 1ms of inter-command spacing.
6. Host de-asserts nWAKE after detecting nHOST_INT assertion.
7. SN260 will de-assert nHOST_INT within 25s of nWAKE de-asserting.
8. After 25µs, any change on nHOST_INT will be an indication of a normal asynchronous (callback) event.

6.2.7 Error conditions

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 SN260.

- **Oversized EZSP frame**

  If the transaction includes an EZSP Frame, the Length Byte cannot be a value greater than 125. If the SN260 detects a length byte greater than 125, it will drop the incoming Command and abort the entire transaction. The SN260 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. Not only is the Command in the problematic transaction dropped by the SN260, but the next Command is also dropped, because it is responded to with the Oversized EZSP 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 SN260 will assert nHOST_INT to inform the Host through an error code in the Response section what has happened. When a transaction is aborted, not only does the Command in the problematic transaction get dropped by the SN260, but the next Command also gets dropped since 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 SN260 will drop any Command that is missing the Frame Terminator. The SN260 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, since the SN260 will send only `0xFF` outside of the Response section as well as ignore incoming bytes outside of the Command section.
- **Unresponsive**

  Unresponsive can mean the SN260 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 SN260 unresponsive to the Command section. This maximum length is 200 milliseconds, measured from the end of the last byte sent in the Command Section. If the SN260 ever fails to respond during the Wait section, it is valid for the Host to consider the SN260 unresponsive and to reset the SN260. Additionally, if nHOST_INT does not assert within 10ms of nWAKE asserting during the wake handshake, the Host can consider the SN260 unresponsive and reset the SN260.

6.3 SPI protocol timing

Figure 6 illustrates all critical timing parameters in the SPI Protocol. These timing parameters are a result of the SN260's internal operation and both constrain Host behavior and characterize SN260 operation. The parameters shown are discussed elsewhere in this document. Note that *Figure 6* is not drawn to scale, but is provided to illustrate where the parameters are measured.

Figure 6. SPI protocol timing waveform

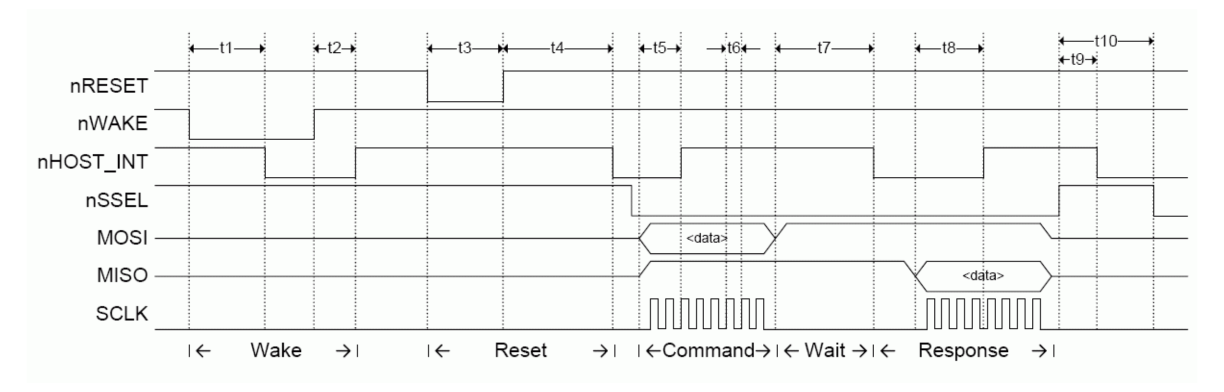


Table 14 lists the timing parameters of the SPI protocol. These parameters are illustrated in *Figure 6*.

Table 14. SPI protocol timing parameters

Parameter	Description	Min.	Typ.	Max.	Unit
t1 (a)	Wake handshake, while 260 is awake		133	150	s
t1 (b)	Wake handshake, while 260 is asleep		7.3	10	ms
t2	Wake handshake finish	1.1	1.2	25	s
t3	Reset pulse width	8			s
t4	Startup time		250	1500	ms
t5	nHOST_INT de-asserting after Command	13	35	75	s
t6	Clock rate	200			ns
t7	Wait section	25	755	200000	s
t8	nHOST_INT de-asserting after Response	20	130	800	s
t9	nHOST_INT asserting after transaction	25	70	800	s
t10	Inter-command spacing	1			ms

6.4 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 *Figure 7*.

Figure 7. SPI protocol data format

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

The total length of a command must not exceed 128 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 EZSP Frame (EmberZNet Serial Protocol Frame) and the Length Byte defines the length of just the EZSP Frame. Therefore, if a command includes an EZSP Frame, the Length Byte can have a value from 2 through 125 and the overall command size will be 5 through 128 bytes. The SPI Byte can be a specific value indicating if there is an EZSP Frame or not, and if there is an EZSP 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 *EZSP Frame* contains the data needed for operating EmberZNet. The EZSP Frame and its format are explained in *Section 7: EmberZNet serial protocol on page 34*.

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 SN260 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 SN260 just reset and the corrupt Response should be discarded.

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

6.5 SPI byte

Table 15 lists the possible commands and their responses in the SPI Byte.

Table 15. SPI commands & responses

Command value	Command	Response value	Response
Any	Any	0x00	SN260 reset occurred—This is never used in another response; it always indicates an SN260 Reset.
Any	Any	0x01	Oversized EZSP Frame received—This is never used in another response; it always indicates an overflow occurred.
Any	Any	0x02	Aborted Transaction occurred—This is never used in another response; it always indicates an aborted transaction occurred.
Any	Any	0x03	Missing Frame Terminator—This is never used in another response; it always indicates a missing frame terminator in the command.
Any	Any	0x04	Reserved
0x00 – 0x0F	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 1–63.
0x0B	SPI Status	0xC0 – 0xC1	bit[7] is always set. bit[6] is always set. bit[0]—Set if Alive.
0xF0 – 0xFD	Reserved	[none]	[none]
0xFE	EZSP Frame	0xFE	EZSP frame
0xFF	Invalid	0xFF	Invalid

6.5.1 Primary SPI bytes

There are three primary SPI bytes: SPI protocol version, SPI status, 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 `0x81`, because the version number corresponding to this set of Command-Response values is version number 1. The version number can be a value from 1 to 63 (`0x81-0xBF`).
- SPI status [0x0B]

  Sending this command asks for the SN260 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 SN260 is alive and ready for commands.
- EZSP frame [0xFE]

  This byte indicates that the current transaction is an EZSP transaction and there is more data to follow. This SPI Byte, and only this SPI Byte, will cause the transaction to look like the full data format illustrated in *Figure 7*. 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 section *Section 7: EmberZNet serial protocol on page 34*. If the SPI Byte is `0xFE`, it means the minimum transaction size is five bytes. All other SPI Bytes mean the transaction size is two or three bytes.

6.5.2 Special response bytes

There are only five SPI Byte values, `0x00-0x04`, ever used as error codes (see *Table 16*). When the error condition occurs, any command sent to the SN260 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.

Table 16. Byte values used as error codes

SPI byte value	Error message	Error description	Error byte description
0x00	SN260 Reset	See *Section 6.6: Powering on, power cycling, and rebooting*.	The reset type. Refer to the API documentation discussing EmberResetType.
0x01	Oversized EZSP Frame	The command contained an EZSP frame with a Length Byte greater than 125. The SN260 was forced to drop the entire command.	Reserved
0x02	Aborted Transaction	The transaction was not completed properly and the SN260 was forced to abort the transaction.	Reserved
0x03	Missing Frame Terminator	The command was missing the Frame Terminator. The SN260 was forced to drop the entire command.	Reserved
0x04	Reserved	[none]	[none]

6.6 Powering on, power cycling, and rebooting

When the Host powers on (or reboots), it cannot guarantee that the SN260 is awake and ready to receive commands. Therefore, the Host should always perform the Wake SN260 handshake to guarantee that the SN260 is awake. If the SN260 resets, it needs to inform the Host so that the Host can reconfigure the stack if needed.

When the SN260 resets, it will assert the nHOST_INT signal, telling the Host that it has data. The Host should request data from the SN260 as usual. The SN260 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 `EmberResetType`. This specialty SPI Byte is never used in another Response SPI Byte. If the Host sees `0x00` from the SN260, it knows that the SN260 has been reset. The SN260 will de-assert 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 SN260, the EmberZNet Serial Protocol also provides a mechanism for a software reboot.

6.6.1 Unexpected resets

The SN260 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 SN260's internal SPI Protocol uses Slave Select returning to idle as a trigger to re-initialize its SPI Protocol. By always re-initializing, the SN260 is protected against the Host unexpectedly resetting or terminating a transaction. Additionally, if Slave Select is active when the SN260 powers on, the SN260 will ignore SPI data until Slave Select returns to idle. By ignoring SPI traffic until idle, the SN260 will not begin receiving in the middle of a transaction.

If the Host resets, in most cases it should reset the SN260 as well so that both devices are once again in the same state: freshly booted. Alternately, the Host can attempt to recover from the reset by recovering its previous state and resynchronizing with the state of the SN260.

If the SN260 resets during a transaction, the Host can expect either a Wait Section timeout or a missing Frame Terminator indicating an invalid Response.

If the SN260 resets outside of a transaction, the Host should proceed normally.

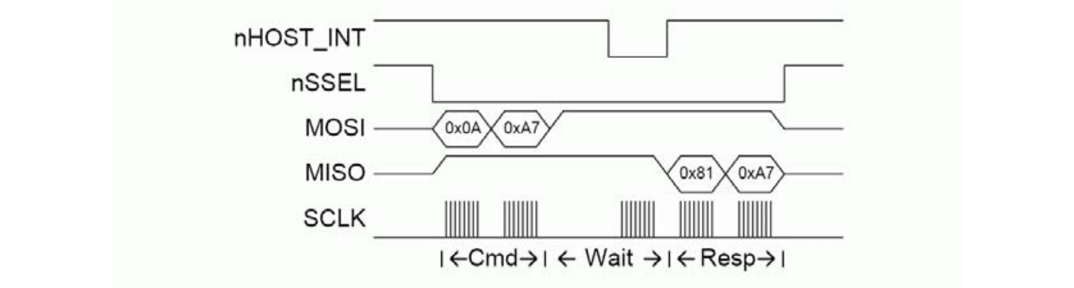
6.7 Transaction examples

This section contains the following transaction examples:

- SPI protocol version
- EmberZNet serial protocol frame — NOP command
- SN260 reset
- Three-part transaction: Wake, Get Version, Stack Status Callback

6.7.1 SPI protocol version

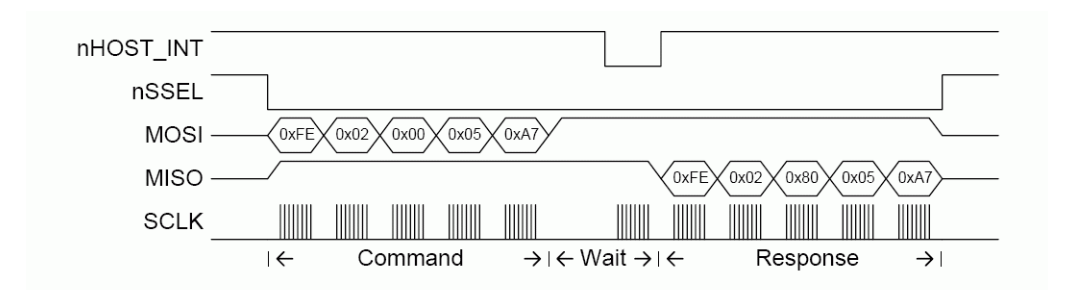
Figure 8. SPI protocol version example



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 `0x81` (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 1.
8. De-activate Slave Select.

6.7.2 EmberZNet serial protocol frame — NOP command

Figure 9. EmberZNet serial protocol frame - NOP command example

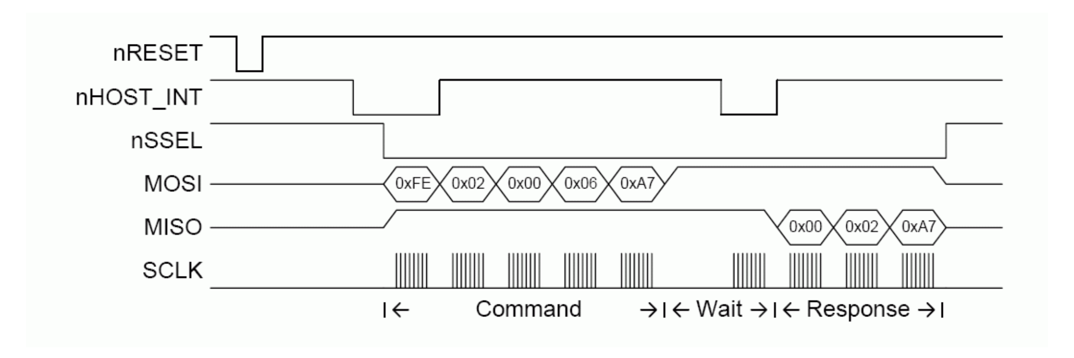


1. Activate Slave Select (nSSEL).
2. Transmit the appropriate command:
   - `0xFE`: SPI Byte indicating an EZSP Frame
   - `0x02`: Length Byte showing the EZSP Frame is 2 bytes long
   - `0x00`: EZSP Frame Control Byte indicating a command with no sleeping
   - `0x05`: EZSP Frame Type Byte indicating the `NOP` command
   - `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
   – `0x02`: Length Byte showing the EZSP Frame is 2 bytes long
   – `0x80`: EZSP Frame Control Byte indicating a response with no overflow
   – `0x05`: EZSP Frame Type Byte indicating the `NOP` response
   – `0xA7`: Frame Terminator
8. De-activate Slave Select.

6.7.3 SN260 reset

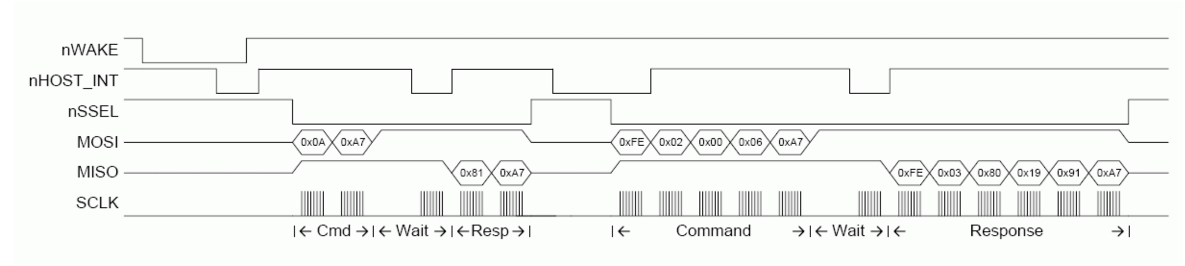
Figure 10. SN260 reset example



1. nHOST_INT asserts.
2. Activate Slave Select (nSSEL).
3. Transmit the command:
   – `0xFE`: SPI Byte indicating an EZSP Frame
   – `0x02`: Length Byte showing the EZSP Frame is 2 bytes long
   – `0x00`: EZSP Frame Control Byte indicating a command with no sleeping
   – `0x06`: EZSP Frame Type Byte indicating the `callback` command
   – `0xA7`: Frame Terminator
4. Wait for nHOST_INT to assert.
5. Transmit and receive `0xFF` until a byte other than `0xFF` is received.
6. Receive response `0x00` (a byte other than `0xFF`).
7. Receive the Error Byte and decode (`0x02` is enumerated as RESET_POWERON).
8. Receive the Frame Terminator (`0xA7`).
9. Response `0x00` indicates the SN260 has reset and the Host should respond appropriately.
10. Deactivate Slave Select.
11. Since nHOST_INT does not assert again, there is no more data for the Host.

