Instruction

Serial API Host Appl. Prg. Guide

Document No.:	INS12350
Version:	23
Description:	Guideline for developing serial API based host applications
Written By:	JFR;JSI;PSH;AES;JKA;JSMILJANIC;JROSEVALL;SSE;GAFARKAS
Date:	2022-12-01
Reviewed By:	JFR;JSI;SSE;GAFARKAS
Restrictions:	Public

Approved by:

Date	CET	Initials	Name	Justification
2022-12-01	15:26:53	JFR	Jorgen Franck	on behalf of NTJ

REVISION RECORD

Doc. Ver.	Date	By	Pages affected	Brief description of changes
1	20121026	ABR AES JSI JFR	ALL	Initial draft
2	20121109	AES	6.1	Initialization
3	20121130	ABR	6.4.2, 6.6.1 & 6.6.3	Added details to description of exception handling when receiving data frames.
4	20140602	JFR	7.1 5.4.5.1	Added application node information command Added funcID parameter description
5	20150327	PSH	7.19	Added description of the new NVM backup/restore command
6	20150915	JFR	4.1	Overview of communication interface versions
7	20151015	JFR	7.6	Added description of FUNC_ID_SERIAL_API_SETUP command
8	20160229	JSI	7.2	Added description of FUNC_ID_SERIAL_API_APPL_NODE_INFORMATION_CMD_CLASSES
8	20170119	PSH	7.4	Updated Node List command with SIS flag
9	20170126	PSH JSI	7.2	Added End Device Enhanced 232 based SerialAPI initialization list for FUNC_ID_SERIAL_API_APPL_NODE_INFORMATION_CMD_CLASSES
10	20170203	PSH	7.7 7.16	Updated with new RF powerlevel setup functions in 6.71.01 Added description of the FUNC_ID_SERIAL_API_STARTED command
11	20170215	JFR	4.1.5	Serial API interface version incremented to 7 in 6.71.0x due to introduction of S2 security
12	20170721	JFR	4.1.6	Serial API interface version incremented to 8 in 6.8x.0x due to introduction of Smart Start.
13	20180205	JSI	7.11	Added ZW_GetMaxPayloadSize description.
14	20180306	BBR	All	Added Silicon Labs template
15	20181005	KEWAHID	7.10.2 & 7.10.3	Added SERIAL_API_SETUP_CMD_RF_REGION_GET/SET commands
15	20181017	KEWAHID	-	Removed the section that explained how to backup and restore the RF Region setting when doing a firmware update. No longer relevant.
15	20181121	PSH	7.7	Changed the description of setting default Tx power.
16	20181204	JSI	7.2 & 7.19	Updated descriptions in connection with SDK 7.00.00 release.
17	20190201	JFR	7.15 7.19	FUNC_ID name corrected. New 'Return Values' added (0x02 & 0x03).
18	20200806	KEWAHID	7.14 & 7.16	Serial API interface version incremented to 9. Added description of new command to set "Node ID Base Type", and updated the "Serial API Started Command" with new Capabilities field.
18	20201104	JOROSEVA	7.20 & 7.21	Added documentation of new commands to get/set Long Range channel. Added documentation of new command to enable Long Range virtual node IDs.
18	20201216	JESMILJA	7.7 7.4.2 7.22 7.23 & 7.24	Added new chapter Supported SERIAL_API_SETUP_CMD commands Added documentation for function Get Long Range Nodes Update link to section "Serial API started Command" Add description for functions Zw_SendData, ApplicationCommandHandler_Bridge
18	20210108	LAMICCON	7.11	Added documentation of new command for Setting/Getting DCDC Configuration (FUNC_ID_SET_DCDC_CONFIG and FUNC_ID_GET_DCDC_CONFIG).
19	20210113	LAMICCON	7.9	Added documentation of command for retrieving the Background RSSI levels of the available channels (FUNC_ID_ZW_GET_BACKGROUND_RSSI).
20	20210125	JFR	7.10.2 7	Added Region US Z-Wave Long Range Obsoleted Power Manager
21	20210309	JOROSEVA	7.25 & 3	Added command for enabling PTI Zniffer functionality.
22	20220203	SASEOUD	7.4.1	Updates the table of supported chip types
23	20221201	JFR	Front Page & All	Set to public and fixed typos.

Table of Contents

Table of Figures

Table of Tables

1 ABBREVIATIONS

Abbreviation	Explanation
ACK	Acknowledgement
AES	The Advanced Encryption Standard is a symmetric block cipher algorithm. The AES is a NIST-standard cryptographic cipher that uses a block length of 128 bits and key lengths of 128, 192 or 256 bits. Officially replacing the Triple DES method in 2001, AES uses the Rijndael algorithm developed by Joan Daemen and Vincent Rijmen of Belgium.
ANZ	Australia/New Zealand
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
CAN	Cancel
DLL	Dynamic Link Library
DUT	Device Under Test
EU	Europe
GNU	An organization devoted to the creation and support of Open-Source software
HK	Hong Kong
HW	Hardware
IN	India
ISR	Interrupt Service Routines
JP	Japan
LRC	Longitudinal Redundancy Check
MY	Malaysia
NAK	Not Acknowledged
NWI	Network Wide Inclusion
PA	Power Amplifier
POR	Power on Reset
PRNG	Pseudo-Random Number Generator
PWM	Pulse Width Modulator
RF	Radio Frequency
RS-232	TIA-232-F Interface Between Data Terminal Equipment and Data Circuit-Terminating Equipment Employing Serial Binary Data Interchange
RU	Russian Federation
SDK	Software Developer's Kit
SIS	SUC ID Server
SOF	Start of Frame
SPI	Serial Peripheral Interface
SUC	Static Update Controller
US	United States
USB	Universal Serial Bus
USB CDC	Universal Serial Bus Communications Device Class
WUT	Wake Up Timer

2 INTRODUCTION

2.1 Purpose

This document describes the host processor application development using the serial API interface.

2.2 Terms

This document describes mandatory and optional aspects of the required compliance of a Z-Wave product to the Z-Wave standard.

The guidelines outlined in IETF RFC 2119 [1] with respect to key words used to indicate requirement levels are followed. Essentially, the key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

3 OVERVIEW

The Serial Applications Programming Interface (Serial API) allows a host to communicate with a Z-Wave chip. The host may be a PC or a less powerful embedded host CPU, e.g., in a remote control or in a gateway device. Depending on the chip family, the Serial API may be accessed via RS-232 or USB physical interfaces.

Sample applications demonstrate how to communicate with a Z-Wave chip via the Serial API.

The following host-based sample applications are available on the SDK:

- Z/IP Gateway – Gateway application using Serial API features of the bridge controller API
- PC Controller – Demonstrates Serial API features of the bridge controller API

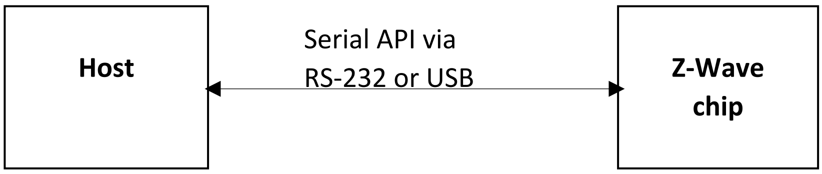


Figure 1. Communication via Serial API

The host-based sample applications are described in the respective SDK overview documents. For details refer to [2] and [4] for 500 and 700 SoCs respectively.

The serial API leverages the Z-Wave Protocol API. The serial API introduces additional messages related to inter-host communications. Mapping of serial API commands to the Z-Wave Protocol API calls can be found in [3]. Dedicated serial API commands are presented in section 7. A Z-Wave Alliance document describing the Serial API commands will also soon be available.

Serial API based applications MUST ensure that the required features are available in the actual Z-Wave library, using the “Capabilities Command” see 7.1.

4 COMMUNICATION INTERFACE

The following sections describe the Serial API.

4.1 Communication Interface Versions

4.1.1 Version 1-3

The differences between serial API communication interface versions 1 to 3 is not documented.

4.1.2 Version 4

The SDK 4.23-28 are based on the serial API communication interface version 4. This version introduces Serial API commands to better support a host application, especially the Serial API Capabilities Command. The Serial API Capabilities Command determines the Serial API functions that a specific Serial API Z-Wave Module supports.

4.1.3 Version 5

The SDK 4.51-55, 5.03.00, 6.02.00, 6.11.00-01, and 6.51.00-06 versions are based on the serial API communication interface version 5. The destNode is appended to end of ApplicationCommandHandler REQ and promiscuously received frames are returned in a FUNC_ID_PROMISCUOUS_APPLICATION_COMMAND_HANDLER REQ.

Some of the SDKs introduce new Serial API functions, which can be determined by the Serial API Capabilities Command.

4.1.4 Version 6

SDK 6.60.00 changes the Serial API communication interface to version 6 due to several extensions of the serial API [3] to better support installation and maintenance procedures (RSSI feedback, Routing algorithm feedback, and Network statistics). The interface is backward compatible with version 5, provided appended parameters in version 6 are ignored.

The ZW_SendData (and variations) callback parameters are changed and extended to include more information (transmission metrics) about the successful/unsuccessful transmission. The change affects the Serial API functionality FUNC_ID_ZW_SEND_DATA (and FUNC_ID_SEND_DATA_BRIDGE) because the transmission metrics are appended to the callback parameter list to ensure that an application, which ignores the extra data in the callback parameter list can function with no change. A Serial API functionality FUNC_ID_SERIAL_API_SETUP is implemented to enable or disable appending transmission metrics in the FUNC_ID_ZW_SEND_DATA callback.