6.7.4 Three-part transaction: Wake, Get Version, Stack Status Callback

Figure 11. Timing diagram of the three-part transaction



1. Activate nWAKE and activate timeout timer.
2. SN260 wakes up (if not already) awake and enables communication.
3. nHOST_INT asserts, indicating the SN260 can accept commands.
4. Host sees nHOST_INT activation within 10ms and deactivates nWAKE and timeout timer.
5. nHOST_INT de-asserts immediately after nWAKE.
6. Activate Slave Select.
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 `0x81` (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 1.
13. Deactivate Slave Select.
14. Host begins timing the inter-command spacing of 1ms 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 1ms before responding.
17. Activate Slave Select.
18. Transmit the command:
    - `0xFE`: SPI Byte indicating an EZSP Frame
    - `0x02`: Length Byte showing the EZSP Frame is 2 bytes long
    - `0x00`: EZSP Frame Control Byte indicating a command with no sleeping
    - `0x06`: EZSP Frame Type Byte indicating the `callback` command
    - `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
    - `0x03`: Length Byte showing the EZSP Frame is 3 bytes long
    - `0x80`: EZSP Frame Control Byte indicating a response with no overflow
    - `0x19`: EZSP Frame Type Byte indicating the `emberStackStatusHandler` command
    - `0x91`: `EmberStatus EMBER_NETWORK_DOWN` from `emberStackStatusHandler`
    - `0xA7` – Frame Terminator
24. Deactivate Slave Select.
25. Since nHOST_INT does not assert again, there is no more data for the Host.

7 EmberZNet serial protocol

EmberZnet Serial Protocol (EZSP)has been designed to be very familiarr to customers who have used the EmberZNet 2.x stack API. The majority of the commands and responses are functionally identical to those found in EmberZNet 2.x. The variations are due mainly to the timing differences of running the application on a separate processor across a serial interface. Communication between the SN260 and the Host consists of a two-message transaction. The Host sends a command message to the SN260 and then the SN260 sends a response message to the Host. If the SN260 needs to communicate asynchronously with the Host, it will indicate this by using the interrupt line and then waiting for the Host to send the `callback` command.

All EZSP frames begin with a Frame Control Byte followed by a Frame ID Byte. The format of the rest of the frame depends on the frame ID. *Section 7.3: Protocol format on page 38* defines the format for all the frame IDs. Most of the frames have a fixed length. A few, such as those containing application messages, are of variable length. The frame control indicates the direction of the message (command or response). For commands, the frame control also contains power management information, and for responses it also contains status information.

When a command contains an application message, the Host must supply a one-byte tag. This tag is used in future commands and responses to refer to the message. For example, when sending a message, the Host provides both the message contents and a tag. The tag is then used to report the fate of the message in a later response from the SN260.

7.1 Byte order

All multiple octet fields are transmitted and received with the least significant octet first, also referred to as little endian. This is the same byte order convention specified by 802.15.4 and ZigBee. Note that EUI64 fields are treated as a 64-bit number and are therefore transmitted and received in little endian order. Each individual octet is transmitted with the most significant bit first, as shown in *Section 6.1: Physical interface configuration on page 22*.

7.2 Conceptual overview

This section provides an overview of the concepts that are specific to the SN260 or that differ from the EmberZNet 2.x stack API. The commands and responses mentioned in this overview are described in more detail later in this document.

7.2.1 Stack configuration

The Host can use the `version` command to obtain information about the firmware running on the SN260. There are a number of configuration values that affect the behavior of the stack. The Host can read these values at any time using the `getConfigurationValue` command. After the SN260 has reset, the Host can modify any of the default values using the `setConfigurationValue` command. The Host must then provide information about the application endpoints using the `addEndpoint` command.

Table 17 gives the minimum, default and maximum values for each of the configuration values. Also listed is the RAM cost. This is the number of bytes of additional RAM required to increase the configuration value by one. Since the total amount of RAM is fixed, the additional RAM required must be made available by reducing one of the other configuration values.

Table 17. Configuration values

Value	Min.	Def.	Max.	Units	RAM Cost	Description
EZSP_CONFIG_PACKET_BUFFER_COUNT	5	24		packet buffers	40	The number of packet buffers available to the stack.
EZSP_CONFIG_NEIGHBOR_TABLE_SIZE	8	16	16	neighbors	18	The maximum number of router neighbors the stack can keep track of. A neighbor is a node within radio range.
EZSP_CONFIG_TRANSPORT_PACKET_COUNT	0	10		messages	10	The maximum number of datagram and sequenced messages the stack can have in the process of being either transmitted or received at any given time.
EZSP_CONFIG_BINDING_TABLE_SIZE (A)	0	8	32 + (B)	entries	3	The maximum number of bindings supported by the stack. It includes the bindings in EEPROM and in RAM.
EZSP_CONFIG_TEMPORARY_BINDING_ENTRIES (B)	0	8	(A)	entries	12	The number of binding table entries in RAM.
EZSP_CONFIG_TRANSPORT_CONNECTION_COUNT	0	0		entries	12	The number of binding table entries that can concurrently support an open sequenced connection.
EZSP_CONFIG_ROUTE_TABLE_SIZE (C)	0	16		entries	6	The maximum number of destinations to which a node can route messages. This include both messages originating at this node and those relayed for others.
EZSP_CONFIG_DISCOVERY_TABLE_SIZE	0	8		entries	10	The number of simultaneous route discoveries that a node will support.
EZSP_CONFIG_DISCOVERY_CACHE_ENDPOINTS (D)	0	4		endpoints	0	End-device child endpoints larger than this value will not have their discovery information cached by their router parent.
EZSP_CONFIG_DISCOVERY_CACHE_ENTRY_SIZE	11 + (D)	15	15	bytes	0	The size of an entry in the end device discovery cache on a router. Endpoint descriptions longer than this will not be cached.
EZSP_CONFIG_DISCOVERY_CACHE_SIZE	0	35	35	entries	0	The number of entries in the discovery cache on a router. Each end device child requires 1 + (D) entries. The cache is held in EEPROM.
EZSP_CONFIG_STACK_PROFILE	0	0			0	Specifies the stack profile.
EZSP_CONFIG_SECURITY_LEVEL	0	5	5		0	The security level used for security at the MAC and network layers. The supported values are 0 (no security) and 5 (payload is encrypted and a four-byte MIC is used for authentication).
EZSP_CONFIG_MAX_HOPS	0	10		hops	0	The maximum number of hops for a message.
EZSP_CONFIG_MAX_END_DEVICE_CHILDREN (E)	0	6	32	children	4	The maximum number of end device children that a router will support.
EZSP_CONFIG_INDIRECT_TRANSMISSION_TIMEOUT	0	3000	30000	milli-seconds	0	The maximum amount of time that the MAC will hold a message for indirect transmission to a child.

Table 17. Configuration values (continued)

Value	Min.	Def.	Max.	Units	RAM Cost	Description
`EZSP_CONFIG_RESERVED_ROUTING_ENTRIES`	0	0	(C)	entries	0	The number of route table entries that are reserved for temporary aggregation routes in the mesh stack.
`EZSP_CONFIG_MOBILE_NODE_POLL_TIMEOUT`	0	20		quarter seconds	0	The maximum amount of time that a mobile node can wait between polls. If no poll is heard within this timeout, then the parent removes the mobile node from its tables.
`EZSP_CONFIG_RESERVED_MOBILE_CHILD_ENTRIES`	0	0	(E)	entries	0	The number of child table entries reserved for use only by mobile nodes.
`EZSP_CONFIG_HOST_RAM`	0	0	255	bytes	1	The amount of RAM available for use by the Host.
`EZSP_CONFIG_TX_POWER_MODE`	0	0	3		0	Enables boost power mode and/or the alternate transmitter output.

7.2.2 Policy settings

There are some situations when the SN260 must make a decision but there isn't enough time to consult with the Host. The Host can control what decision is made by setting the policy in advance. The SN260 will then make decisions according to the current policy. The Host is informed via callbacks each time a decision is made, but by the time the news reaches the Host, it is too late to change that decision. You can change the policies at any time by using the `setPolicy` command.

A policy is used for trust center behavior, external binding modification requests, datagram replies, generating `pollHandler` callbacks, and the contents of the `unicastSent` and `messageSent` callbacks.

7.2.3 Datagram replies

The policy for datagram replies allows the Host to decide whether it wants to supply the SN260 with a reply payload for every datagram received. If the Host sets the policy to not supply a reply, the SN260 will automatically send an empty reply (containing no payload) for every datagram received. If the Host sets the policy to supply the reply, then the SN260 will only send a reply when instructed by the Host.

If the reply does not reach the sender before the transport retry timeout expires, the sender will transmit the datagram again. The Host must process the incoming message and supply the reply quickly enough to avoid retransmission by the sender. Provided this timing constraint is met, multiple datagrams can be received before the first reply is supplied and the replies can be supplied in any order.

7.2.4 Callbacks

Asynchronous callbacks from the SN260 are sent to the Host as the response to a `callback` command. The SN260 uses the interrupt line to indicate that the Host should send a `callback` command. The SN260 will queue multiple callbacks while it waits for the Host, and each response only delivers one callback. If the SN260 receives the `callback` command when there are no pending callbacks, it will reply with the `noCallbacks` response.

7.2.5 Power management

The SN260 will always idle its processor whenever possible. To further reduce power consumption, the SN260 can be put to sleep by the Host. In power down mode, only an external interrupt will wake the SN260. In deep sleep mode, the SN260 will use its internal timer to wake up for scheduled events. The SN260 provides two independent timers that the Host can use for any purpose, including waking up the SN260 from deep sleep mode. Timers are set using the `setTimer` command and generate `timerHandler` callbacks.

The initial frame control byte of every command tells the SN260 which sleep mode to enter after it has responded to the command. Including this information in every command (instead of having a separate power management command) allows the SN260 to be put to sleep faster. If the Host needs to put the SN260 to sleep without also performing another action, the `nop` command can be used.

In deep sleep mode, the SN260 will wake up for an internal event. If the event does not produce a callback for the Host, the SN260 will go back to sleep once the event has been handled. If the event does produce a callback, the SN260 will signal the Host and remain awake waiting for the `callback` command. If the frame control byte of the `callback` command specifies deep sleep mode, then the SN260 would normally go back to sleep after responding with the callback. However, if there is a second callback pending, the SN260 will remain awake waiting for another `callback` command.

To avoid disrupting the operation of the network, only put the SN260 to sleep when it is not joined to a network or when it is joined as a sleeping end device. If the SN260 is joined as a sleeping end device, then it must poll its parent in order to receive messages. The Host controls the polling behavior using the `pollForData` command. Polls are sent periodically with the interval set by the Host or a single poll can be sent. The result of every poll attempt is optionally reported using the `pollCompleteHandler` callback.

7.2.6 Tokens

Some of the non-volatile storage on the SN260 is made available for use by the Host. Up to 8 manufacturing tokens stored in the flash information area can be read using the `getMfgToken` command and up to 8 tokens stored in the simulated EEPROM can be read and written using the `setToken` and `getToken` commands. Each token is 8 bytes. Tokens preserve their values between reboots. Refer to section Simulated EEPROM for a description of the simulated EEPROM and write cycle estimates.

7.2.7 RAM

Some of the RAM on the SN260 can be reserved by the Host for its own use. The amount of space reserved is the `EZSP_CONFIG_HOST_RAM` configuration value (set using the `setConfigurationValue` command). The Host can then read and write data using the `setRam` and `getRam` commands. If the Host chooses to reserve RAM, this will reduce the number of messages and callbacks that the SN260 can buffer.

7.2.8 SN260 status

The frame control byte of every response sent by the SN260 contains two status bits:

- The overflow bit is set if the SN260 ran out of memory at any time since the previous response was sent. If this bit is set, then messages may have been lost.
- The truncated bit is set if the SN260 truncated the current response. If this bit is set, the command from the Host produced a response larger than the maximum EZSP frame length.

You can use the `nop` command to check the status of the SN260 without also performing another action.

7.2.9 Random number generator

The Host can obtain a random number from the SN260 using the `getRandomNumber` command. The random number is generated from analog noise in the radio and can be used to seed a random number generator on the Host.

7.3 Protocol format

All EZSP frames begin with a frame control byte. *Table 18* describes the meaning of this byte for command and response frames. *Table 19* describes the sleep modes, *Table 20* describes the overflow status bit and *Table 21* describes the truncated status bit. The second byte of all EZSP frames is the frame ID byte.

Table 18. Frame control byte

Bit	Command	Response
7 (MSB)	0	1
6	0 (reserved)	0 (reserved)
5	0 (reserved)	0 (reserved)
4	0 (reserved)	0 (reserved)
3	0 (reserved)	0 (reserved)
2	0 (reserved)	0 (reserved)
1	sleepMode[1]	truncated
0 (LSB)	sleepMode[0]	overflow

Table 19. Sleep modes

sleepMode[1]	sleepMode[0]	Description
1	1	Reserved.
1	0	Power down.
0	1	Deep sleep.
0	0	Idle.

Table 20. Overflow status

overflow	Description
1	The SN260 ran out of memory since the previous response.
0	No memory shortage since the previous response.

Table 21. Truncated status

truncated	Description
1	The SN260 truncated the current response to avoid exceeding the maximum EZSP frame length.
0	The current response was not truncated.

Section 7.3.1: Type definitions defines all the types used by the SN260 and *Section 7.3.2: Structure definitions* defines all the structures. *Section 7.3.3: Named values* enumerates all the named values for the different types. The subsequent sections list all the frames supported by the SN260, specifying the Frame ID, the command parameters and the response parameters. The list is divided into five sections:

- *Section 7.3.4* lists Configuration frames.
- *Section 7.3.5* lists Utilities frames.
- *Section 7.3.6* lists Networking frames.
- *Section 7.3.7* lists Binding frames.
- *Section 7.3.8* lists Messaging frames.