The ApplicationCommandHandler (and ApplicationCommandHandler_Bridge) parameter list is changed and extended to also include the RSSI value with which the received frame is received. The change affects the Serial API functionality FUNC_ID_APPLICATION_COMMANDHANDLER (and

FUNC_ID_APPLICATION_COMMAND_HANDLER_BRIDGE) because it appends the RSSI value to the functionality parameter for HOST implementations, which do not ignore the extra data.

In the Z-Wave protocol, the functions ZW_GetLastWorkingRoute and ZW_SetLastWorkingRoute are obsolete and replaced with ZW_GetPriorityRoute and ZW_SetPriorityRoute respectively. FUNC_ID_ZW_GET_LAST_WORKING_ROUTE and FUNC_ID_ZW_SET_LAST_WORKING_ROUTE functions are obsolete and replaced with FUNC_ID_ZW_GET_PRIORITY_ROUTE / FUNC_ID_ZW_SET_PRIORITY_ROUTE respectively.

End device enhanced-based Serial API targets is extended with two new functionalities to accommodate for the new protocol functionalities ZW_AssignPriorityReturnRoute and ZW_AssignPrioritySUCReturnRoute: FUNC_ID_ZW_ASSIGN_PRIORITY_SUC_RETURN_ROUTE and FUNC_ID_ZW_ASSIGN_PRIORITY_SUC_RETURN_ROUTE.

The new Z-Wave protocol function ZW_ExploreRequestExclusion is implemented in the Serial API with the funcID FUNC_ID_ZW_EXPLORE_REQUEST_EXCLUSION.

The new protocol functionalities ZW_GetNetworkStats and ZW_ClearNetworkStats is implemented in the Serial API with the funcIDs FUNC_ID_ZW_GET_NETWORK_STATS and FUNC_ID_ZW_CLEAR_NETWORK_STATS respectively.

New Serial API functionality (FUNC_ID_NVM_BACKUP_RESTORE) to back up and restore NVM contents is added to the serial API in all controllers and enhanced end devices.

The FUNC_ID_ZW_REDISCOVERY_NEEDED is obsolete.

SDK 6.70.00 introduces new Serial API functions for End Device Enhanced 232 library-based targets to enable HOSTs to leverage the new functionality.

4.1.5 Version 7

SDK 6.71.0x changes the Serial API communication interface to version 7, enabling host software to ensure that this version of the serial API supports S2.

4.1.6 Version 8

SDK 6.80.0x changes the Serial API communication interface to version 8 enabling the host software to ensure that this version of the serial API supports Smart Start.

4.1.7 Version 9

SDK 7.14.x changer changes the Serial API communication interface to version 9 enabling the host software to ensure that this version of the serial API supports Z-Wave Long Range.

4.2 Communication Channel Settings

4.2.1 RS-232 Serial Port

A host communicating to a Serial API library via a serial port MUST use the following settings.

Parameter	Value
Baud rate	115200 bits/s
Parity	No
Data bits	8
Stop bits	1

Table 1. Serial API RS-232 Parameters

The least significant bit (LSB) b0 of each byte MUST be transmitted first on the physical wire.

4.2.2 USB Serial Port

A host communicating to a Serial API library via a USB connection MUST follow the guidelines for the USB communications device class (USB CDC). In many cases, Linux® OS distributions and Mac OS releases will immediately present the Z-Wave chip USB interface as a serial port to applications.

Windows OS releases may need an .inf file structure to present the Z-Wave chip USB interface as a serial port to applications:

Key	Value
[Version]	Signature="$Windows NT$" Class=Ports ClassGuid={4D36E978-E325-11CE-BFC1-08002BE10318} Provider=%manu% DriverVer=02/17/2010,0.0.3.0
[Manufacturer]	%manu%=ZComDev, NTx86, NTamd64
[ZComDev.NTx86]	%dev%=ZComInst, USB\VID_0658&PID_0200
[ZComDev.NTamd64]	%dev%=ZComInst, USB\VID_0658&PID_0200
[ZComInst]	include=mdmcpq.inf CopyFiles=FakeModemCopyFileSection AddReg=LowerFilterAddReg,SerialPropPageAddReg
[ZComInst.Services]	include = mdmcpq.inf AddService = usbser, 0x00000002, LowerFilter_Service_Inst
[SerialPropPageAddReg]	HKR,,EnumPropPages32,,"MsPorts.dll,SerialPortPropPageProvider"
[Strings]	manu = "Silicon Labs" dev = "UZB" svc = "UZB"

Table 2. Serial API USB Windows .inf File Structure

5 FRAME LAYOUT

The host and the Z-Wave chip (ZW) communicate through a simple protocol, which uses ACK, NAK, CAN, and Data frame types.

5.1 ACK Frame

The ACK frame indicates that the receiving end has received a valid Data frame.

The host MUST wait for an ACK frame after transmitting a Data frame to the Z-Wave chip. If transmission errors or race conditions occur, the host may receive other frames or no frames at all. The host MUST be robust to handle such events. The host SHOULD queue up requests for processing once the expected ACK frame is received or timed out. The host MUST wait 1500 ms before timing out waiting for the ACK frame.

A receiving Z-Wave chip MUST return an ACK frame in response to a valid Data frame.

7 (MSB)	6	5	4	3	2	1	0 (LSB)
ACK (0x06)							

Figure 2. ACK Frame

5.2 NAK Frame

The NAK frame indicates that the receiving end has received a Data frame with errors.

If a transmitting host or Z-Wave chip receives a NAK frame in response to a Data frame, it MAY retransmit the Data frame.

7 (MSB)	6	5	4	3	2	1	0 (LSB)
NAK (0x15)							

Figure 3. NAK Frame

A transmitting host or Z-Wave chip receiving a NAK frame MUST wait for a while before retransmitting the Data frame. See section 6.3.

5.3 CAN Frame

The CAN frame indicates that the receiving end has discarded an otherwise valid Data frame. The CAN frame is used to resolve race conditions, where both ends send a Data frame and subsequently expects an ACK frame from the other end.

.7 (MSB)	6	5	4	3	2	1	0 (LSB)
CAN (0x18)							

Figure 4. CAN Frame

If a Z-Wave chip expects to receive an ACK frame but receives a Data frame from the host, the Z-Wave chip SHOULD return a CAN frame. A host which receives a CAN frame MUST consider the Data frame lost. The host MUST wait for a while before retransmitting the Data frame. See section 6.3.

5.4 Data Frame

Data frame contains the Serial API command including parameters for the command in question.

Each Data frame MUST consist of a Serial API command including parameters for the command prepended with Start of Frame (SOF), Length and Type fields, and a Checksum byte appended.

A transmitting host or Z-Wave chip may time out waiting for an ACK frame after transmitting a Data frame. The transmitting end MUST wait for ACK frame for a period. If no ACK frame is received, the Data frame MAY be retransmitted. See section 6.3.

7 (MSB)	6	5	4	3	2	1	0 (LSB)
SOF (0x01)							
Length							
Type							
Serial API Command ID							
Serial API Command Parameter 1							
…							
Serial API Command Parameter n							
Checksum							

Figure 5. Data Frame

5.4.1 Start of Frame (SOF)

The Start of Frame (SOF) field is used for synchronization. The SOF field MUST have a value of 0x01. A host or a Z-Wave chip waiting for new traffic MUST ignore all other byte values except 0x06 (ACK), 0x15 (NAK), 0x18 (CAN), or 0x01 (Data frame). This way, both receivers will flush garbage bytes from the receive buffer to get back in sync after a connection glitch or a firmware restart in one of the ends.

5.4.2 Length

The Length field MUST report the number of bytes in the Data frame. The value of the Length Field MUST NOT include the SOF and Checksum fields. A host or a Z-Wave chip receiving a Data frame SHOULD validate the length field by comparing the number of received bytes and the Length field (expecting a difference of 2 bytes).

5.4.3 Type

The Type field MUST indicate if the Data frame type is Request or Response.

Value	Type	Description
0x00	REQ	Request. This type MUST be used for unsolicited messages. API callback messages MUST use the Request type.
0x01	RES	Response. This type MUST be used for messages that are responses to Requests.
0x02..0xFF	*Reserved*	Reserved values MUST NOT be used. A receiving end MUST ignore reserved Type values.

Table 3. Data Frame :: Type Values

5.4.4 Serial API Command ID

The Serial API Command ID field MUST carry one of the valid API function codes defined in section 7. A host or Z-Wave chip MUST report the same Serial API Command ID in a response Data frame (see section 5.4.3).

5.4.5 Serial API Command Parameters

The Serial API Command Parameters field MAY have a variable number of bytes. The field MUST be at least one byte long. A receiving end MUST derive the actual number of bytes from the Length field. See section 5.4.2.

Information carried in the Serial API Command Parameters field MUST comply with the API function prototype for the Serial API Command ID carried in the Serial API Command ID field. See section 5.4.4.

API function prototypes may be found in section 7.

5.4.5.1 funcID Parameter

Some Serial API calls contain a funcID parameter. Any funcID value different than zero is returned in the callback function making it possible to correlate the callback with the original request. Setting funcID to zero disables the callback function via serial API.

5.4.6 Checksum

The Checksum field MUST carry a checksum to enable frame integrity checks. The checksum calculation MUST include the **Length**, **Type**, **Serial API Command Data,** and **Serial API Command Parameters** fields.