Finally, section *Section 7.3.9* provides an alphabetical list of all the frames.

7.3.1 Type definitions

Table 22. Type definitions

Type	Alias	Description
boolean	int8u	True or false.
EzspConfigId	int8u	Identifies a configuration value.
EzspConfigTxPowerMode	int16u	Values for EZSP_CONFIG_TX_POWER_MODE.
EzspConfigStatus	int8u	Return type for configuration commands.
EzspPolicyId	int8u	Identifies a policy.
EzspDecisionId	int8u	Identifies a policy decision.
EmberStatus	int8u	Return type for stack functions.
EmberEventUnits	int8u	Either marks an event as inactive or specifies the units for the event execution time.
EmberNodeType	int8u	The type of the node.
EmberNetworkStatus	int8u	The possible join states for a node.
EmberIncomingMessageType	int8u	Incoming message types.
EmberBindingType	int8u	Binding types.
EmberUnicastOption	int8u	Options to use when sending a unicast message.
EmberNetworkScanType	int8u	Network scan types.
EmberJoinDecision	int8u	Decision made by the trust center when a node attempts to join.
EmberNodeId	int16u	16-bit ZigBee network address.
EmberPanId	int16u	802.15.4 PAN ID.
EmberEUI64	int8u[8]	EUI 64-bit ID (an IEEE address).

7.3.2 Structure definitions

Table 23. Structure definitions

Structure	Field	Description
EmberNetworkParameters		**Network parameters.**
	int16u panId	The network's PAN identifier.
	int8s radioTxPower	A power setting, in dBm.
	int8u radioChannel	A radio channel.
EmberApsFrame		**ZigBee APS frame parameters.**
	int16u profileId	The application profile ID that describes the format of the message.
	int8u clusterId	The cluster ID for this message.
	int8u sourceEndpoint	The source endpoint.
	int8u destinationEndpoint	The destination endpoint.
	EmberUnicastOption options	A bitmask of options.
EmberBindingTableEntry		**An entry in the binding table.**
	EmberBindingType type	The type of binding.
	int8u local	The endpoint on the local node.
	int8u remote	The endpoint on the remote node (specified by identifier).
	int8u clusterId	A cluster ID that matches one from the local endpoint's simple descriptor. This cluster ID is set by the provisioning application to indicate which part an endpoint's functionality is bound to this particular remote node and is used to distinguish between unicast and multicast bindings. A binding can be used to send messages with any cluster ID, not just the one listed in the binding.
	EmberEUI64 identifier	A 64-bit identifier. This is either the destination EUI64 (for unicasts) or the 64-bit group address (for multicasts).

7.3.3 Named values

Table 24. boolean

Structure	Field	Description
FALSE	0x00	An alias for zero, used for clarity.
TRUE	0x01	An alias for one, used for clarity.

Table 25. EzspConfigId

Structure	Field	Description
EZSP_CONFIG_PACKET_BUFFER_COUNT	0x01	The number of packet buffers available to the stack.
EZSP_CONFIG_NEIGHBOR_TABLE_SIZE	0x02	The maximum number of router neighbors the stack can keep track of. A neighbor is a node within radio range.
EZSP_CONFIG_TRANSPORT_PACKET_COUNT	0x03	The maximum number of datagram and sequenced messages the stack can have 'in-flight' at any time. Here, 'in-flight' means 'in the process of being either transmitted or received'.
EZSP_CONFIG_BINDING_TABLE_SIZE	0x04	The maximum number of bindings supported by the stack. It includes the bindings in EEPROM and in RAM.
EZSP_CONFIG_TEMPORARY_BINDING_ENTRIES	0x05	The number of binding table entries in RAM.
EZSP_CONFIG_TRANSPORT_CONNECTION_COUNT	0x06	The number of binding table entries that can concurrently support an open sequenced connection.
EZSP_CONFIG_ROUTE_TABLE_SIZE	0x07	The maximum number of destinations to which a node can route messages. This include both messages originating at this node and those relayed for others.
EZSP_CONFIG_DISCOVERY_TABLE_SIZE	0x08	The number of simultaneous route discoveries that a node will support.
EZSP_CONFIG_DISCOVERY_CACHE_ENDPOINTS	0x09	End-device child endpoints larger than this value will not have their discovery information cached by their router parent.
EZSP_CONFIG_DISCOVERY_CACHE_ENTRY_SIZE	0x0A	The size of an entry in the end device discovery cache on a router. Endpoint descriptions longer than this will not be cached.
EZSP_CONFIG_DISCOVERY_CACHE_SIZE	0x0B	The number of entries in the discovery cache on a router. Each end device child requires 1 + EZSP_CONFIG_DISCOVERY_CACHE_ENDPOINTS entries. The cache is held in EEPROM.
EZSP_CONFIG_STACK_PROFILE	0x0C	Specifies the stack profile.
EZSP_CONFIG_SECURITY_LEVEL	0x0D	The security level used for security at the MAC and network layers. The supported values are 0 (no security) and 5 (payload is encrypted and a four-byte MIC is used for authentication).
EZSP_CONFIG_MAX_HOPS	0x10	The maximum number of hops for a message.
EZSP_CONFIG_MAX_END_DEVICE_CHILDREN	0x11	The maximum number of end device children that a router will support.

Table 25. EzspConfigId (continued)

Structure	Field	Description
EZSP_CONFIG_INDIRECT_TRANSMISSION TIMEOUT	0x12	The maximum amount of time that the MAC will hold a message for indirect transmission to a child.
EZSP_CONFIG_RESERVED_ROUTING_ ENTRIES	0x13	The number of route table entries that are reserved for temporary aggregation routes in the mesh stack.
EZSP_CONFIG_MOBILE_NODE_POLL_ TIMEOUT	0x14	The maximum amount of time that a mobile node can wait between polls. If no poll is heard within this timeout, then the parent removes the mobile node from its tables.
EZSP_CONFIG_RESERVED_MOBILE_CHILD ENTRIES	0x15	The number of child table entries reserved for use only by mobile nodes.
EZSP_CONFIG_HOST_RAM	0x16	The amount of RAM available for use by the Host.
EZSP_CONFIG_TX_POWER_MODE	0x17	Enables boost power mode and/or the alternate transmitter output.

Table 26. EzspConfigTxPowerMode

Structure	Field	Description
EMBER_TX_POWER_MODE_DEFAULT	0x00	Normal power mode and bi-directional RF transmitter output.
EMBER_TX_POWER_MODE_BOOST	0x01	Enable boost power mode. This is a high performance radio mode which offers increased receive sensitivity and transmit power at the cost of an increase in power consumption.
EMBER_TX_POWER_MODE_ALTERNATE	0x02	Enable the alternate transmitter output. This allows for simplified connection to an external power amplifier via the RF_TX_ALT_P and RF_TX_ALT_N pins.
EMBER_TX_POWER_MODE_BOOST_AND_ ALTERNATE	0x03	Enable both boost mode and the alternate transmitter output.

Table 27. EzspConfigStatus

Structure	Field	Description
EZSP_CONFIG_SUCCESS	0x00	The command was successful.
EZSP_CONFIG_OUT_OF_MEMORY	0x01	Insufficient memory was available.
EZSP_CONFIG_INVALID_VALUE	0x02	The value was out of bounds.
EZSP_CONFIG_INVALID_TAG	0x03	The configuration tag was not recognized.
EZSP_CONFIG_INVALID_CALL	0x04	Configuration values can no longer be modified.

Table 28. EzspPolicyId

Structure	Field	Description
EZSP_TRUST_CENTER_POLICY	0x00	Controls trust center behavior.
EZSP_BINDING_MODIFICATION_POLICY	0x01	Controls how external binding modification requests are handled.
EZSP_DATAGRAM_REPLIES_POLICY	0x02	Controls whether the Host supplies datagram replies.

Table 28. EzspPolicyId (continued)

Structure	Field	Description
EZSP_POLL_HANDLER_POLICY	0x03	Controls whether pollHandler callbacks are generated.
EZSP_MESSAGE_CONTENTS_IN_CALLBACK_POLICY	0x04	Controls whether the message contents are included in unicastSent and messageSent callbacks.

Table 29. EzspDecisionId

Structure	Field	Description
EZSP_ALLOW_SECURE_JOINS_ONLY	0x00	EZSP_TRUST_CENTER_POLICY default decision. Only allow nodes that are joining securely using the network key to join.
EZSP_ALLOW_ALL_JOINS	0x01	EZSP_TRUST_CENTER_POLICY decision. Allow all nodes to join, sending the key to nodes that are not joining securely.
EZSP_DISALLOW_ALL_JOINS	0x02	EZSP_TRUST_CENTER_POLICY decision. Reject all join attempts.
EZSP_ASK_TRUST_CENTER	0x03	EZSP_TRUST_CENTER_POLICY decision. Forward the request to the trust center (this value should not be used for the trust center itself).
EZSP_DISALLOW_BINDING_MODIFICATION	0x10	EZSP_BINDING_MODIFICATION_POLICY default decision. Do not allow the local binding table to be changed by remote nodes.
EZSP_ALLOW_BINDING_MODIFICATION	0x11	EZSP_BINDING_MODIFICATION_POLICY decision. Allow remote nodes to change the local binding table.
EZSP_HOST_WILL_NOT_SUPPLY_REPLY	0x20	EZSP_DATAGRAM_REPLIES_POLICY default decision. The SN260 will automatically send an empty reply (containing no payload) for every datagram received.
EZSP_HOST_WILL_SUPPLY_REPLY	0x21	EZSP_DATAGRAM_REPLIES_POLICY decision. The SN260 will only send a reply if it receives a sendReply command from the Host.
EZSP_POLL_HANDLER_IGNORE	0x30	EZSP_POLL_HANDLER_POLICY default decision. Do not inform the Host when a child polls.
EZSP_POLL_HANDLER_CALLBACK	0x31	EZSP_POLL_HANDLER_POLICY decision. Generate a pollHandler callback when a child polls.
EZSP_MESSAGE_TAG_ONLY_IN_CALLBACK	0x40	EZSP_MESSAGE_CONTENTS_IN_CALLBACK_POLICY default decision. Include only the message tag in unicastSent and messageSent callbacks.
EZSP_MESSAGE_TAG_AND_CONTENTS_IN_CALLBACK	0x41	EZSP_MESSAGE_CONTENTS_IN_CALLBACK_POLICY decision. Include both the message tag and the message contents in unicastSent and messageSent callbacks.

Table 30. EmberStatus

Structure	Field	Description
EMBER_SUCCESS	0x00	Generic 'no error' message.
EMBER_ERR_FATAL	0x01	Generic 'fatal error' message.
EMBER_EEPROM_MFG_STACK_VERSION_MISMATCH	0x04	Manufacturing and stack token format in non-volatile memory is different than what the stack expects (returned at initialization).
EMBER_INCOMPATIBLE_STATIC_MEMORY_DEFINITIONS	0x05	Static memory definitions in ember-static-memory.h are incompatible with this stack version.
EMBER_EEPROM_MFG_VERSION_MISMATCH	0x06	Manufacturing token format in non-volatile memory is different than what the stack expects (returned at initialization).
EMBER_EEPROM_STACK_VERSION_MISMATCH	0x07	Stack token format in non-volatile memory is different than what the stack expects (returned at initialization).
EMBER_NO_BUFFERS	0x18	There are no more buffers.
EMBER_SERIAL_INVALID_BAUD_RATE	0x20	Specified an invalid baud rate.
EMBER_SERIAL_INVALID_PORT	0x21	Specified an invalid serial port.
EMBER_SERIAL_TX_OVERFLOW	0x22	Tried to send too much data.
EMBER_SERIAL_RX_OVERFLOW	0x23	There was not enough space to store a received character and the character was dropped.
EMBER_SERIAL_RX_FRAME_ERROR	0x24	Detected a UART framing error.
EMBER_SERIAL_RX_PARITY_ERROR	0x25	Detected a UART parity error.
EMBER_SERIAL_RX_EMPTY	0x26	There is no received data to process.
EMBER_SERIAL_RX_OVERRUN_ERROR	0x27	Receive interrupt was not handled in time, and a character was dropped.
EMBER_MAC_TRANSMIT_QUEUE_FULL	0x39	MAC transmit queue is full.
EMBER_MAC_UNKNOWN_HEADER_TYPE	0x3A	MAC header FCR error on receive.
EMBER_MAC_SCANNING	0x3D	MAC can't complete this task because it is scanning.
EMBER_MAC_NO_DATA	0x31	No pending data exists for device doing a data poll.
EMBER_MAC_JOINED_NETWORK	0x32	Attempt to scan when we are joined to a network.
EMBER_MAC_BAD_SCAN_DURATION	0x33	Scan duration must be 0 to 14 inclusive. Attempt was made to scan with an incorrect duration value.
EMBER_MAC_INCORRECT_SCAN_TYPE	0x34	emberStartScan was called with an incorrect scan type.
EMBER_MAC_INVALID_CHANNEL_MASK	0x35	emberStartScan was called with an invalid channel mask.
EMBER_MAC_COMMAND_TRANSMIT_FAILURE	0x36	Failed to scan current channel because we were unable to transmit the relevant MAC command.
EMBER_MAC_NO_ACK_RECEIVED	0x40	We expected to receive an ACK following the transmission, but the MAC level ACK was never received.
EMBER_MAC_INDIRECT_TIMEOUT	0x42	Indirect data message timed out before polled.

Table 30. EmberStatus (continued)

Structure	Field	Description
EMBER_SIM_EEPROM_ERASE_PAGE_GREEN	0x43	Simulated EEPROM is telling the application that there is at least one flash page to be erased. GREEN status means the current page has not filled above the ERASE_CRITICAL_THRESHOLD. The application should call the function halSimEepromErasePage() when it can to erase a page.
EMBER_SIM_EEPROM_ERASE_PAGE_RED	0x44	Simulated EEPROM is telling the application that there is at least one flash page to be erased. RED status means the current page has filled above the ERASE_CRITICAL_THRESHOLD. Due to the shrinking availability of write space, there is a danger of data loss. The application must call the function halSimEepromErasePage() as soon as possible to erase a page.
EMBER_SIM_EEPROM_FULL	0x45	Simulated EEPROM has run out of room to write any new data and the data trying to be set has been lost. This error code is the result of ignoring the SIM_EEPROM_ERASE_PAGE_RED error code. The application must call the function halSimEepromErasePage() to make room for any further calls to set a token.
EMBER_SIM_EEPROM_FLASH_WRITE_ FAILED	0x46	A fatal error has occurred while trying to write data to the Flash and the write verification has failed. The data in the flash cannot be trusted after this error, and it is possible this error is the result of exceeding the life cycles of the flash.
EMBER_SIM_EEPROM_INIT_1_FAILED	0x47	Attempt 1 to initialize the simulated EEPROM has failed. This failure means the information already stored in Flash (or a lack thereof), is fatally incompatible with the token information compiled into the code image being run.
EMBER_SIM_EEPROM_INIT_2_FAILED	0x48	Attempt 2 to initialize the simulated EEPROM has failed. This failure means Attempt 1 failed, and the token system failed to properly reload default tokens and reset the simulated EEPROM.
EMBER_SIM_EEPROM_INIT_3_FAILED	0x49	Attempt 3 to initialize the simulated EEPROM has failed. This failure means one or both of the tokens TOKEN_MFG_NVDATA_VERSION or TOKEN_STACK_NVDATA_VERSION were incorrect and the token system failed to properly reload default tokens and reset the simulated EEPROM.
EMBER_ERR_TOKEN_UNKNOWN	0x4B	An unknown flash token was specified.
EMBER_ERR_TOKEN_EXISTS	0x4C	Could not create new flash token because it already exists.
EMBER_ERR_TOKEN_INVALID_SIZE	0x4D	An incorrect size was specified when retrieving token data.
EMBER_ERR_TOKEN_READ_ONLY	0x4E	Couldn't write token because it is marked read-only.
EMBER_ERR_BOOTLOADER_TRAP_ TABLE_BAD	0x58	Bootloader received an invalid message (failed attempt to go into bootloader).