The checksum value MUST be calculated as an 8-bit Longitudinal Redundancy Check (LRC) value. The RECOMMENDED way to calculate the checksum is to initialize the checksum to 0xFF and then XOR each of the bytes of the fields mentioned above one at a time to the checksum value.

$$\text{Checksum} = \text{0xFF} \oplus \text{Length} \oplus \text{Type} \oplus \text{Cmd ID} \oplus \text{Cmd Parm}[1] \oplus \ldots \oplus \text{Cmd Parm}[n]$$

A Data frame MUST be considered invalid if it is received with an invalid checksum. See section 5.4.6. A host or Z-Wave chip MUST return a NAK frame in response to an invalid Data frame.

6 TRANSMISSION

6.1 Initialization

To ensure the host and the Z-Wave module are in sync at application startup, the host should begin an initialization sequence. The initialization sequence is different depending on whether the host has access to a module hard reset.

6.1.1 With Hard Reset

1) Close host serial port if it is open.
2) Assert module reset.
3) Open the host serial port at 115200 baud 8N1.
4) Release module reset.
5) Wait 500 ms.

6.1.2 Without Hard Reset

1) Close host serial port if it is open.
2) Open the host serial port at 115200 baud 8N1.
3) Send the NAK.
4) Send Serial API command: FUNC_ID_SERIAL_API_SOFT_RESET.
5) Wait 1.5 s.

This solution is not recommended because it relies on retrieval and execution of the Serial API command FUNC_ID_SERIAL_API_SOFT_RESET.

6.2 Frame Timing

6.2.1 Data Frame Reception Timeout

A receiving host or Z-Wave chip MUST abort reception of a Data frame if the reception has lasted for more than 1500 ms after the reception of the SOF byte. A host or Z-Wave chip MUST NOT issue a NAK frame after aborting the reception of a Data frame.

6.2.2 Data Frame Delivery Timeout

A host or Z-Wave chip MUST wait for an ACK frame after transmitting a Data frame. The receiver may be waiting for up to 1500 ms for the remains of a corrupted frame (see section 6.2.1). Therefore, the transmitter MUST wait for at least 1600 ms before deeming the Data frame lost.

The loss of a Data frame MUST be treated as the reception of a NAK frame. See section 6.3. The transmitter MAY compensate for the 1600 ms already elapsed when calculating the retransmission waiting period.

6.3 Retransmission

A transmitter may time out waiting for an ACK frame after transmitting a Data frame or it may receive a NAK or a CAN frame. In either case, the transmitter SHOULD retransmit the Data frame. A waiting period MUST be applied before the retransmission.

The waiting period MUST be calculated per the following formula:

$$T_{waiting} = 100ms + n*1000ms$$

where n is incremented at each retransmission. n=0 is used for the first waiting period.

A host or Z-Wave chip MUST NOT carry out more than three retransmissions. Note that a host MAY choose to do a hard reset of the Z-Wave module if it is not able to successfully deliver the frame after three retransmissions. Flush/reopen the serial port after the three retransmissions.

6.4 Exception Handling

6.4.1 Unresponsive Z-Wave Module

In the unlikely event that the Z-Wave module becomes unresponsive for more than 4 seconds, it is RECOMMENDED to issue a hard reset of the module. A module may be deemed unresponsive if it has not responded with any character after three consecutive frame retransmissions, each with a 1600 ms interval. See section 6.1.

6.4.2 Persistent CRC Errors

If a host application detects an invalid checksum three times in a row when receiving data frames, the host application SHOULD invoke a hard reset of the device. If a hard reset line is not available, a soft reset indication SHOULD be issued for the device.

6.4.3 Missing Callbacks

In some situations, a serial API callback may be lost due to an overflow in the UART transmit buffer. This condition may occur if a lot of unsolicited traffic comes in from the Z-Wave side. For this reason, a Serial API-based host application SHOULD guard all its callbacks with a timer. The timer values are given in references [3] for each of the Z-Wave API functions which use callbacks.

6.5 Frame Flow

The frame flow between a host and a Z-Wave module (ZW) running the Serial API embedded sample code depends on the API call. There are two classes of communication between a host and a Z-Wave chip: Unsolicited and Request/Response. Each of the classes is presented below.

6.5.1 Unsolicited Frame Flow

The most basic frame flow is a Request (REQ) Data frame that is acknowledged by an ACK frame.

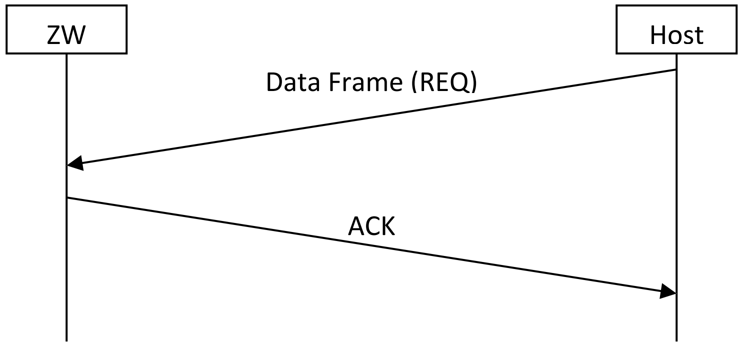


Figure 6. Unsolicited Data Frame

API call **ZW_SetExtIntLevel** is an example of the frame flow outlined in the figure above.

A variant of the REQ Data frame flow is a request (REQ) Data frame in one direction followed by a request (REQ) Data frame in the opposite direction. The first Data frame is acknowledged before a Data frame is transmitted in the opposite direction.

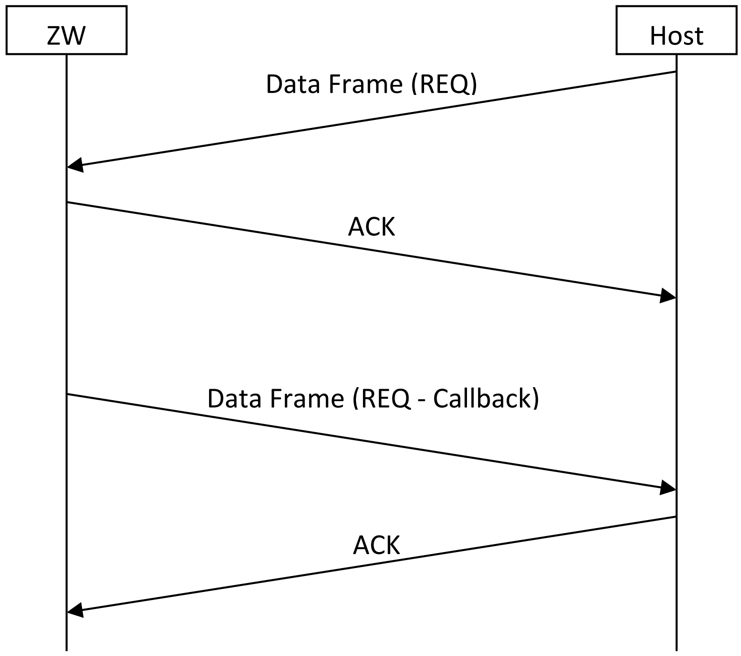


Figure 7. Unsolicited Data Frame Followed by Unsolicited Data Frame

Typically, the REQ Data frame in the opposite direction follows after some time.

The API call **ZW_SetDefault** is an example of the frame flow outlined in the figure above, where the second Data frame is carrying a callback message indicating the completion of the operation.

6.5.2 Request/Response Frame Flow

A Request (REQ) Data frame may be followed by a Result (RES) Data frame within a few second interval. This flow is used for all functions which have a non-void return value. Note that, due to the simple nature of the simple acknowledge mechanism, only one REQ->RES session is allowed.

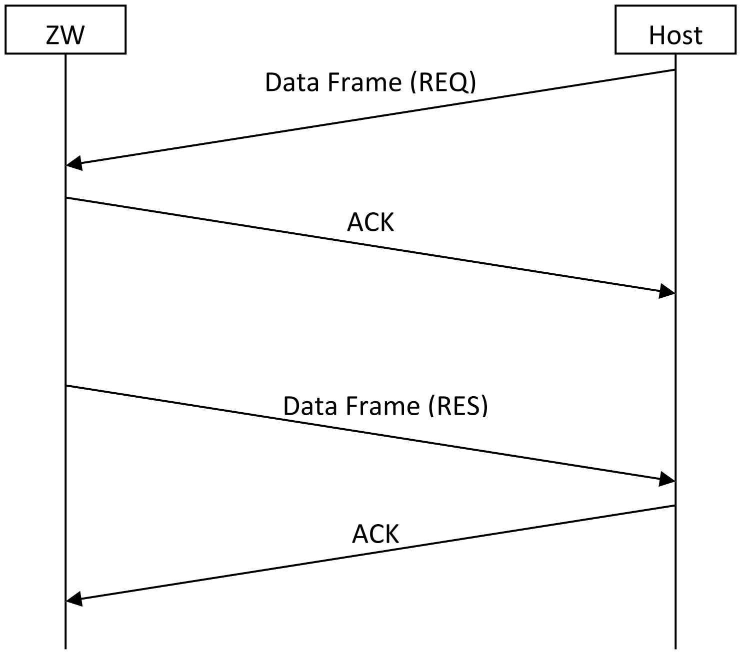


Figure 8. Request/Response Data Frames

The API call **ZW_GetControllerCapabilities** is an example of the frame flow outlined in the figure above, where the Result Data frame is carrying the requested controller capabilities.

A variant of the Request/Response Data frame involves an unsolicited Data frame following the Request/Response Data frame pair. Typically, the REQ Data frame in the opposite direction follows after some time.