Table 30. EmberStatus (continued)

Structure	Field	Description
EMBER_ERR_BOOTLOADER_TRAP_UNKNOWN	0x59	Bootloader received an invalid message (failed attempt to go into bootloader).
EMBER_ERR_BOOTLOADER_NO_IMAGE	0x5A	Bootloader cannot complete the bootload operation because either an image was not found or the image exceeded memory bounds.
EMBER_TOO_MANY_CONNECTIONS	0x60	EMBER_TRANSPORT_CONNECTION_COUNT limit has been reached.
EMBER_CONNECTION_OPEN	0x61	A connection has either been opened or is already open.
EMBER_CONNECTION_FAILED	0x63	A connection experienced a catastrophic error. The connection is now closed and messages may have been lost.
EMBER_CONNECTION_CLOSED	0x64	Transport layer successfully closed a connection.
EMBER_CONNECTION_CLOSING	0x65	Transport layer is in process of closing a connection (waiting for a response from the remote device).
EMBER_DELIVERY_FAILED	0x66	Transport layer attempted to send or deliver a message, but it failed.
EMBER_BINDING_INDEX_OUT_OF_RANGE	0x69	This binding index is out of range of the current binding table.
EMBER_INVALID_BINDING_TERMINAL	0x6B	Could not set or find a binding index given the specified terminal.
EMBER_INVALID_BINDING_INDEX	0x6C	An invalid binding table index was given to a function.
EMBER_TERMINAL_HAS_MULTIPLE_BINDINGS	0x6F	Multiple binding table entries were found for the specified terminal.
EMBER_INVALID_CALL	0x70	API call is not allowed given the current state of the stack (for example, opening a connection from a sleepy node.).
EMBER_COST_NOT_KNOWN	0x71	Link cost to a node is not known.
EMBER_MAX_MESSAGE_LIMIT_REACHED	0x72	Maximum number of in-flight messages (such as EMBER_TRANSPORT_PACKET_COUNT) has been reached.
EMBER_CONNECTION_NOT_YET_OPEN	0x73	A connection is not open yet.
EMBER_MESSAGE_TOO_LONG	0x74	Message to be transmitted is too big to fit into a single over-the-air packet.
EMBER_BINDING_IS_ACTIVE	0x75	Application is trying to delete or overwrite a binding that is in use.
EMBER_EUI64_NOT_AVAILABLE	0x76	EUI64 is not available in the current packet.
EMBER_INCOMING_SEQUENCED_MESSAGES LOST	0x77	One or more sequenced messages failed to be received.
EMBER_ADC_CONVERSION_DONE	0x80	Conversion is complete.
EMBER_ADC_CONVERSION_BUSY	0x81	Conversion cannot be done because a request is being processed.
EMBER_ADC_CONVERSION_DEFERRED	0x82	Conversion is deferred until the current request has been processed.
EMBER_ADC_NO_CONVERSION_PENDING	0x84	No results are pending.

Table 30. EmberStatus (continued)

Structure	Field	Description
EMBER_SLEEP_INTERRUPTED	0x85	Sleeping (for a duration) has been abnormally interrupted and exited prematurely.
EMBER_PHY_TX_UNDERFLOW	0x88	Transmit hardware buffer underflowed.
EMBER_PHY_TX_INCOMPLETE	0x89	Transmit hardware did not finish transmitting a packet.
EMBER_PHY_INVALID_CHANNEL	0x8A	An unsupported channel setting was specified.
EMBER_PHY_INVALID_POWER	0x8B	An unsupported power setting was specified.
EMBER_PHY_TX_BUSY	0x8C	Packet cannot be transmitted because the physical MAC layer is currently transmitting a packet. (This is used for the MAC backoff algorithm.)
EMBER_PHY_UNKNOWN_RADIO_TYPE	0x8D	The software installed on the hardware doesn't recognize the hardware radio type.
EMBER_PHY_OSCILLATOR_CHECK_FAILED	0x8E	The software installed on the hardware doesn't recognize the hardware radio type.
EMBER_PHY_PARTIAL_PACKET	0x8F	PHY did not receive the entire packet it was expecting from the radio.
EMBER_NETWORK_UP	0x90	Stack software has completed initialization and is ready to send and receive packets over the air.
EMBER_NETWORK_DOWN	0x91	Network is not operating.
EMBER_NETWORK_PENDING_ACTIVITY	0x92	Network has activity pending and should not be shut down.
EMBER_NOT_JOINED	0x93	Node has not joined a network.
EMBER_JOIN_FAILED	0x94	An attempt to join a network failed.
EMBER_INVALID_SECURITY_LEVEL	0x95	The chosen security level (the value of EMBER_SECURITY_LEVEL) is not supported by the stack.
EMBER_MOVE_FAILED	0x96	After moving, a mobile node's attempt to re-establish contact with the network failed.
EMBER_NETWORK_BUSY	0xA1	A message cannot be sent because the network is currently overloaded.
EMBER_NODEID_INVALID	0xA2	A datagram was sent to a node and the EUI64 address in the datagram did not match the node's EUI64 address. The NodeId was invalid.
EMBER_INVALID_ENDPOINT	0xA3	The application tried to send a message using an endpoint that it has not defined.
EMBER_BINDING_HAS_CHANGED	0xA4	The application tried to use a binding that has been remotely modified and the change has not yet been reported to the application.
EMBER_STACK_AND_HARDWARE_MISMATCH	0xB0	A critical and fatal error indicating that the version of the stack trying to run does not match with the chip it is running on. The software (stack) on the chip must be replaced with software that is compatible with the chip.

Table 31. EmberEventUnits

Structure	Field	Description
EMBER_EVENT_INACTIVE	0x00	Event is not scheduled to run.
EMBER_EVENT_MS_TIME	0x01	Execution time is in approximate milliseconds.
EMBER_EVENT_QS_TIME	0x02	Execution time is in 'binary' quarter seconds (256 approximate milliseconds each).
EMBER_EVENT_MINUTE_TIME	0x03	Execution time is in 'binary' minutes (65536 approximate milliseconds each).

Table 32. EmberNodeType

Structure	Field	Description
EMBER_COORDINATOR	0x01	Will relay messages and can act as a parent to other nodes.
EMBER_ROUTER	0x02	Will relay messages and can act as a parent to other nodes.
EMBER_END_DEVICE	0x03	Communicates only with its parent and will not relay messages.
EMBER_SLEEPY_END_DEVICE	0x04	An end device whose radio can be turned off to save power. The application must poll to receive messages.
EMBER_MOBILE_END_DEVICE	0x05	A sleepy end device that can move through the network.

Table 33. EmberNetworkStatus

Structure	Field	Description
EMBER_NO_NETWORK	0x00	The node is not associated with a network in any way.
EMBER_JOINING_NETWORK	0x01	The node is currently attempting to join a network.
EMBER_JOINED_NETWORK	0x02	The node is joined to a network.
EMBER_JOINED_NETWORK_NO_PARENT	0x03	The node is an end device joined to a network but its parent is not responding.
EMBER_LEAVING_NETWORK	0x04	The node is in the process of leaving its current network.

Table 34. EmberIncomingMessageType

Structure	Field	Description
EMBER_INCOMING_DATAGRAM	0x00	Datagram.
EMBER_INCOMING_DATAGRAM_REPLY	0x01	Datagram reply.
EMBER_INCOMING_SEQUENCED	0x02	Sequenced message.
EMBER_INCOMING_MULTICAST	0x03	Multicast.
EMBER_INCOMING_SHARED_MULTICAST	0x04	Shared multicast.
EMBER_INCOMING_MULTICAST_LOOPBACK	0x05	Multicast loopback.
333EMBER_INCOMING_UNICAST	0x06	Unicast.
EMBER_INCOMING_BROADCAST	0x07	Broadcast.

Table 35. EmberBindingType

Structure	Field	Description
EMBER_UNUSED_BINDING	0x00	A binding that is currently not in use.
EMBER_UNICAST_BINDING	0x01	A unicast binding whose 64-bit identifier is the destination EUI64.
EMBER_AGGREGATION_BINDING	0x02	A unicast binding whose 64-bit identifier is the aggregator EUI64.
EMBER_MULTICAST_BINDING	0x03	A multicast binding whose 64-bit identifier is the group address. A multicast binding can be used to send messages to the group and to receive messages sent to the group.

Table 36. EmberUnicastOption

Structure	Field	Description
EMBER_UNICAST_OPTION_NONE	0x00	No options.
EMBER_UNICAST_OPTION_APS_INDIRECT	0x04	Reserved.
EMBER_UNICAST_OPTION_HAVE_SOURCE	0x10	Reserved.
EMBER_UNICAST_OPTION_APS_RETRY	0x40	Resend the message using the APS retry mechanism.
EMBER_UNICAST_OPTION_ENABLE_ROUTE DISCOVERY	0x80	Causes a route discovery to be initiated if no route to the destination is known.
EMBER_UNICAST_OPTION_FORCE_ROUTE_DISCOVERY	0x20	Causes a route discovery to be initiated even if one is known.
EMBER_UNICAST_OPTION_POLL_RESPONSE	0x01	Reserved.

Table 37. EmberNetworkScanType

Structure	Field	Description
EMBER_ENERGY_SCAN	0x00	An energy scan scans each channel for its RSSI value.
EMBER_ACTIVE_SCAN	0x01	An active scan scans each channel for available networks.

Table 38. EmberJoinDecision

Structure	Field	Description
EMBER_HAS_KEY	0x00	Allow the node to join. The node has the key.
EMBER_SEND_KEY	0x01	Allow the node to join. Send the key to the node.
EMBER_DENY_JOIN	0x02	Deny join.
EMBER_ASK_TRUST_CENTER	0x03	Ask the trust center.

7.3.4 Configuration frames

Table 39. version

Name: version	**ID: 0x00**
Description: The command allows the Host to specify the desired EZSP version. This document describes version 1 of the protocol. The response provides information about the firmware running on the SN260.	
Command parameters:	
int8u desiredProtocolVersion	The EZSP version the Host wishes to use.
Response parameters:	
int8u protocolVersion	The EZSP version the SN260 is using. If the SN260 does not support the version requested by the Host, it will use the highest version it does support.
int8u stackType	The type of stack running on the SN260. The available EZSP commands and their parameters depend on the stack type. The mesh stack is type 2.
int16u stackVersion	The version number of the stack.

Table 40. getConfigurationValue

Name: getConfigurationValue	**ID: 0x52**
Description: Reads a configuration value from the SN260.	
Command parameters:	
EzspConfigId configId	Identifies which configuration value to read.
Response parameters:	
EzspConfigStatus status	EZSP_CONFIG_SUCCESS if the value was read successfully, EZSP_CONFIG_INVALID_ID if the SN260 does not recognize `configId`.
int16u value	The configuration value.

Table 41. setConfigurationValue

Name: setConfigurationValue	**ID: 0x53**
Description: Writes a configuration value to the SN260. Configuration values can be modified by the Host after the SN260 has reset. After the stack status changes to EMBER_NETWORK_UP, configuration values can no longer be modified and this command will respond with EZSP_CONFIG_INVALID_CALL.	
Command parameters:	
EzspConfigId configId	Identifies which configuration value to change.
int16u value	The new configuration value.
Response parameters:	
EzspConfigStatus status	EZSP_CONFIG_SUCCESS if the configuration value was changed, EZSP_CONFIG_OUT_OF_MEMORY if the new value exceeded the available memory, EZSP_CONFIG_INVALID_VALUE if the new value was out of bounds, EZSP_CONFIG_INVALID_ID if the SN260 does not recognize `configId`, EZSP_CONFIG_INVALID_CALL if configuration values can no longer be modified.

Table 42. addEndpoint

Name: addEndpoint	ID: 0x02
Description: Configures endpoint information on the SN260. The SN260 does not remember these settings after a reset. Endpoints can be added by the Host after the SN260 has reset. After the stack status changes to EMBER_NETWORK_UP, endpoints can no longer be added and this command will respond with EZSP_CONFIG_INVALID_CALL.	
Command parameters:	
int8u endpoint	The application endpoint to be added.
int16u profileId	The endpoint's application profile.
int16u deviceId	The endpoint's device ID within the application profile.
int8u appFlags	The device version and flags indicating description availability.
int8u inputClusterCount	The number of input clusters.
int8u outputClusterCount	The number of output clusters.
int8u[] inputClusterList	Input cluster IDs the endpoint will accept.
int8u[] outputClusterList	Output cluster IDs the endpoint may send.
Response parameters:	
EzspConfigStatus status	EZSP_CONFIG_SUCCESS if the endpoint was added, EZSP_CONFIG_OUT_OF_MEMORY if there is not enough memory available to add the endpoint, EZSP_CONFIG_INVALID_VALUE if the endpoint already exists, EZSP_CONFIG_INVALID_CALL if endpoints can no longer be added.

Table 43. setPolicy

Name: setPolicy	ID: 0x55
Description: Allows the Host to change the policies used by the SN260 to make fast decisions.	
Command parameters:	
EzspPolicyId policyId	Identifies which policy to modify.
EzspDecisionId decisionId	The new decision for the specified policy.
Response parameters:	
EzspConfigStatus status	EZSP_CONFIG_SUCCESS if the policy was changed, EZSP_CONFIG_INVALID_ID if the SN260 does not recognize `policyId`.

Table 44. getPolicy

Name: getPolicy	ID: 0x56
Description: Allows the Host to read the policies used by the SN260 to make fast decisions.	
Command parameters:	
EzspPolicyId policyId	Identifies which policy to read.

Table 44. getPolicy (continued)

Response parameters:	
EzspConfigStatus status	EZSP_CONFIG_SUCCESS if the policy was read successfully, EZSP_CONFIG_INVALID_ID if the SN260 does not recognize policyId.
EzspDecisionId decisionId	The current decision for the specified policy.

7.3.5 Utilities frames

Table 45. nop

Name: nop	**ID: 0x05**
Description: A transaction which does nothing. The Host can use this to set the sleep mode or to check the status of the SN260.	
Command parameters: None	
Response parameters: None	

Table 46. invalidCommand

Name: invalidCommand	**ID: 0x58**
Description: Indicates that the SN260 received a command containing an unsupported frame ID.	
This frame is a response to an invalid command.	
Response parameters: None	

Table 47. callback

Name: callback	**ID: 0x06**
Description: Allows the SN260 to respond with a pending callback.	
Command parameters: None	
The response to this command can be any of the callback responses.	

Table 48. noCallbacks

Name: noCallbacks	**ID: 0x07**
Description: Indicates that there are currently no pending callbacks.	
This frame is a response to the callback command.	
Response parameters: None	

Table 49. reset

Name: reset	**ID: 0x08**
Description: Allows the Host to reset the SN260.	
Command parameters: None	
Response parameters: None	

Table 50. setToken

Name: setToken	**ID: 0x09**
Description: Sets a token (8 bytes of non-volatile storage) in the simulated EEPROM of the SN260.	
Command parameters:	
int8u tokenId	Which token to set (0 to 7).
int8u[8] tokenData	The data to write to the token.
Response parameters:	
EmberStatus status	An EmberStatus value indicating success or the reason for failure.

Table 51. getToken

Name: getToken	**ID: 0x0A**
Description: Retrieves a token (8 bytes of non-volatile storage) from the simulated EEPROM of the SN260.	
Command parameters:	
int8u tokenId	Which token to read (0 to 7).
Response parameters:	
EmberStatus status	An EmberStatus value indicating success or the reason for failure.
int8u[8] tokenData	The contents of the token.

Table 52. getMfgToken

Name: getMfgToken	**ID: 0x0B**
Description: Retrieves a manufacturing token (8 bytes of non-volatile storage) from the Flash Information Area of the SN260.	
Command parameters:	
int8u tokenId	Which manufacturing token to read (0 to 7).
Response parameters:	
EmberStatus status	An EmberStatus value indicating success or the reason for failure.
int8u[8] tokenData	The contents of the manufacturing token.

Table 53. setRam

Name: setRam	**ID: 0x46**
Description: Writes data supplied by the Host to RAM in the SN260. The amount of RAM available for use by the Host must be set using the `setConfigurationValue` command.	
parameters	
int8u startIndex	The location to start writing the data.
int8u dataLength	The length of the `data` parameter in bytes.
int8u[] data	The data to write to RAM.
Response parameters:	
EmberStatus status	An EmberStatus value indicating success or the reason for failure.

Table 54. getRam

Name: getRam	**ID: 0x47**
Description: Reads data from RAM in the SN260 and returns it to the Host.	
Command parameters:	
int8u startIndex	The location to start reading the data.
int8u length	The number of bytes to read.
Response parameters:	
EmberStatus status	An EmberStatus value indicating success or the reason for failure.
int8u dataLength	The length of the `data` parameter in bytes.
int8u[] data	The data read from RAM.

Table 55. getRandomNumber

Name: getRandomNumber	**ID: 0x49**
Description: Returns a random number, generated using noise from the radio.	
Command parameters: None	
Response parameters:	
EmberStatus status	An EmberStatus value indicating success or the reason for failure.
int16u value	If `status` is EMBER_SUCCESS, a random number. Otherwise, zero.

Table 56. getMillisecondTime

Name: getMillisecondTime	**ID: 0x0D**
Description: Returns the current time in milliseconds according to the SN260's internal clock.	
Command parameters: None	
Response parameters:	
int32u time	The current time in milliseconds.

Table 57. setTimer

Name: setTimer	**ID: 0x0E**
Description: Sets a timer on the SN260. There are 2 independent timers available for use by the Host. A timer can be cancelled by setting `time` to 0 or `units` to EMBER_EVENT_INACTIVE.	
Command parameters:	
int8u timerId	Which timer to set (0 or 1).
int16u time	The delay before the `timerHandler` callback will be generated. Note that the timer clock is free running and is not synchronized with this command. This means that the actual delay will be between `time` and (`time - 1`). The maximum delay is 32767.
EmberEventUnits units	The units for `time`.
boolean repeat	If true, a `timerHandler` callback will be generated repeatedly. If false, only a single `timerHandler` callback will be generated.

Table 57. setTimer (continued)

Response parameters	
EmberStatus status	An EmberStatus value indicating success or the reason for failure.

Table 58. getTimer

Name: getTimer	**ID: 0x4E**
Description: Gets information about a timer. The Host can use this command to find out how much longer it will be before a previously set timer will generate a callback.	
Command parameters:	
int8u timerId	Which timer to get information about (0 or 1).
Response parameters:	
int16u time	The delay before the `timerHandler` callback will be generated.
EmberEventUnits units	The units for `time`.
boolean repeat	True if a `timerHandler` callback will be generated repeatedly. False if only a single `timerHandler` callback will be generated.

Table 59. timerHandler

Name: timerHandler	**ID: 0x0F**
Description: A callback from the timer.	
This frame is a response to the `callback` command.	
Response parameters:	
int8u timerId	Which timer generated the callback (0 or 1).

Table 60. serialWrite

Name: serialWrite	**ID: 0x10**
Description: Sends a serial message from the Host to the InSight debug system via the SN260.	
Command parameters:	
int8u messageLength	The length of the `messageContents` parameter in bytes.
int8u[] messageContents	The serial message.
Response parameters:	
EmberStatus status	An EmberStatus value indicating success or the reason for failure.

Table 61. serialRead

Name: serialRead	ID: 0x11
Description: Allows the Host to read a serial message from the InSight debug system via the SN260.	
Command parameters:	
int8u length	The maximum number of bytes to read.
Response parameters:	
int8u messageLength	The length of the `messageContents` parameter in bytes.
int8u[] messageContents	The serial message.

Table 62. debugWrite

Name: debugWrite	ID: 0x12
Description: Sends a debug message from the Host to the InSight debug system via the SN260.	
Command parameters:	
boolean binaryMessage	TRUE if the message should be interpreted as binary data, FALSE if the message should be interpreted as ASCII text.
int8u messageLength	The length of the `messageContents` parameter in bytes.
int8u[] messageContents	The binary message.
Response parameters:	
EmberStatus status	An EmberStatus value indicating success or the reason for failure.

Table 63. debugHandler

Name: debugHandler	ID: 0x13
Description: Delivers a binary message from the InSight debug system to the Host via the SN260.	
This frame is a response to the `callback` command.	
Response parameters:	
int8u messageLength	The length of the `messageContents` parameter in bytes.
int8u[] messageContents	The binary message.

7.3.6 Networking frames

Table 64. setEncryptionKey

Name: setEncryptionKey	ID: 0x14
Description: Sets the encryption key used to encrypt and decrypt radio messages. This function does not work if the stack is already associated with a network.	
Command parameters:	
int8u[16] key	A pointer to a 16-byte encryption key.
int8u keySequenceNumber	The sequence number associated with this key.

Table 64. setEncryptionKey (continued)

Response parameters:	
EmberStatus status	An EmberStatus value indicating success or the reason for failure.

Table 65. setManufacturerCode

Name: setManufacturerCode	**ID: 0x15**
Description: Sets the manufacturer code to the specified value. The manufacturer code is one of the fields of the node descriptor.	
Command parameters:	
int16u code	The manufacturer code for the local node.
Response parameters: None	

Table 66. setPowerDescriptor

Name: setPowerDescriptor	**ID: 0x16**
Description: Sets the power descriptor to the specified value. The power descriptor is a dynamic value, therefore you should call this function whenever the value changes.	
Command parameters:	
int16u descriptor	The new power descriptor for the local node.
Response parameters: None	

Table 67. networkInit

Name: networkInit	**ID: 0x17**
Description: Resume network operation after a reboot. The node retains its original type. This should be called on startup whether or not the node was previously part of a network. EMBER_NOT_JOINED is returned if the node is not part of a network.	
Command parameters: None	
Response parameters:	
EmberStatus status	An EmberStatus value that indicates one of the following: successful initialization, EMBER_NOT_JOINED if the node is not part of a network, or the reason for failure.

Table 68. networkState

Name: networkState	**ID: 0x18**
Description: Returns a value indicating whether the node is joining, joined to, or leaving a network.	
Command parameters: None	
Response parameters:	
EmberNetworkStatus status	An EmberNetworkStatus value indicating the current join status.

Table 69. stackStatusHandler

Name: stackStatusHandler	**ID: 0x19**
Description: A callback invoked when the status of the stack changes. If the status parameter equals EMBER_NETWORK_UP, then the `getNetworkParameters` command can be called to obtain the new network parameters. If any of the parameters are being stored in nonvolatile memory by the Host, the stored values should be updated.	
This frame is a response to the `callback` command.	
Response parameters:	
EmberStatus status	Stack status. One of the following: EMBER_NETWORK_UP, EMBER_NETWORK_DOWN, EMBER_JOIN_FAILED, EMBER_MOVE_FAILED

Table 70. startScan

Name: startScan	**ID: 0x1A**
Description: This function will start a scan.	
Command parameters:	
EmberNetworkScanType scanType	Indicates the type of scan to be performed. Possible values: EMBER_ENERGY_SCAN, EMBER_ACTIVE_SCAN.
int32u channelMask	Bits set as 1 indicate that this particular channel should be scanned. Bits set to 0 indicate that this particular channel should not be scanned. For example, a channelMask value of 0x00000001 would indicate that only channel 0 should be scanned. Valid channels range from 11 to 26 inclusive. This translates to a channel mask value of 0x07FFF800.
int8u duration	Sets the exponent of the number of scan periods, where a scan period is 960 symbols. The scan will occur for ((2^duration) + 1) scan periods.
Response parameters:	
EmberStatus status	EMBER_SUCCESS signals that the scan successfully started. Possible error responses and their meanings: EMBER_MAC_SCANNING, we are already scanning; EMBER_MAC_JOINED_NETWORK, we are currently joined to a network and can not begin a scan; EMBER_MAC_BAD_SCAN_DURATION, we have set a duration value that is not 0..14 inclusive; EMBER_MAC_INCORRECT_SCAN_TYPE, we have requested an undefined scanning type; EMBER_MAC_INVALID_CHANNEL_MASK, our channel mask did not specify any valid channels.

Table 71. energyScanResultHandler

Name: energyScanResultHandler	**ID: 0x48**
Description: Reports the result of an energy scan for a single channel. The scan is not complete until the `scanCompleteHandler` callback is called.	
This frame is a response to the `callback` command.	
Response parameters:	
int8u channel	The 802.15.4 channel number that was scanned.
int8u maxRssiValue	The maximum RSSI value found on the channel.

Table 72. networkFoundHandler

Name: networkFoundHandler	**ID: 0x1B**
Description: Reports that a network was found, and gives the network parameters useful for deciding which network to join.	
This frame is a response to the `callback` command.	
Response parameters:	
int8u channel	The 802.15.4 channel number on which the current network was found.
int16u panId	The PAN ID of the current network.
boolean expectingJoin	Whether the node that generated this beacon is allowing additional children to join to its network.
int8u stackProfile	The ZigBee profile number of the current network.

Table 73. scanCompleteHandler

Name: scanCompleteHandler	**ID: 0x1C**
Description: Returns the status of the current scan. EMBER_SUCCESS signals that the scan has completed. Other error conditions signify a failure to scan on the channel specified.	
This frame is a response to the `callback` command.	
Response parameters:	
int8u channel	The channel on which the current error occurred. Undefined for the case of EMBER_SUCCESS.
EmberStatus status	The error condition that occurred on the current channel. Value will be EMBER_SUCCESS when the scan has completed.

Table 74. stopScan

Name: stopScan	**ID: 0x1D**
Description: Terminates a scan in progress.	
Command parameters: None	
Response parameters:	
EmberStatus status	An EmberStatus value indicating success or the reason for failure.

Table 75. formNetwork

Name: formNetwork	**ID: 0x1E**
Description: Forms a new network by becoming the coordinator.	
Command parameters:	
EmberNetworkParameters	Specification of the new network.
Response parameters:	
EmberStatus status	An EmberStatus value indicating success or the reason for failure.

Table 76. joinNetwork

Name: joinNetwork	**ID: 0x1F**
Description: Causes the stack to associate with the network using the specified network parameters. It can take several seconds for the stack to associate with the local network. Do not send messages until the `stackStatusHandler` callback informs you that the stack is up.	
Command parameters:	
EmberNodeType nodeType	Specification of the role that this node will have in the network. This role must not be EMBER_COORDINATOR. To be a coordinator, use the `formNetwork` command.
EmberNetworkParameters	Specification of the network with which the node should associate.
boolean joinSecurely	If true, the node uses the current key to secure messages during the joining process. The proper value for secured networks depends upon their configuration. Some networks use unsecured joining and distribute the key from the coordinator. Other networks require secure joining and accept only nodes that know the correct key. This value has no effect if the security level is 0.
Response parameters:	
EmberStatus status	An EmberStatus value indicating success or the reason for failure.

Table 77. scanAndFormNetwork

Name: scanAndFormNetwork	**ID: 0x4F**
Description: Scan for an available channel and PAN ID then form a network. This performs the following actions: 1. Performs an energy scan on the indicated channels and randomly chooses one from amongst those with the least average energy. 2. Randomly picks a PAN ID that does not appear during an active scan on the chosen channel. 3. Forms a network using the chosen channel and PAN ID. If any errors occur the status code is passed to the `scanErrorHandler` callback and no network is formed. Success is indicated when the `stackStatusHandler` callback is invoked with the EMBER_NETWORK_UP status value.	
Command parameters:	
int32u channelMask	Bits set as 1 indicate that this particular channel should be scanned. Bits set to 0 indicate that this particular channel should not be scanned. For example, a channelMask value of 0x00000001 would indicate that only channel 0 should be scanned. Valid channels range from 11 to 26 inclusive. This translates to a channel mask value of 0x07FFF800.
int8s radioTxPower	A power setting, in dBm.
Response parameters: None	

Table 78. scanAndJoinNetwork

Name: scanAndJoinNetwork	**ID: 0x50**
Description: Scan and join a network. This performs the following actions: 1. Does an active scan to find a network that uses our stack profile and currently allows new nodes to join. 2. Joins the chosen network. If any errors occur the status code is passed to the `scanErrorHandler` callback and no network is joined. Success is indicated when the `stackStatusHandler` callback is invoked with the EMBER_NETWORK_UP status value.	