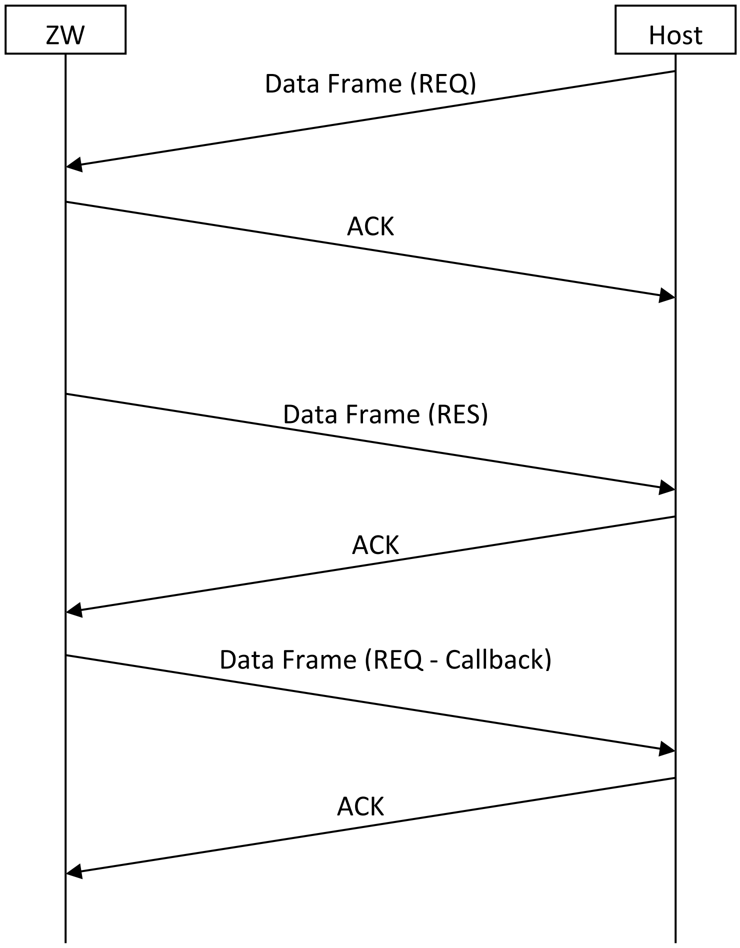


Figure 9. Request/Response Data Frames Followed by Unsolicited Data Frame

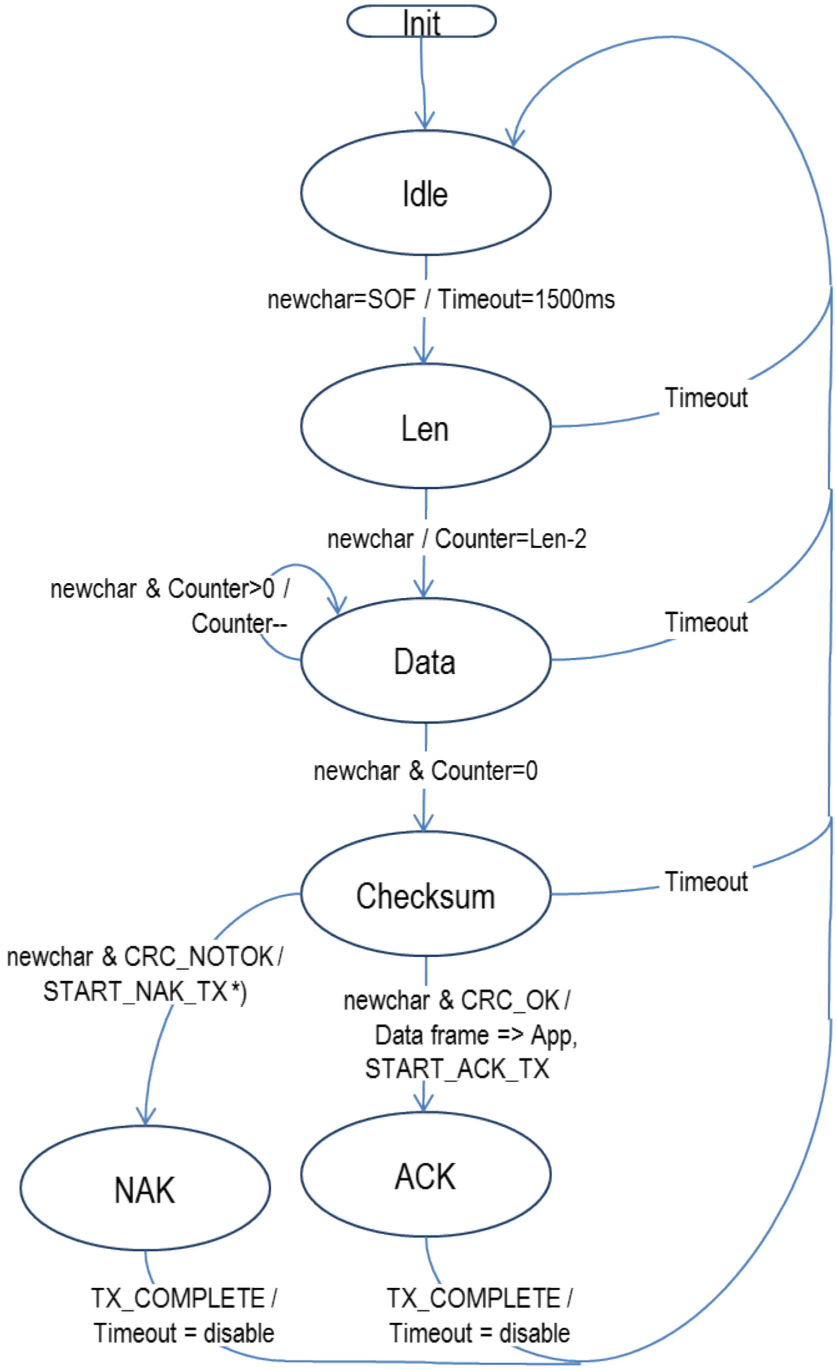
The API call **ZW_SendSUCID** is an example of the frame flow outlined in the figure above**,** where the Result Data frame is carrying the requested controller capabilities and the second Data frame is carrying a callback message indicating the completion of the operation.

If a host application repeatedly receives a reception timeout error indication rather than a valid response Data frame, the host application SHOULD invoke a hard reset of the device. If a hard reset line is not available, a soft reset indication SHOULD be issued for the device.

6.6 State Diagrams

This chapter outlines a transmission and reception of Control and Data frames.

6.6.1 Host Data Frame Reception



*) If a host application detects an invalid checksum three times in a row when receiving data frames, the host application SHOULD invoke a hard reset of the device. If a hard reset line is not available, a soft reset indication SHOULD be issued for the device.

Figure 10. Host Data Frame Reception

States	
Idle	Waiting for a new char Ignore all other values than the SOF code. Event: newchar & [SOF] => New state: <Len> Actions: Enable receive timeout timer
Len	Waiting for the Len byte Event: newchar => New state: <Data> Actions: Set counter=Len-2 Event: timeout => New state: <Idle>
Data	Waiting for data: Type, Cmd and parameter fields. Information is passed on to the Serial API command handler when validated Event: newchar & Counter>0 => New state: <Data> Actions: Counter-- Event: newchar&Counter=0 => New state: <Checksum> Actions: (none) Event: timeout => New state: <Idle>
Checksum	Waiting for the Checksum byte Event: newchar & CRC_NOTOK => New state: <NAK> Actions: Initiate transmission of NAK frame Event: newchar & CRC_NOTOK => New state: <ACK> Actions: Forward Data frame to Serial API command handler Initiate transmission of ACK frame If a host application detects an invalid checksum three times in a row when receiving data frames, the host application SHOULD invoke a hard reset of the device. If a hard reset line is not available, a soft reset indication SHOULD be issued for the device.
NAK	Waiting for transmission of NAK frame Event: TX completion => New state: <Idle> Actions: Disable timeout
ACK	Waiting for transmission of ACK frame Event: TX completion => New state: <Idle> Actions: Disable timeout