Table 78. scanAndJoinNetwork (continued)

Command parameters:	
EmberNodeType nodeType	Specification of the role that this node will have in the network. This role must not be EMBER_COORDINATOR. To be a coordinator, use the `scanAndformNetwork` command.
int32u channelMask	Bits set as 1 indicate that this particular channel should be scanned. Bits set to 0 indicate that this particular channel should not be scanned. For example, a channelMask value of 0x00000001 would indicate that only channel 0 should be scanned. Valid channels range from 11 to 26 inclusive. This translates to a channel mask value of 0x07FFF800.
int8s radioTxPower	A power setting, in dBm.
boolean joinSecurely	If true, the node uses the current key to secure messages during the joining process. The proper value for secured networks depends upon their configuration. Some networks use unsecured joining and distribute the key from the coordinator. Other networks require secure joining and accept only nodes that know the correct key. This value has no effect if the security level is 0.
Response parameters: None	

Table 79. scanErrorHandler

Name: scanErrorHandler	**ID: 0x51**
Description: This callback is invoked if an error occurs while attempting to `scanAndFormNetwork` or `scanAndJoinNetwork`.	
This frame is a response to the `callback` command.	
Response parameters:	
EmberStatus status	An EmberStatus value indicating the reason for the `scanAndFormNetwork` or `scanAndJoinNetwork` failure.

Table 80. leaveNetwork

Name: leaveNetwork	**ID: 0x20**
Description: Causes the stack to leave the current network. This generates a `stackStatusHandler` callback to indicate that the network is down. The radio will not be used until after sending a `formNetwork` or `joinNetwork` command.	
Command parameters: None	
Response parameters:	
EmberStatus status	An EmberStatus value indicating success or the reason for failure.

Table 81. mobileNodeHasMoved

Name: mobileNodeHasMoved	**ID: 0x21**
Description: Informs the stack that contact with the network has been lost. Only devices that are joined to a network with a node type of EMBER_MOBILE_END_DEVICE may call this function. This generates a `stackStatusHandler` callback to indicate that the network is down. The stack will try to re-establish contact with the network. A second `stackStatusHandler` callback indicates either the success or the failure of the attempt.	
Command parameters: None	

Table 81. mobileNodeHasMoved (continued)

Response parameters:	
EmberStatus status	An EmberStatus value indicating success or the reason for failure.

Table 82. permitJoining

Name: permitJoining	**ID: 0x22**
Description: Tells the stack to allow other nodes to join the network with this node as their parent. Joining is initially disabled by default.	
Command parameters:	
int8u duration	A value of 0x00 disables joining. A value of 0xFF enables joining. Any other value enables joining for that number of seconds.
Response parameters:	
EmberStatus status	An EmberStatus value indicating success or the reason for failure.

Table 83. childJoinHandler

Name: childJoinHandler	**ID: 0x23**
Description: Indicates that a child has joined or left.	
This frame is a response to the `callback` command.	
Response parameters:	
int8u index	The index of the child of interest.
boolean joining	True if the child is joining. False the child is leaving.
EmberNodeId childId	The node ID of the child.
EmberEUI64 childEui64	The EUI64 of the child.
EmberNodeType childType	The node type of the child.

Table 84. trustCenterJoinHandler

Name: trustCenterJoinHandler	**ID: 0x24**
Description: The SN260 used the trust center behavior policy to decide whether to allow a new node to join the network. The Host cannot change the current decision, but it can change the policy for future decisions using the `setPolicy` command.	
This frame is a response to the `callback` command.	
Response parameters:	
EmberEUI64 newNode	The EUI64 of the node that wished to join.
boolean securedJoin	True if the node was joining securely using the network security key.
EmberJoinDecision policyDecision	An EmberJoinDecision reflecting the decision made.

Table 85. sendDiscoveryInformationToParent

Name: **sendDiscoveryInformationToParent**	**ID: 0x25**
Description: Initiates the upload of discovery information to the parent of this node. Only devices that are joined to a network with a node type of EMBER_SLEEPY_END_DEVICE may call this function. The parent stores the information in its discovery cache. The information is sent using ZDO messages with cluster IDs NODE_DESCRIPTOR_RESPONSE, POWER_DESCRIPTOR_RESPONSE and SIMPLE_DESCRIPTOR_RESPONSE.	
Command parameters: None	
Response parameters:	
EmberStatus status	An EmberStatus value indicating success or the reason for failure.

Table 86. getEui64

Name: getEui64	**ID: 0x26**
Description: Returns the EUI64 ID of the local node.	
Command parameters: None	
Response parameters:	
EmberEUI64 eui64	The 64-bit ID.

Table 87. getNodeId

Name: getNodeId	**ID: 0x27**
Description: Returns the 16-bit node ID of the local node.	
Command parameters: None	
Response parameters:	
EmberNodeId nodeId	The 16-bit ID.

Table 88. getNetworkParameters

Name: getNetworkParameters	**ID: 0x28**
Description: Returns the current network parameters.	
Command parameters: None	
Response parameters:	
EmberStatus status	An EmberStatus value indicating success or the reason for failure.
EmberNodeType nodeType	An EmberNodeType value indicating the current node type.
EmberNetworkParameters	The current network parameters.

Table 89. getParentChildParameters

Name: getParentChildParameters	**ID: 0x29**
Description: Returns information about the children of the local node and the parent of the local node.	
Command parameters: None	

Table 89. getParentChildParameters (continued)

Response parameters:	
int8u childCount	The number of children the node currently has.
EmberEUI64 parentEui64	The parent's EUI64. The value is undefined for nodes without parents (coordinators and nodes that are not joined to a network).
EmberNodeId parentNodeId	The parent's node ID. The value is undefined for nodes without parents (coordinators and nodes that are not joined to a network).

Table 90. getChildData

Name: getChildData	**ID: 0x4A**
Description: Returns information about a child of the local node.	
Command parameters:	
int8u index	The index of the child of interest in the child table. Possible indexes range from zero to EMBER_CHILD_TABLE_SIZE.
Response parameters:	
EmberStatus status	EMBER_SUCCESS if there is a child at `index`. EMBER_NOT_JOINED if there is no child at `index`.
EmberNodeId childId	The node ID of the child.
EmberEUI64 childEui64	The EUI64 of the child.
EmberNodeType childType	The EmberNodeType value for the child.

7.3.7 Binding frames

Table 91. clearBindingTable

Name: clearBindingTable	**ID: 0x2A**
Description: Deletes all binding table entries.	
Command parameters: None	
Response parameters:	
EmberStatus status	An EmberStatus value indicating success or the reason for failure.

Table 92. setBinding

Name: setBinding	**ID: 0x2B**
Description: Sets an entry in the binding table.	
Command parameters:	
int8u index	The index of a binding table entry.
EmberBindingTableEntry value	The contents of the binding entry.
Response parameters:	
EmberStatus status	An EmberStatus value indicating success or the reason for failure.

Table 93. getBinding

Name: getBinding	**ID: 0x2C**
Description: Gets an entry from the binding table.	
Command parameters:	
int8u index	The index of a binding table entry.
Response parameters:	
EmberStatus status	An EmberStatus value indicating success or the reason for failure.
EmberBindingTableEntry value	The contents of the binding entry.

Table 94. deleteBinding

Name: deleteBinding	**ID: 0x2D**
Description: Deletes a binding table entry.	
Command parameters:	
int8u index	The index of a binding table entry.
Response parameters:	
EmberStatus status	An EmberStatus value indicating success or the reason for failure.

Table 95. bindingIsActive

Name: bindingIsActive	**ID: 0x2E**
Description: Indicates whether a binding table entry is active—that is, whether a connection to it is open or any messages are en route from it. Note that this command does not indicate whether a binding is clear. To determine whether a binding is clear, check whether the type field of the EmberBindingTableEntry has the value EMBER_UNUSED_BINDING.	
Command parameters:	
int8u index	The index of a binding table entry.
Response parameters:	
boolean active	True if the binding table entry is active. False if the binding table entry is not active.

Table 96. getBindingDestinationNodeId

Name: getBindingDestinationNodeId	**ID: 0x2F**
Description: Returns the node ID for the binding's destination, if the ID is known. If a message is sent using the binding and the destination's ID is not known, the stack will discover the ID by broadcasting a ZDO address request. The application can avoid the need for this discovery by using `setBindingDestinationNodeId` when it knows the correct ID via some other means. The destination's node ID is forgotten when the binding is changed, when the local node reboots or, much more rarely, when the destination node changes its ID in response to an ID conflict.	
Command parameters:	
int8u index	The index of a binding table entry.

Table 96. getBindingDestinationNodeId (continued)

Response parameters:	
EmberNodeId nodeId	The short ID of the destination node or EMBER_NULL_NODE_ID if no destination is known.

Table 97. setBindingDestinationNodeId

Name: setBindingDestinationNodeId	**ID: 0x30**
Description: Set the node ID for the binding's destination. See `getBindingDestinationNodeId` for a description.	
Command parameters:	
int8u index	The index of a binding table entry.
EmberNodeId nodeId	The short ID of the destination node.
Response parameters: None	

Table 98. remoteSetBindingHandler

Name: remoteSetBindingHandler	**ID: 0x31**
Description: The SN260 used the external binding modification policy to decide how to handle a remote set binding request. The Host cannot change the current decision, but it can change the policy for future decisions using the `setPolicy` command.	
This frame is a response to the `callback` command.	
Response parameters:	
EmberBindingTableEntry entry	The requested binding.
int8u index	The index at which the binding was added.
EmberStatus policyDecision	EMBER_SUCCESS if the binding was added to the table and any other status if not.

Table 99. remoteDeleteBindingHandler

Name: remoteDeleteBindingHandler	**ID: 0x32**
Description: The SN260 used the external binding modification policy to decide how to handle a remote delete binding request. The Host cannot change the current decision, but it can change the policy for future decisions using the `setPolicy` command.	
This frame is a response to the `callback` command.	
Response parameters:	
int8u index	The index of the binding whose deletion was requested.
EmberStatus policyDecision	EMBER_SUCCESS if the binding was removed from the table and any other status if not.

7.3.8 Messaging frames

Table 100. maximumPayloadLength

Name: maximumPayloadLength	**ID: 0x33**
Command parameters: None	
Response parameters:	
int8u apsLength	The maximum APS payload length.
int8u transportLength	The maximum transport payload length.

Table 101. sendUnicast

Name: sendUnicast	**ID: 0x34**
Description: Sends a unicast message as per the ZigBee specification. The message will arrive at its destination only if there is a known route to the destination node. Setting the ENABLE_ROUTE_DISCOVERY option will cause a route to be discovered if none is known. Setting the FORCE_ROUTE_DISCOVERY option will force route discovery. Routes to end-device children of the local node are always known. Setting the APS_RETRY option will cause the message to be retransmitted until either a matching acknowledgement is received or three transmissions have been made. The ZigBee APS retry mechanism does not use sequence numbers. If multiple messages are sent to the same destination at the same time any acknowledgement from that node will stop transmission of all outstanding messages. **Note:** Using the FORCE_ROUTE_DISCOVERY option will cause the first transmission to be consumed by a route request as part of discovery, so the application payload of this packet will not reach its destination on the first attempt. If you want the packet to reach its destination, the APS_RETRY option must be set so that another attempt is made to transmit the message with its application payload after the route has been constructed.	
Command parameters:	
EmberNodeId destination	The node ID to which the message will be sent.
EmberApsFrame apsFrame	The APS frame for the message.
int8u messageTag	A value chosen by the Host. This value is used in the `emberUnicastSent` response to refer to this message.
int8u messageLength	The length of the `messageContents` parameter in bytes.
int8u[] messageContents	The unicast message.
Response parameters:	
EmberStatus status	An EmberStatus value indicating success or the reason for failure.

Table 102. unicastSent

Name: unicastSent	**ID: 0x35**
Description: A callback indicating the stack has completed sending a non-transport unicast message. Except for the status value, the parameters are identical to those of the `sendUnicast` command used to send the message.	
This frame is a response to the `callback` command.	
Response parameters:	
EmberNodeId destination	The node ID to which the message was be sent.
EmberApsFrame apsFrame	The APS frame for the message.

Table 102. unicastSent (continued)

int8u messageTag	The value supplied by the Host in the `emberSendUnicast` command.
EmberStatus status	An EmberStatus value indicating success or the reason for failure.
int8u messageLength	The length of the `messageContents` parameter in bytes.
int8u[] messageContents	The unicast message supplied by the Host. The message contents are only included here if the decision for the `messageContentsInCallback` policy is `messageTagAndContentsInCallback`.

Table 103. sendBroadcast

Name: sendBroadcast	**ID: 0x36**
Description: Sends a broadcast message as per the ZigBee specification.	
Command parameters:	
EmberApsFrame apsFrame	The APS frame for the message.
int8u radius	The message will be delivered to all nodes within `radius` hops of the sender. A radius of zero is converted to EMBER_MAX_HOPS.
int8u messageTag	Reserved for future use. This value is ignored by the SN260.
int8u messageLength	The length of the `messageContents` parameter in bytes.
int8u[] messageContents	The broadcast message.
Response parameters:	
EmberStatus status	An EmberStatus value indicating success or the reason for failure.

Table 104. sendDatagram

Name: sendDatagram	**ID: 0x37**
Description: Sends a datagram to the node and endpoint specified in a binding table entry. The status of the delivery will be reported by a `messageSent` callback.	
Command parameters:	
int8u bindingTableIndex	The index of the binding table entry.
int8u clusterId	The cluster ID to use.
int8u messageTag	A value chosen by the Host. This value is used in the `emberCancelMessage` command and the `emberMessageSent` response to refer to this message.
int8u messageLength	The length of the `messageContents` parameter in bytes.
int8u[] messageContents	The datagram message.

Table 104. sendDatagram (continued)

Response parameters:	
EmberStatus status	An EmberStatus value. For any result other than EMBER_SUCCESS, the message will not be sent. EMBER_SUCCESS: The message has been submitted for transmission. EMBER_INVALID_BINDING_INDEX: The bindingTableIndex refers to a non-unicast binding. EMBER_NETWORK_DOWN: The node is not part of a network. EMBER_MESSAGE_TOO_LONG: The message is too large to fit in a MAC layer frame. EMBER_MAX_MESSAGE_LIMIT_REACHED: The EMBER_TRANSPORT_PACKET_COUNT limit has been reached.

Table 105. sendMulticast

Name: sendMulticast	**ID: 0x38**
Description: Sends a multicast message to all endpoints that share a specific multicast ID and are within a specified number of hops of the sender.	
Command parameters:	
int8u bindingTableIndex	The index of the binding table entry specifying the multicast group.
int8u clusterId	The cluster ID to use.
int8u messageTag	Reserved for future use. This value is ignored by the SN260.
int8u hops	The message will be delivered to all nodes within this number of hops of the sender. A value of zero is converted to EMBER_MAX_HOPS.
int8u messageLength	The length of the `messageContents` parameter in bytes.
int8u[] messageContents	The multicast message.
Response parameters:	
EmberStatus status	An EmberStatus value. For any result other than EMBER_SUCCESS, the message will not be sent. EMBER_SUCCESS: The message has been submitted for transmission. EMBER_INVALID_BINDING_INDEX: The bindingTableIndex refers to a non-multicast binding. EMBER_NETWORK_DOWN: The node is not part of a network. EMBER_MESSAGE_TOO_LONG: The message is too large to fit in a MAC layer frame. EMBER_NO_BUFFERS: The free packet buffer pool is empty. EMBER_NETWORK_BUSY: Insufficient resources available in Network or MAC layers to send message.

Table 106. sendReply

Name: sendReply	**ID: 0x39**
Description: Sends a reply to a received datagram message. The `incomingMessageHandler` callback for the datagram being replied to supplies the values for all the parameters except the reply itself.	
Command parameters:	
EmberNodeId sender	Value supplied by incoming datagram.

Table 106. sendReply (continued)

EmberApsFrame apsFrame	Value supplied by incoming datagram.
int8u datagramReplyTag	Value supplied by incoming datagram.
int8u messageLength	The length of the `messageContents` parameter in bytes.
int8u[] messageContents	The reply message.
Response parameters:	
EmberStatus status	An EmberStatus value indicating success or the reason for failure.

Table 107. openConnection

Name: openConnection	**ID: 0x3A**
Description: Opens a sequenced connection to a node.	
Command parameters:	
int8u bindingTableIndex	The index of the binding table entry to which a connection will be opened.
Response parameters:	
EmberStatus status	An EmberStatus value indicating success or the reason for failure.

Table 108. connectionStatus

Name: connectionStatus	**ID: 0x3B**
Description: Returns the connection status of a binding table entry.	
Command parameters:	
int8u bindingTableIndex	The index of the binding table entry whose status is being queried.
Response parameters:	
EmberStatus status	An EmberStatus value: EMBER_CONNECTION_CLOSED: The connection is closed. EMBER_CONNECTION_NOT_YET_OPEN: The connection is in the process of being established. EMBER_CONNECTION_OPEN: The connection is currently established. EMBER_CONNECTION_CLOSING: The connection is in the process of being closed.

Table 109. connectionStatusHandler

Name: connectionStatusHandler	**ID: 0x3C**
Description: A callback indicating the status of a connection has changed.	
This frame is a response to the `callback` command.	

Table 109. connectionStatusHandler (continued)

Response parameters:	
int8u bindingTableIndex	The index of the binding table entry whose connection status has changed.
EmberStatus status	An EmberStatus value: EMBER_CONNECTION_OPEN: A sequenced connection has successfully been established for the binding. It may have been initiated locally or remotely. EMBER_CONNECTION_CLOSING: The sequenced connection for the binding is being closed gracefully. The close may have been initiated locally or remotely. As soon as the disposition of all in-flight messages has been resolved the connection will be completely closed (and the EMBER_CONNECTION_CLOSED status will be reported). EMBER_CONNECTION_CLOSED: The sequenced connection has been successfully closed. The disposition of every message sent over the connection has already been reported (via the various callbacks). There will be no further message related callbacks. EMBER_CONNECTION_FAILED: The sequenced connection has been closed unexpectedly. If there were messages in-flight their disposition will never be known or reported via callbacks. This error may be reported during the opening of a connection, while a connection is established or during the closing of a connection. EMBER_INCOMING_SEQUENCED_MESSAGES_LOST: One or more sequenced messages have not been received on the connection and it has been determined they will never be received.

Table 110. sendSequenced

Name: sendSequenced	**ID: 0x3D**
Description: Sends a sequenced message over the connection associated with a specified binding table entry.	
command parameters:	
int8u bindingTableIndex	The index of the binding table entry specifying the message destination.
int8u clusterId	The cluster ID to use.
int8u messageTag	A value chosen by the Host. This value is used in the `emberCancelMessage` command and the `emberMessageSent` response to refer to this message.
int8u messageLength	The length of the `messageContents` parameter in bytes.
int8u[] messageContents	The sequenced message.

Table 110. sendSequenced (continued)

Response parameters:	
EmberStatus status	An EmberStatus value. For any result other than EMBER_SUCCESS, the message will not be sent. EMBER_SUCCESS: The message has been submitted for transmission. EMBER_CONNECTION_CLOSED: The connection associated with bindingTableIndex is either closed or in the process of closing. EMBER_INVALID_BINDING_INDEX: The bindingTableIndex refers to a non-unicast binding. EMBER_NETWORK_DOWN: The node is not part of a network. EMBER_MESSAGE_TOO_LONG: The message is too large to fit in a MAC layer frame. EMBER_MAX_MESSAGE_LIMIT_REACHED: Either the EMBER_TRANSPORT_PACKET_COUNT limit has been reached or the transmit window is full (i.e. there are already 8 sequenced messages in flight on the connection).

Table 111. closeConnection

Name: closeConnection	**ID: 0x3E**
Description: Closes a connection. Any sequenced messages previously sent on the connection will be delivered before the connection is closed. Similarly, all messages sent by the remote node before the connection close is initiated will be received before the connection closes locally.	
Command parameters:	
int8u bindingTableIndex	The index of the binding table entry whose connection is to be closed.
Response parameters:	
EmberStatus status	An EmberStatus value indicating success or the reason for failure.

Table 112. messageSent

Name: messageSent	**ID: 0x3F**
Description: A callback indicating the stack has completed sending a datagram or sequenced message.	
This frame is a response to the `callback` command.	
Response parameters:	
int8u bindingTableIndex	The index of the binding table entry to which the message was sent.
int8u clusterId	The cluster ID that was used.
int8u messageTag	The value supplied by the Host in the `emberSendDatagram` or `emberSendSequenced` command.
EmberStatus status	An EmberStatus value of EMBER_SUCCESS if an ACK was received from the destination or EMBER_DELIVERY_FAILED if no ACK was received.
int8u messageLength	The length of the `messageContents` parameter in bytes.
int8u[] messageContents	The unicast message supplied by the Host. The message contents are only included here if the decision for the `messageContentsInCallback` policy is `messageTagAndContentsInCallback`.

Table 113. cancelMessage

Name: cancelMessage	**ID: 0x40**
Description: Cancels an outgoing message.	
Command parameters:	
int8u messageTag	The value supplied by the Host in the `emberSendDatagram` or `emberSendSequenced` command.
Response parameters:	
EmberStatus status	Always returns EMBER_SUCCESS.

Table 114. createAggregationRoutes

Name: createAggregationRoutes	**ID: 0x41**
Description: Sends a route request that creates routes from every node in the network back to this node. This function should be called by the application if it wishes to aggregate data from many nodes. The data sources will not have to discover routes individually. Additionally, incoming data will set up temporary reverse routes that allow acknowledgement messages to return without a route discovery. The temporary routes expire and become reusable after a single use, or 10-20 seconds.	
Command parameters: None	
Response parameters:	
EmberStatus status	EMBER_SUCCESS if the route request was successfully submitted to the transmit queue, and EMBER_ERR_FATAL otherwise.

Table 115. pollForData

Name: pollForData	**ID: 0x42**
Description: Periodically request any pending data from our parent. Setting `interval` to 0 or `units` to EMBER_EVENT_INACTIVE will generate a single poll.	
Command parameters:	
int16u interval	The time between polls. Note that the timer clock is free running and is not synchronized with this command. This means that the time will be between `interval` and (`interval - 1`). The maximum interval is 32767.
EmberEventUnits units	The units for `interval`.
int8u failureLimit	The number of poll failures that will be tolerated before a `pollCompleteHandler` callback is generated. A value of zero will result in a callback for every poll. Any status value apart from EMBER_SUCCESS and EMBER_MAC_NO_DATA is counted as a failure.
Response parameters:	
EmberStatus status	The result of sending the first poll.

Table 116. pollCompleteHandler

Name: pollCompleteHandler	**ID: 0x43**
Description: Indicates the result of a data poll to the parent of the local node.	
This frame is a response to the `callback` command.	

Table 116. pollCompleteHandler (continued)

Response parameters:	
EmberStatus status	An EmberStatus value: EMBER_SUCCESS: Data was received in response to the poll. EMBER_MAC_NO_DATA: No data was pending. EMBER_DELIVERY_FAILED: The poll message could not be sent. EMBER_MAC_NO_ACK_RECEIVED: The poll message was sent but not acknowledged by the parent.

Table 117. pollHandler

Name: pollHandler	**ID: 0x44**
Description: Indicates that the local node received a data poll from a child.	
This frame is a response to the `callback` command.	
Response parameters:	
EmberNodeId childId	The node ID of the child that is requesting data.

Table 118. incomingMessageHandler

Name: incomingMessageHandler	**ID: 0x45**
Description: A callback indicating a message has been received.	
This frame is a response to the `callback` command.	
Response parameters:	
EmberIncomingMessageType type	The type of the incoming message. One of the following: EMBER_INCOMING_DATAGRAM, EMBER_INCOMING_DATAGRAM_REPLY, EMBER_INCOMING_SEQUENCED, EMBER_INCOMING_MULTICAST, EMBER_INCOMING_SHARED_MULTICAST, EMBER_INCOMING_MULTICAST_LOOPBACK, EMBER_INCOMING_UNICAST, EMBER_INCOMING_BROADCAST
EmberApsFrame apsFrame	The APS frame from the incoming message.
int8u lastHopLqi	The link quality from the node that last relayed the message.
int8s lastHopRssi	The energy level (in units of dBm) observed during the reception.
EmberNodeId sender	The sender of the message.
int8u bindingIndex	The index of a binding that matches the message or 0xFF if there is no matching binding.
int8u datagramReplyTag	If the incoming message is a datagram and the Host wishes to send a reply, this value must be supplied to the `emberSendReply` command.
int8u messageLength	The length of the `messageContents` parameter in bytes.
int8u[] messageContents	The incoming message.

7.3.9 Alphabetical list of frames

Table 119. Alphabetical list of frames

Frame Name	ID
addEndpoint	0x02
bindingIsActive	0x2E
callback	0x06
cancelMessage	0x40
childJoinHandler	0x23
clearBindingTable	0x2A
closeConnection	0x3E
connectionStatus	0x3B
connectionStatusHandler	0x3C
createAggregationRoutes	0x41
debugHandler	0x13
debugWrite	0x12
deleteBinding	0x2D
energyScanResultHandler	0x48
formNetwork	0x1E
getBinding	0x2C
getBindingDestinationNodeId	0x2F
getChildData	0x4A
getConfigurationValue	0x52
getEui64	0x26
getMfgToken	0x0B
getMillisecondTime	0x0D
getNetworkParameters	0x28
getNodeId	0x27
getParentChildParameters	0x29
getPolicy	0x56
getRam	0x47
getRandomNumber	0x49
getTimer	0x4E
getToken	0x0A
incomingMessageHandler	0x45
invalidCommand	0x58
joinNetwork	0x1F

Table 119. Alphabetical list of frames (continued)

Frame Name	ID
leaveNetwork	0x20
maximumPayloadLength	0x33
messageSent	0x3F
mobileNodeHasMoved	0x21
networkFoundHandler	0x1B
networkInit	0x17
networkState	0x18
noCallbacks	0x07
nop	0x05
openConnection	0x3A
permitJoining	0x22
pollCompleteHandler	0x43
pollForData	0x42
pollHandler	0x44
remoteDeleteBindingHandler	0x32
remoteSetBindingHandler	0x31
reset	0x08
scanAndFormNetwork	0x4F
scanAndJoinNetwork	0x50
scanCompleteHandler	0x1C
scanErrorHandler	0x51
sendBroadcast	0x36
sendDatagram	0x37
sendDiscoveryInformationToParent	0x25
sendMulticast	0x38
sendReply	0x39
sendSequenced	0x3D
sendUnicast	0x34
serialRead	0x11
serialWrite	0x10
setBinding	0x2B
setBindingDestinationNodeId	0x30
setConfigurationValue	0x53
setEncryptionKey	0x14
setManufacturerCode	0x15

Table 119. Alphabetical list of frames (continued)

Frame Name	ID
setPolicy	0x55
setPowerDescriptor	0x16
setRam	0x46
setTimer	0x0E
setToken	0x09
stackStatusHandler	0x19
startScan	0x1A
stopScan	0x1D
timerHandler	0x0F
trustCenterJoinHandler	0x24
unicastSent	0x35
version	0x00

7.4 Sample transactions

This section provides illustrations of the following sample transactions:

- Joining
- Binding
- Sending
- Receiving

7.4.1 Joining

```
1)  frame control        = 0x00 (command frame, don't sleep)
    joinNetwork command = 0x1F
    nodeType             = 0x02 (EMBER_ROUTER)
    panId                = 0x1234
    radioTxPower         = 0xFF (-1)
    radioChannel         = 0x0B (11)
    useKey               = 0x00 (FALSE)

    HOST -> SN260: | 00 | 1F | 02 | 34 | 12 | FF | 0B | 00 |

    frame control         = 0x80 (response frame, no overflow, not
                                 truncated)
    joinNetwork response = 0x1F
    status                = 0x00 (EMBER_SUCCESS)

    SN260 -> HOST: | 80 | 1F | 00 |


2)  Host waits for callback signal while SN260 tries to join the
network.
```

```
3)  frame control    = 0x00 (command frame, don't sleep)
    callback command = 0x06

    HOST -> SN260: | 00 | 06 |

    frame control                = 0x80 (response frame, no overflow,
                                         not truncated)
    stackStatusHandler response = 0x19
    status                       = 0x90 (EMBER_NETWORK_UP)

    SN260 -> HOST: | 80 | 19 | 90 |
```

7.4.2 Binding

```
1)  frame control      = 0x00 (command frame, don't sleep)
    setBinding command = 0x2B
    index              = 0x00
    type               = 0x01 (EMBER_UNICAST_BINDING)
    local              = 0x11
    remote             = 0x12
    clusterId          = 0x55
    identifier         = 0x1122334455667788

    HOST -> SN260: | 00 | 2B | 00 | 01 | 11 | 12 | 55 | 88 | 77 | 66
| 55
                   | 44 | 33 | 22 | 11 |

    frame control       = 0x80 (response frame, no overflow, not
                                truncated)
    setBinding response = 0x2B
    status              = 0x00 (EMBER_SUCCESS)

    SN260 -> HOST: | 80 | 2B | 00 |
```

7.4.3 Sending

```
1)  frame control        = 0x00 (command frame, don't sleep)
    sendDatagram command = 0x37
    bindingTableIndex    = 0x00
    clusterId            = 0x55
    messageTag           = 0x01
    messageLength        = 0x03
    messageContents      = 0xE1, 0xE2, 0xE3

    HOST -> SN260: | 00 | 37 | 00 | 55 | 01 | 03 | E1 | E2 | E3 |

    frame control         = 0x80 (response frame, no overflow, not
                                  truncated)
    sendDatagram response = 0x37
    status                = 0x00 (EMBER_SUCCESS)

    SN260 -> HOST: | 80 | 37 | 00 |
```