6.6.1.1 Counter Maintenance

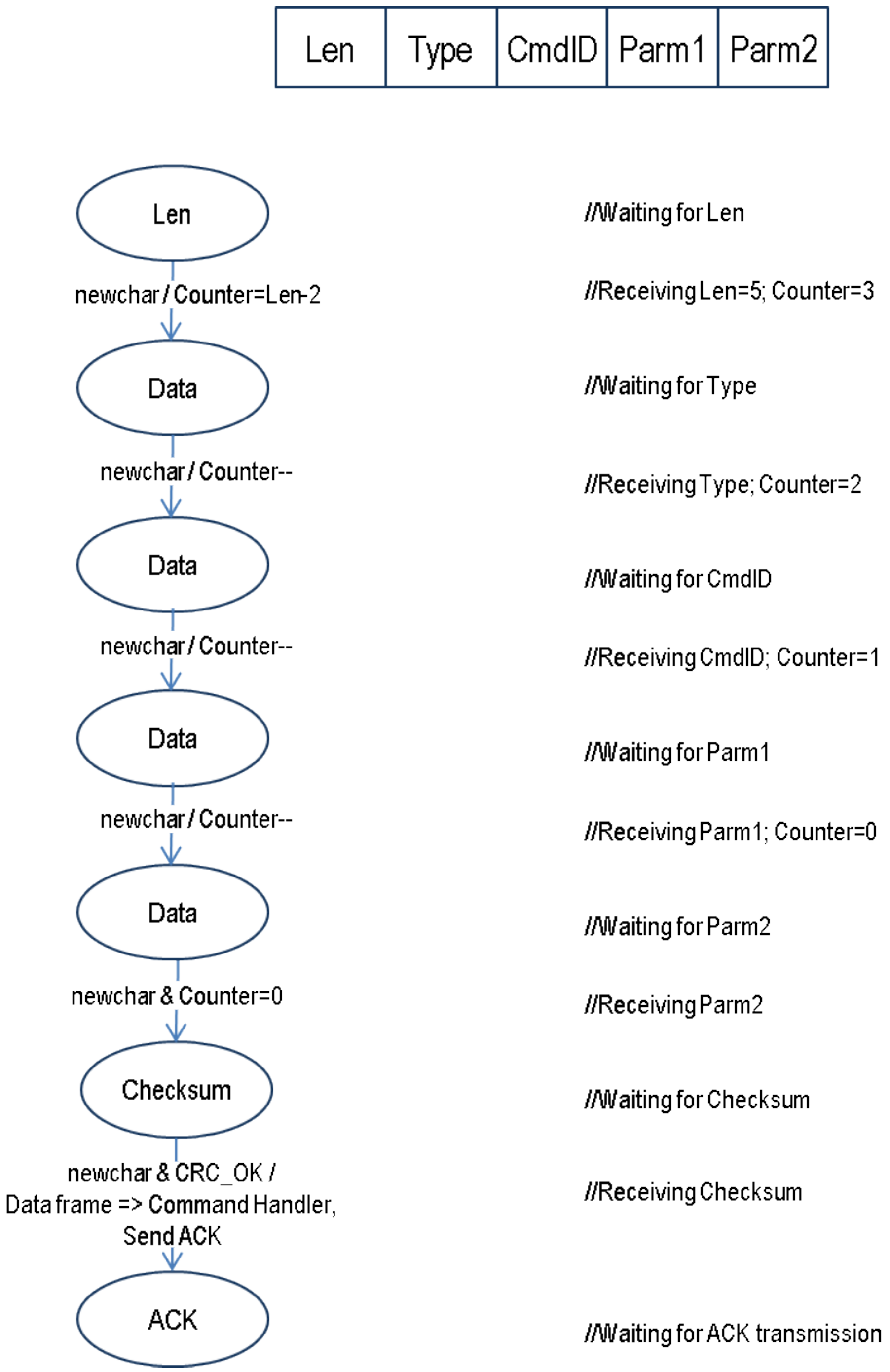


Figure 11. Counter Maintenance

6.6.2 Host Media Access Control

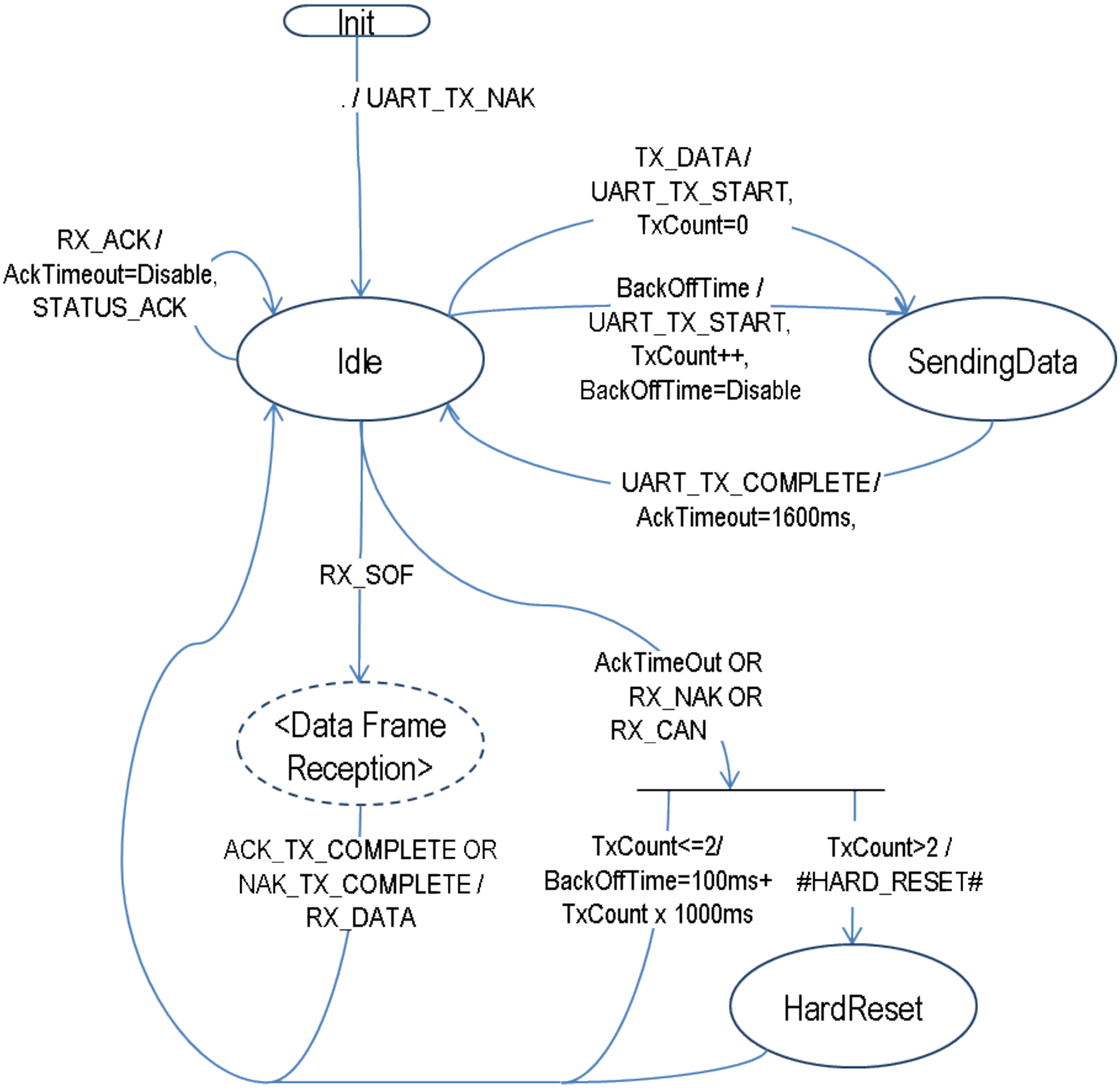


Figure 12. Host Media Access Control

States	

Idle	Waiting for events
	
	Event: (Init) => // Send NAK frame to force the remote end to re-send a Data frame if such
	// a frame is waiting for acknowledgment.
	
	Event: TX_DATA =>
	New state: <SendingData>
	Actions: Generate UART_TX_START event
	Initialize TxCount retransmission counter
	
	Event: BackOffTime =>
	New state: < SendingData >
	Actions: Generate UART_TX_START event
	Disable BackOffTime timer
	Increment TxCount retransmission counter
	
	Event: AckTimeOut OR RX_NAK OR RX_CAN =>
	New state: <Idle>
	Actions: IF (TxCount<=2) THEN
	Set BackOffTime = 100ms + TxCount x 1000ms
	ELSE
	Do a module hard reset
	ENDIF
	
	Event: RX_SOF =>
	New state: <Data Frame Reception> - SEPARATE CHART
	Actions: (none)
	
	Event: RX_ACK =>
	New state: <Idle>
	Actions: Disable ACK timeout timer

SendingData	Waiting for frame transmission to be completed Event: UART_TX_COMPLETE => New state: <Idle> Actions: Set ACK timeout = 1600ms
<Data Frame Reception>	Waiting for data frame reception to be completed <Data Frame Reception> is a self-contained state diagram. Stay here until finished. Event: ACK_TX_COMPLETE OR NAK_TX_COMPLETE => New state: <Idle> Actions: (none)
HardReset	Waiting for Hard Reset to be invoked Event: (Hard Reset) => New state: <Idle> Actions: (none)

6.6.3 Host Request/Response Session

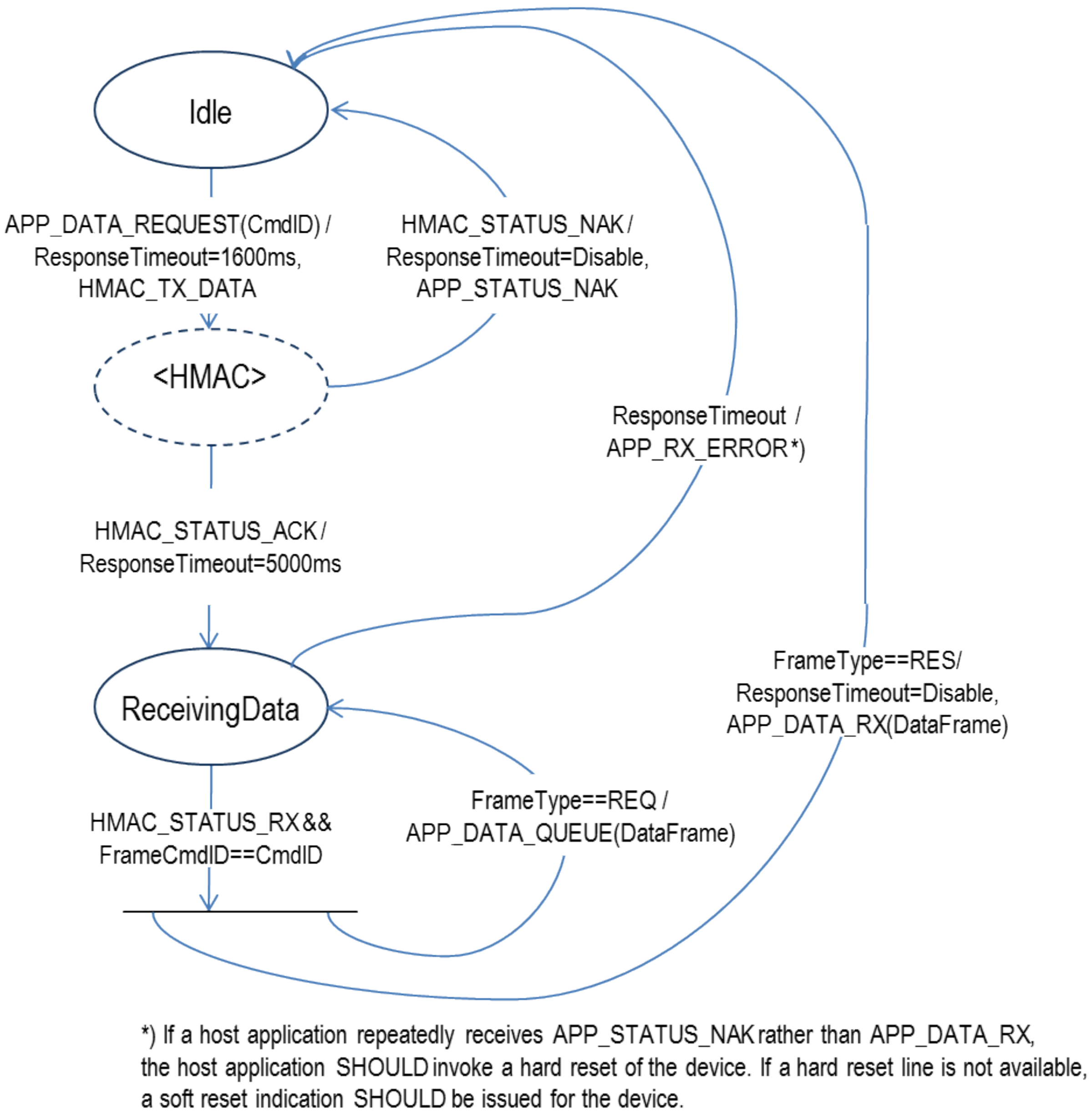


Figure 13. Host Request/Response Session

States	
Idle	Waiting for application request Event: APP_DATA_REQUEST(CmdID) => New state: <Host Media Access Control (HMAC)> Actions: Enable response timeout timer for ACK timeout period = 1600ms, Generate HMAC_TX_DATA event for <HMAC> state diagram.
<HMAC>	Waiting for transmission and acknowledgment of Data frame <HMAC> is a self-contained state diagram. Stay here until finished. Event: HMAC_STATUS_ACK => New state: <ReceivingData> Actions: Enable response timeout timer for response timeout period = 5000ms Event: HMAC_STATUS_NAK => New state: <Idle>, Actions: Disable response timeout timer, Generate APP_STATUS_NAK event for application.

ReceivingData	Waiting for response Data frame
	
	Event: HMAC_STATUS_RX && FrameCmdID<>CmdID && FrameType==REQ =>
	New state: <ReceivingData>
	Actions: Generate APP_DATA_QUEUE(Data frame) event for application.
	// Queue Data frame for later processing. Keep waiting for response.
	
	Event: HMAC_STATUS_RX && FrameCmdID<>CmdID && FrameType==RES =>
	New state: <Idle>
	Actions: Disable response timeout timer,
	Generate APP_DATA_RX(Data frame) event for application
	// Notify application that a response Data frame was received
	
	Event: ResponseTimeout =>
	New state: <Idle>
	Actions: Notify application that a response Data frame was not received
	NOTE: If a host application repeatedly receives APP_RX_ERROR rather than APP_DATA_RX, the host application SHOULD invoke a hard reset of the device. If a hard reset line is not available, a soft reset indication SHOULD be issued for the device.

7 SERIAL API COMMANDS

Besides the Z-Wave API function calls described in "Z-Wave application programmers' guide" the Serial API support a set of additional commands.

7.1 Application Node Information Command

Starting with the Serial API protocol version 4, it is possible to call Serial API Application Node Information Command to store a new Node Information Frame (NIF). Prior to either starting or joining a Z-Wave network, the HOST needs to initially set up the Node Information Frame (NIF), which should define the type of Z-Wave node the Serial API module is supposed to be. To store the NIF in the protocol NVM area as well as in the application NVM area, the HOST needs to perform the following steps:

1. HOST->ZW: send **SerialAPI_ApplicationNodeInformation()** with NIF information.

2. HOST->ZW: send **ZW_SetDefault().**

Serial API:

HOST->ZW: REQ | 0x03 | deviceOptionsMask | generic | specific | parmLength | nodeParm[ ]

For more details, see the relevant Application Programming Guide [3].

7.2 Application Node Information Command Classes Command

Starting with SDK 6.71.00, HOSTs connected to Serial API modules based on the End Device Routing or End Device Enhanced 232 library can set the Command Classes which should be supported in NOT Included, Insecurely Included, and Securely Included inclusion states. Supported command classes as set through the Serial API Application Node Information Command Classes Command with the

FUNC_ID_SERIAL_API_APPL_NODE_INFORMATION_CMD_CLASSES Serial API command.

Serial API:

HOST->ZW: REQ | 0x0C | unincluded_parmLength | unincluded_nodeParm[unincluded_parmLength] | included_unsecure_parmLength | included_unsecure_nodeParm[included_unsecure_parmLength] | included_secure_parmLength | included_secure_nodeParm[included_secure_parmLength]

ZW->HOST: RES | 0x0C | status

Status:
0x01: Success

SDK 6.71.00 adds Security to the protocol in the End Device Routing and End Device Enhanced 232 libraries resulting in additional setup steps before entering the inclusion process. Prior to joining a Z-Wave network, the HOST needs to initially set up which Security keyclasses (S0, S2_UNATHENTICATED, S2_AUTHENTICATED, S2_ACCESS) the node should apply for (if any). Afterwards the Security Authentication method must be specified. The Node Information Frame (NIF) which should define the type of Z-Wave node the Serial API module should be defined with regard to Listening, FLiRS, Generic type, Specific Type. Finally, the supported Command Classes for the various inclusion states should be set up.

Serial API:

In the Serial API, the Security API functions are reached through the FUNC_ID_ZW_SECURITY_SETUP (0x9C). Serial API FUNC_ID makes it possible to set the Requested Security Keys and Requested Authentication method in a End Device Routing/Enhanced 232-based Serial API Node prior to inclusion (add). The protocol requests the Requested Security Keys and Authentication during S2 inclusion.

Set Security Inclusion Requested Keys
(E_SECURITY_SETUP_CMD_SET_SECURITY_INCLUSION_REQUESTED_KEYS):

HOST->ZW: REQ | 0x9C | 5 | registeredSecurityKeysLen(1) | registeredSecurityKeys
ZW->HOST: RES | 0x9C | 5 | retValLen(1) | retVal
- retVal == TRUE => success

Set Security Inclusion Requested Authentication
(E_SECURITY_SETUP_CMD_SET_SECURITY_INCLUSION_REQUESTED_AUTHENTICATION):

HOST->ZW: REQ | 0x9C | 6 | registeredSecurityAuthenticationLen(1) | registeredSecurityAuthentication
ZW->HOST: RES | 0x9C | 6 | retValLen(1) | retVal

Or if unsupported

ZW->HOST: RES | 0x9C | 0xFF | retValLen(1) | retVal

- retVal == TRUE => success

HOSTs, which are connected to a Serial API module based on either End Device Enhanced 232 or End Device Routing libraries should follow the list below for correct module setup prior to joining a Z-Wave network:

1. HOST->ZW: send **SerialAPI_SetSecurityInclusionRequestedKeys**
2. HOST->ZW: send **SerialAPI_SetSecurityInclusionRequestedAuthentication**
3. HOST->ZW: send **SerialAPI_ApplicationNodeInformation()** with NIF information (listening, generic, specific)
4. HOST->ZW: send **SerialAPI_ApplicationNodeInformationCmdClasses**
5. HOST->ZW: send **ZW_SetDefault()**

7.3 Capabilities Command

Starting with the Serial API protocol version 4, users can call the Serial API Capabilities Command to determine which Serial API functions a specific Serial API Z-Wave Module supports with the FUNC_ID_SERIAL_API_GET_CAPABILITIES Serial API function:

Serial API:

HOST->ZW: REQ | 0x07

ZW->HOST: RES | 0x07 | SERIAL_APPL_VERSION | SERIAL_APPL_REVISION | SERIALAPI_MANUFACTURER_ID1 | SERIALAPI_MANUFACTURER_ID2 | SERIALAPI_MANUFACTURER_PRODUCT_TYPE1 | SERIALAPI_MANUFACTURER_PRODUCT_TYPE2 | SERIALAPI_MANUFACTURER_PRODUCT_ID1 | SERIALAPI_MANUFACTURER_PRODUCT_ID2 | FUNCID_SUPPORTED_BITMASK[ ]

SERIAL_APPL_VERSION is the Serial API application Version number.

SERIAL_APPL_REVISION is the Serial API application Revision number.

SERIALAPI_MANUFACTURER_ID1 is the Serial API application manufacturer_id (MSB).