2) Host waits for callback signal while SN260 tries to send the message.

```
3)  frame control    = 0x00 (command frame, don't sleep)
    callback command = 0x06

    HOST -> SN260: | 00 | 06 |

    frame control          = 0x80 (response frame, no overflow, not
truncated)
    messageSent response = 0x3F
    bindingTableIndex      = 0x00
    clusterId              = 0x55
    messageTag             = 0x01
    status                 = 0x00 (EMBER_SUCCESS)

    SN260 -> HOST: | 80 | 3F | 00 | 55 | 01 | 00 |
```

7.4.4 Receiving

1) Host waits for callback signal after a message is received by the SN260.

```
2)  frame control    = 0x00 (command frame, don't sleep)
    callback command = 0x06

    HOST -> SN260: | 00 | 06 |

    frame control                     = 0x80 (response frame, no
                                       overflow, not truncated)
    incomingMessageHandler response = 0x45
    type                             = 0x00 (EMBER_INCOMING_DATAGRAM)
    profileId                         = 0xABCD
    clusterId                         = 0x55
    sourceEndpoint                    = 0x11
    destinationEndpoint               = 0x12
    options                           = 0x00
    lastHopLqi                        = 0xF0
    lastHopRssi                       = 0xC4 (-60)
    sender                            = 0x0001
    bindingIndex                      = 0xFF
    datagramReplyTag                  = 0x01
    messageLength                     = 0x03
    messageContents                   = 0xE1, 0xE2, 0xE3

    SN260 -> HOST: | 80 | 45 | 00 | CD | AB | 55 | 11 | 12 | 00 | F0 | C4
                  | 01 | 00 | FF | 01 | 03 | E1 | E2 | E3 |
```

8 SIF module programming and debug interface

SIF is a synchronous serial interface developed by Cambridge Consultants Ltd. It is the primary programming and debug interface of the SN260. Therefore, any design implementing the SN260 should make the SIF signals readily available. The SIF module allows external devices to read and write memory-mapped registers in real-time without changing the functionality or timing of the XAP2b core. See the *SN260 reference design* for details regarding the implementation of the SIF interface.Go to www.stmcu.com for details.

The SIF interface provides the following:

- IC production test (especially analog)
- PCB production test
- Firmware download
- Product control and characterization

The pins are:

- nSIF_LOAD
- SIF_CLK
- SIF_MOSI
- SIF_MISO

The maximum serial shift speed for the SIF interface is 48MHz. SIF interface accesses can be initiated even when the chip is in idle, deep sleep, or power down modes. An edge on nSIF_LOAD wakes the chip to allow SIF cycles.

9 Typical application

Figure 12 illustrates the typical application circuit for the SN260. This figure does not contain all decoupling capacitance required by the SN260. The Balun provides the impedance transformation from the antenna to the SN260 for both TX and RX modes. The harmonic filter provides additional suppression of the second harmonic, which increases the margin over the FCC limit. The 24MHz crystal with loading capacitors is required and provides the high frequency source for the SN260. The RC debounce filter (R4 and C7) is suggested to improve the noise immunity of the nRESET logic (Pin 11).

The SIF (nSIF_LOAD, SIF_MOSI, SIF_MISO, and SIF_CLK) and packet trace signals (PTI_EN and PTI_TXD) should be brought out test points or, if space permits to a 10-pin, dual row, 0.05-inch pitch header footprint. With a header populated, a direct connection to the InSight Adapter is possible which enhances the debug capability of the SN260. For more information, refer to www.stmcu.com.

Figure 12. Typical application circuit

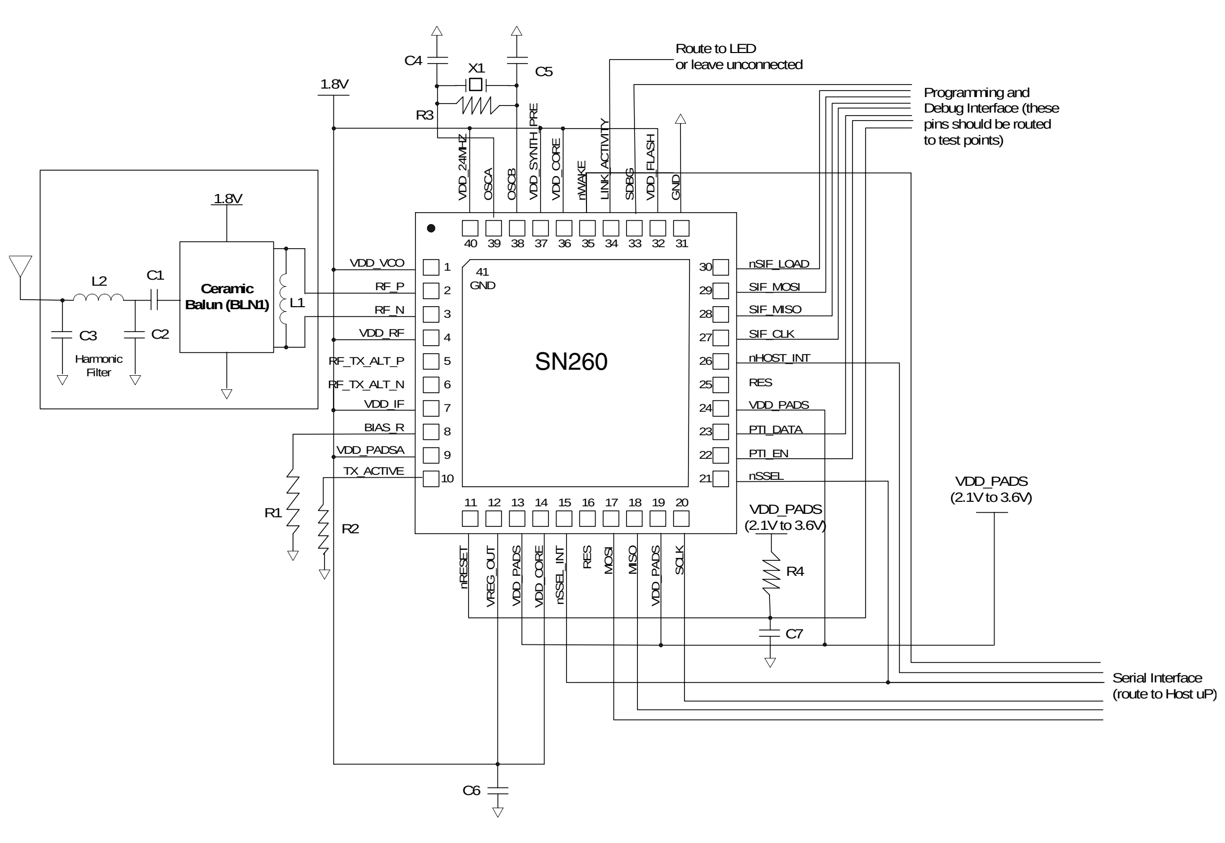


Table 120 contains the bill of materials for the application circuit shown in *Figure 12*.

Table 120. Bill of materials

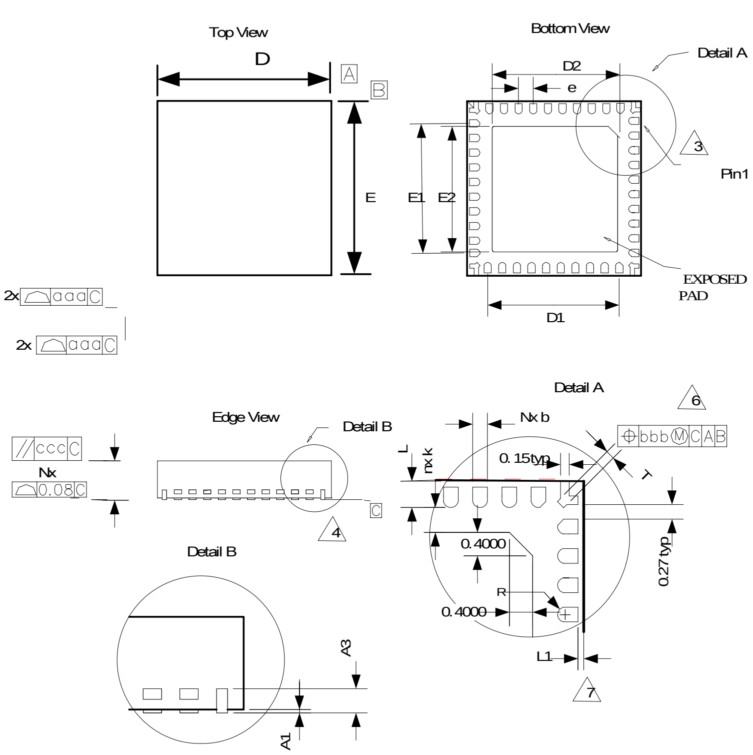
Item	Quantity	Reference	Description	Manufacturer/Part No.
1	1	C2	Capacitor, 5pF, 50V, NPO, 0402	
2	2	C1,C3	Capacitor, 0.5pF, 50V, NPO, 0402	
3	4	C4,C5	Capacitor, 27pF, 50V, NPO, 0402	
4	1	C6	Capacitor, 10µF, 10V, TANTALUM, 3216 (SIZE A)	
5	1	C7	Capacitor, 10pF, 5V, NPO, 0402	
6	1	L1	Inductor, 2.7nH, +/- 5%, 0603, multi-layer	MURATA LQG18HN2N7
7	2	L2	Inductor, 3.3nH, +/- 5%, 0603, multi-layer	MURATA LQG18HN3N3
8	1	R1	Resistor, 169 KΩ, 1%, 0402	
9	1	R2	Resistor, 100 KΩ, 5% O402	
10	1	R3	Resistor, 3.3 KΩ, 5% 0402	
11	1	R4	Resistor, 10 KΩ, 5%, 0402	
12	1	U1	SN260 single-chip Zigbee/802.15.4 solution	STMicroelectronics SN260
13	1	X1	Crystal, 24.000MHz, +/- 10PPM tolerance, +/- 25PPM stability, 18pF, - 40°C to + 85°C	ILSI ILCX08-JG5F18-24.000MHZ
14	1	BLN1	BALUN, ceramic	TDK HHM1521

10 Mechanical details

The SN260 package is a plastic 40-pin QFN that is 6mm x 6mm x 0.9mm. A large ground pad in the bottom center of the package forms an extra 41st pin. A number of thermal vias should connect the SN260 decal center to a PCB ground plane. For more information, refer to www.stmcu.com.

Figure 13 illustrates the package drawing.

Figure 13. Package drawing



Notes
1. JEDEC ref MO-220
2. All dimensions are in millimeters
3. Pin 1 orientation identified by chamfer on corner of exposed die pad
4. Datum C and the seating plane are defined by the flat surface of the metallised terminal
5. Dimension'e' represents the terminal pitch
6. Dimension b applies to metallised terminal and is measured 1.25 to 1.30 mm from terminal tip
7. Dimension L1 represents terminal pull back from package edge. Where terminal pull back exists, only upper half of lead is visible on package edge due to half etching of leadframe
8. Package surface shall be matte finish Ra 1.6 - 2.2
9. Package warp shall be 0.150 maximum
10. Leadframe material is copper A194.
11. Coplanarity applies to the exposed pad as well as the terminals

Common Dimensions (mm)			
Sym	Minimum	Nominal	Maximum
A	0.85	1.90	1.0
A1	0	1.02	1.05
A3		0.20ref	
D	5.90	6.11	6.10
D1		4.5 BSC	
D2	4.21	4.31	4.41
E	5.91	6.10	6.10
E1		4.5 BSC	
E2	4.21	4.31	4.41
L	0.35	0.41	1.45
L1			0.1
b	0.18	1.23	0.31
N		40	
e		0.50	
k	1.2		
R	b min/2		
T		0.15	

Sym	Tolerances for Form & Position	Notes
aaa	0.15	
bbb	0.10	
ccc	0.10	

11 Ordering information

Use the following part numbers to order the SN260:

- SN260QT Reel, RoHS
- SN260Q Tray, RoHS

To order parts, contact your local STMicroelectronics sales representative, or go to our Web site: www.st.com.

12 Abbreviations and acronyms

Table 121. Abbreviations and acronyms

Acronym/abbreviation	Meaning
ACR	Adjacent Channel Rejection
AES	Advanced Encryption Standard
CBC-MAC	Cipher Block Chaining—Message Authentication Code
CCA	Clear Channel Assessment
CCM	Counter with CBC-MAC Mode for AES encryption
CCM*	Improved Counter with CBC-MAC Mode for AES encryption
CSMA	Carrier Sense Multiple Access
CTR	Counter Mode
EEPROM	Electrically Erasable Programmable Read Only Memory
ESD	Electro Static Discharge
ESR	Equivalent Series Resistance
FFD	Full Function Device (ZigBee)
FIA	Flash Information Area
GPIO	General Purpose I/O (pins)
HF	High Frequency (24MHz)
I^2C	Inter-Integrated Circuit bus
IDE	Integrated Development Environment
IF	Intermediate Frequency
IP3	Third order Intermodulation Product
ISR	Interrupt Service Routine
kB	Kilobyte
kbps	kilobits/second
LF	Low Frequency
LNA	Low Noise Amplifier
LQI	Link Quality Indicator
MAC	Medium Access Control
MSL	Moisture Sensitivity Level
Msps	Mega samples per second
O-QPSK	Offset-Quadrature Phase Shift Keying
PA	Power Amplifier
PER	Packet Error Rate
PHY	Physical Layer

Table 121. Abbreviations and acronyms (continued)

Acronym/abbreviation	Meaning
PLL	Phase-Locked Loop
POR	Power-On-Reset
PSD	Power Spectral Density
PSRR	Power Supply Rejection Ratio
PTI	Packet Trace Interface
PWM	Pulse Width Modulation
RoHS	Restriction of Hazardous Substances
RSSI	Receive Signal Strength Indicator
SFD	Start Frame Delimiter
SIF	Serial Interface
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver/Transmitter
VCO	Voltage Controlled Oscillator
VDD	Voltage Supply

13 References

- IEEE 802.15.4-2003 (standards.ieee.org/getieee802/download/802.15.4-2003.pdf)
- IEEE 802.11g (standards.ieee.org/getieee802/download/802.11g-2003.pdf)
- Bluetooth Specification v1.2 (www.bluetooth.org/spec)
- ZigBee Specification v1.1 (www.zigbee.org; document number 053474r07)
- ZigBee Security Services Specification v1.0 (www.zigbee.org; document number 03322r13)

14 Revision history

Table 122. Document revision history

Date	Revision	Changes
11-Dec-2006	1	Initial release.
3-Dec-2007	2	Document status promoted from Preliminary Data to Datasheet.

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.

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