SERIALAPI_MANUFACTURER_ID2 is the Serial API application manufacturer_id (LSB).

SERIALAPI_MANUFACTURER_PRODUCT_TYPE1 is the Serial API application manufacturer product type (MSB).

SERIALAPI_MANUFACTURER_PRODUCT_TYPE2 is the Serial API application manufacturer product type (LSB).

SERIALAPI_MANUFACTURER_PRODUCT_ID1 is the Serial API application manufacturer product ID (MSB).

SERIALAPI_MANUFACTURER_PRODUCT_ID2 is the Serial API application manufacturer product ID (LSB).

FUNCID_SUPPORTED_BITMASK[ ] is a bitmask where every supported Serial API function ID has a corresponding bit in the bitmask set to ‘1’. All unsupported Serial API function IDs have their corresponding bit set to ‘0’. The first byte in bitmask corresponds to FuncIDs 1-8, where bit 0 corresponds to FuncID 1 and bit 7 corresponds to FuncID 8. The second byte in bitmask corresponds to FuncIDs 9-16, and so on.

7.4 Node List Command

7.4.1 Get Init Data

Starting with the Serial API protocol version 4, users can call the Serial API Node List Command to determine the Serial API protocol version number, Serial API capabilities, nodes currently stored in the external NVM (only controllers), and a chip used in a specific Serial API Z-Wave Module with the FUNC_ID_SERIAL_API_GET_INIT_DATA Serial API function:

Serial API:

HOST->ZW: REQ | 0x02

(Controller) ZW->HOST: RES | 0x02 | ver | capabilities | 29 | nodes[29] | chip_type | chip_version

(End Device) ZW->HOST: RES | 0x02 | ver | capabilities | 0 | chip_type | chip_version

The parameter 'ver' is the Serial API application Version number.

The parameter 'capabilities' is a byte holding various flags describing the actual mode.

Capabilities flags:
Bit 0: 0 = Controller API; 1 = End device API
Bit 1: 0 = Timer functions not supported; 1 = Timer functions supported.
Bit 2: 0 = Primary Controller; 1 = Secondary Controller
Bit 3: 0 = Not SIS; 1 = Controller is SIS
Bit 4-7: reserved

Timer functions supported comprises of TimerStart, TimerRestart, and TimerCancel.

‘29’ or ‘0’ specifies the length of ‘nodes[]’ array

nodes[29] is a node bitmask. The chip used can be determined as follows:

Z-Wave Chip	Chip_type	Chip_version
ZW0102	0x01	0x02
ZW0201	0x02	0x01
ZW0301	0x03	0x01
ZM0401	0x04	0x01
ZM4102	0x04	0x01
SD3402	0x04	0x01
ZW050x	0x05	0x00
ZGM130S	0x07	0x00
ZG14	0x07	0x00
ZG23	0x08	0x00
ZGM230S	0x08	0x00

7.4.2 Get Long Range Nodes

In Z-Wave Long Range network, function FUNC_ID_SERIAL_API_GET_LR_NODES is used to obtain the list of Long Range nodes:

HOST->ZW: REQ | 0xDA | BITMASK_OFFSET

ZW->HOST: RES | 0xDA | MORE_NODES | BITMASK_OFFSET | BITMASK_LEN | BITMASK_ARRAY

MORE_NODES – byte, has value 1 if more nodes is available and the host can request the next chunk of the nodes bitmask array. Currently it always returns 0 as max supported number of LR nodes can fit in 128 bytes.

BITMASK_OFFSET - 16 bit value. Supported values: 0, 1, 2, 3 that corresponds to offset 0, 128, 256, 384. If BITMASK_OFFSET is not one of the values mentioned above it will be round down to the nearest value. Currently values higher than 0 won’t contain any nodes.

BITMASK_LEN – byte, hardcoded to 128 bytes.

BITMASK_ARRAY is an array of 128 bytes, with least significant bytes first, contains a bitmask of the available nodes in the network.

If bit **N** in byte **J** (J >= 0) is set to 1, then node with ID = **BASE + 8*J + N + BITMASK_OFFET**, where BASE = 256, exists in network.

7.5 Set Timeouts Command

The timeout in the Serial API (starting with the Serial API version 4) can be set in 10 ms ticks by using the FUNC_ID_SERIAL_API_SET_TIMEOUTS Serial API function:

Serial API:

HOST->ZW: REQ | 0x06 | RXACKtimeout | RXBYTEtimeout

ZW->HOST: RES | 0x06 | oldRXACKtimeout | oldRXBYTEtimeout

7.6 Set up ZW_SendData Callback Parameters

The callback parameter list extension (starting with the Serial API version 6) can be controlled by using FUNC_ID_SERIAL_API_SETUP Serial API function:

Serial API:

HOST->ZW: REQ | 0x0B | 0x02 | enableTxStatusReport

ZW->HOST: RES | 0x0B | 0x02 | CmdRes[]

enableTxStatusReport = 0, No extra parameters can be transmitted on callback

enableTxStatusReport = 1, Extra parameters should be transmitted on callback (Default)

Must be called after reset if none Default setting is required.

7.7 Supported SERIAL_API_SETUP_CMD commands

All supported SERIAL_API_SETUP commands can be obtained by using SERIAL_API_SETUP_CMD_SUPPORTED subcommand of SERIAL_API_SETUP_CMD

Host -> ZW:

<table>
<tr><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="8">FUNC_ID_SERIAL_API_SETUP</td></tr>
<tr><td colspan="8">SERIAL_API_SETUP_CMD_ SUPPORTED</td></tr>
</table>

ZW -> Host:

7	6	5	4	3	2	1	0
FUNC_ID_SERIAL_API_SETUP							
SERIAL_API_SETUP_CMD_ SUPPORTED							
Supported flags							
Supported bitmask 0							
Supported bitmask 1							
…							
Supported bitmask 16							

Supported flags

SERIAL_API_SETUP_CMD supported commands besides SERIAL_API_SETUP_CMD_SUPPORTED, represented as bitmask flag.
Example: SERIAL_API_SETUP_CMD_TX_POWERLEVEL_SET | SERIAL_API_SETUP_CMD_TX_POWERLEVEL_GET | …
Includes only commands with values 2^N. Newer commands that don't have such value are reported in **Supported Bitmask**

Supported bitmask

Array of bytes including all supported commands, represented as bitmask of values, with least significant bytes first.
Example: (1 << SERIAL_API_SETUP_CMD_SUPPORTED) | (1<< SERIAL_API_SETUP_CMD_TX_POWERLEVEL_SET) | (1<< SERIAL_API_SETUP_CMD_TX_POWERLEVEL_GET) | …

7.8 Configuration of Default Tx Power Level

By default, the transmit power level is hard coded in the Z-Wave image downloaded to the Z-Wave chip. However, hardware differences in products may require changing the Tx power. Changing the default Tx power will require the following steps:

1. Use the SERIAL_API_SETUP_CMD_TX_POWERLEVEL_SET command to set the power levels to desired values.
2. Use the FUNC_ID_SERIAL_API_SOFT_RESET command to restart the Z-Wave module so the new settings are activated.

7.8.1 Set Default Tx Power Level

The Transmit power can be configured through Serial API (starting with Serial API version 7) by using FUNC_ID_SERIAL_API_SETUP Serial API function, subfunction SERIAL_API_SETUP_CMD_TX_POWERLEVEL_SET.

The power levels set by this function are first used by the Z-Wave protocol next time the module is restarted.

Host -> ZW:

7	6	5	4	3	2	1	0
FUNC_ID_SERIAL_API_SETUP							
SERIAL_API_SETUP_CMD_TX_POWERLEVEL_SET							
NormalTxPower							
Measured0dBmPower							

NormalTxPower

The power level used when transmitting frames at normal power. The power level is in deci dBm, for example 1 dBm output power will be 10 in NormalTxPower and -2 dBm will be -20 in NormalTxPower.

Measured0dBmPower

The output power measured from the antenna when NormalTxPower is set to 0 dBm. The power level is in deci dBm, for example 1d Bm output power will be 10 in Measured0dBmPower and -2 dBm will be -20 in Measured0dBmPower.

ZW->HOST:

7	6	5	4	3	2	1	0
FUNC_ID_SERIAL_API_SETUP							
SERIAL_API_SETUP_CMD_TX_POWERLEVEL_SET							
CmdRes							

CmdRes

Result of the command

CmdRes = 0 – Power levels was not set.

CmdRes = 1 – Power levels was set.

7.8.2 Get Default Tx Power Level

The Transmit power can be read through the serial API (starting with Serial API version 7) by using FUNC_ID_SERIAL_API_SETUP Serial API function, subfunction SERIAL_API_SETUP_CMD_TX_POWERLEVEL_GET:

HOST->ZW:

7	6	5	4	3	2	1	0
FUNC_ID_SERIAL_API_SETUP							
SERIAL_API_SETUP_CMD_TX_POWERLEVEL_GET							

ZW->HOST:

7	6	5	4	3	2	1	0
FUNC_ID_SERIAL_API_SETUP							
SERIAL_API_SETUP_CMD_TX_POWERLEVEL_GET							
NormalTxPower							
Measured0dBmPower							

NormalTxPower

The power level used when transmitting frames at normal power. The power level is in deci dBm, for example 1 dBm output power will be 10 in NormalTxPower and -2 dBm will be -20 in NormalTxPower

Measured0dBmPower

The output power measured from the antenna when NormalTxPower is set to 0 dBm. The power level is in deci dBm, for example 1 dBm output power will be 10 in Measured0dBmPower and -2 dBm will be -20 in Measured0dBmPower.

7.9 Get the Background RSSI Levels for each channel

The command FUNC_ID_ZW_GET_BACKGROUND_RSSI returns the Background RSSI level for each valid channel. The command always returns four RSSI values expressed in dBm (8bit each): only the ones corresponding to valid channels contain the measured Background RSSI levels, the remaining are set to 0x7F (i.e., 127 dBm). The valid background RSSI levels range from -105 dBm and +30 dBm.

The number of valid channels depends on the region, the device type (controller/end device) and whether the end device node is included or not. In the following table there is a summary of the possible configurations. The four Background RSSI levels have assigned an index from 0 to 3 to identify them.

Region	Device Type	Included (for End-devices)	Valid Background RSSI levels (index)
2-channels	Controller/ End-device	-	0 (100kbps), 1 (9.6kbps), 2 (40kbps)
3-channels	Controller/ End-device	-	0 (100kbps), 1 (100kbps), 2 (100kbps)
4-channels (US_LR with ZW_LR_CHANNEL_A as 4th channel)	Controller	-	0 (100kbps), 1 (9.6kbps), 2 (40kbps), 3 (ZW_LR_CHANNEL_A)
	End-device	No	0 (100kbps), 1 (9.6kbps), 2 (40kbps), 3 (ZW_LR_CHANNEL_A)
	End-device	Yes	0 (ZW_LR_CHANNEL_A), 1 (ZW_LR_CHANNEL_B)
4-channel (US_LR with ZW_LR_CHANNEL_B as 4th channel)	Controller	-	0 (100kbps), 1 (9.6kbps), 2 (40kbps), 3 (ZW_LR_CHANNEL_B)
	End-device	No	0 (100kbps), 1 (9.6kbps), 2 (40kbps), 3 (ZW_LR_CHANNEL_B)
	End-device	Yes	0 (ZW_LR_CHANNEL_A), 1 (ZW_LR_CHANNEL_B)

Host -> ZW:

7	6	5	4	3	2	1	0
FUNC_ID_ZW_GET_BACKGROUND_RSSI							

ZW -> Host:

7	6	5	4	3	2	1	0
FUNC_ID_ZW_GET_BACKGROUND_RSSI							
RSSI[0]							
RSSI[1]							
RSSI[2]							
RSSI[3]							

RSSI:

The function returns the Background RSSI levels (8-bit signed values) for the valid channels, 0x7F otherwise (see Table above).

7.10 Configuration of the RF Region Setting

7.10.1 Configuration Any Time

To change the RF Region setting at any time, use the sequence of commands below:

1. Use the SERIAL_API_SETUP_CMD_RF_REGION_SET command to set the RF Region setting to the new value.
2. Use the FUNC_ID_SERIAL_API_SOFT_RESET command to restart the Z-Wave module so the new setting gets activated.

7.10.2 Set RF Region

The RF Region setting can be configured through serial API (starting with the Serial API version 8) by using FUNC_ID_SERIAL_API_SETUP Serial API function, sub-function SERIAL_API_SETUP_CMD_RF_REGION_SET.

The RF Region set by this function is first used by the Z-Wave protocol next time the module is restarted.

Host -> ZW:

7	6	5	4	3	2	1	0
FUNC_ID_SERIAL_API_SETUP							

SERIAL_API_SETUP_CMD_RF_REGION_SET
RfRegion

RfRegion

The RF Region value to be set.

RF Region	Value
Region EU	0x00
Region US	0x01
Region Australia/New Zealand	0x02
Region Hong Kong	0x03
Region Malaysia	0x04
Region India	0x05
Region Israel	0x06
Region Russia	0x07
Region China	0x08
Region US (Z-Wave & Z-Wave Long Range)	0x09
Region Japan	0x20
Region Korea	0x21

ZW->HOST:

7	6	5	4	3	2	1	0
FUNC_ID_SERIAL_API_SETUP							
SERIAL_API_SETUP_CMD_RF_REGION_SET							
CmdRes							

CmdRes

Result of the command

CmdRes = 0 – RF Region was **not** set (invalid value specified or error setting the value in firmware).

CmdRes = 1 – RF Region was successfully set.

7.10.3 Get RF Region

The RF Region setting can be read through serial API (starting with the Serial API version 8) by using FUNC_ID_SERIAL_API_SETUP Serial API function, sub-function SERIAL_API_SETUP_CMD_RF_REGION_GET:

HOST->ZW:

7	6	5	4	3	2	1	0
FUNC_ID_SERIAL_API_SETUP							
SERIAL_API_SETUP_CMD_RF_REGION_GET							

ZW->HOST:

7	6	5	4	3	2	1	0
FUNC_ID_SERIAL_API_SETUP							
SERIAL_API_SETUP_CMD_TX_POWERLEVEL_GET							
RfRegion							

RfRegion

The returned RF Region value

See the SERIAL_API_SETUP_CMD_RF_REGION_SET command (p. 36) for details about the valid RF Region values.

RfRegion = 0xFE – Error retrieving the RF Region value

7.11 DCDC Configuration Commands

The current DCDC configuration can be updated or retrieved using Set DCDC Configuration and Get DCDC Configuration Commands, respectively.

7.11.1 Set DCDC Configuration Command

The host CPU system can set the DCDC Configuration by using the Serial API function FUNC_ID_SET_DCDC_CONFIG (0xDF).

HOST->ZW:

7	6	5	4	3	2	1	0
FUNC_ID_SET_DCDC_CONFIG							
DCDC Configuration							

DCDC Configuration (8 bit):

Value identifying one of the three possible setups for the DCDC Configuration

DCDC Configuration	Value
EDCDCMODE_AUTO	0x00
EDCDCMODE_BYPASS	0x01
EDCDCMODE_DCDC_LOW_NOISE	0x02

ZW->HOST:

7	6	5	4	3	2	1	0
FUNC_ID_SET_DCDC_CONFIG							
CmdRes							

CmdRes (8 bit):

Possible results of the command:

CmdRes	Value
Set DCDC Configuration not successful	0x00
Set DCDC Configuration successful	0x01

7.11.2 Get DCDC Configuration Command

The host CPU system can get the current DCDC Configuration by using the Serial API function FUNC_ID_GET_DCDC_CONFIG (0xDE).

HOST->ZW:

7	6	5	4	3	2	1	0
FUNC_ID_GET_DCDC_CONFIG							

ZW->HOST:

7	6	5	4	3	2	1	0
FUNC_ID_GET_DCDC_CONFIG							
DCDC Configuration							

DCDC Configuration (8 bit):

Value identifying one of the three possible setups for the DCDC Configuration:

DCDC Configuration	Value
EDCDCMODE_AUTO	0x00
EDCDCMODE_BYPASS	0x01
EDCDCMODE_DCDC_LOW_NOISE	0x02

7.12 Ready Command

The Ready Command is used by the host to inform the Z-Wave module that it is ready to receive a command on the UART.

7	6	5	4	3	2	1	0
FUNC_ID_SERIAL_API_READY							
[SerialLinkState]							

SerialLinkState (8 bit):

Set the Serial link state between HOST and the Serial API Z-Wave module.

SERIAL_LINK_DETACHED – The Serial link state should be DETACHED, or Serial API stops sending data to the HOST until either READY is transmitted again in connected state or any valid Serial API command is received from the HOST.

SERIAL_LINK_CONNECTED – The Serial link state should be CONNECTED, or Serial API sends data to the HOST when needed.

The Serial API Z-Wave module starts up after reset in the Serial link state DETACHED.

SerialLinkState define	Value
SERIAL_LINK_DETACHED	0x00
SERIAL_LINK_CONNECTED	0x01

7.13 Get Maximum Payload Size

The maximum supported payload size can be read through serial API (starting with the Serial API version 7) by using FUNC_ID_SERIAL_API_SETUP Serial API function, sub function SERIAL_API_SETUP_CMD_TX_GET_MAX_PAYLOAD_SIZE:

HOST->ZW:

7	6	5	4	3	2	1	0
FUNC_ID_SERIAL_API_SETUP							
SERIAL_API_SETUP_CMD_TX_GET_MAX_PAYLOAD_SIZE							

ZW->HOST:

7	6	5	4	3	2	1	0
FUNC_ID_SERIAL_API_SETUP							
SERIAL_API_SETUP_CMD_TX_GET_MAX_PAYLOAD_SIZE							
MaxPayloadSize							

MaxPayloadSize

Maximum payload size supported by the Z-Wave protocol.

7.14 Set Node ID Base Type

Command to set the *Base Type* of all Serial API Command **Node ID** fields. Introduced in Serial API version 9. The Node ID *Base Type* defines how all Serial API Command **Node ID** fields should be interpreted. The setting can be either **8** or **16 bits**.

The **8 bits** setting is the default (legacy) setting where the **Node ID** field is 1 byte wide as illustrated in the command frame below:

```
| Byte 1 | Byte 2 | Byte3 | Byte 4 | Byte 5 | ...
|  SOF   | Length | Type  |  Cmd   | NodeID | ...
```

The **16 bits** setting means the **Node ID** field is 2 bytes wide, with the most significant byte (MSB) first, as illustrated in the command frame below:

```
| Byte 1 | Byte 2 | Byte3 | Byte 4 |   Byte 5   |   Byte 6   | ...
|  SOF   | Length | Type  |  Cmd   | NodeID MSB | NodeID LSB | ...
```

Notice: The command is not persistent. Must be re-issued after a reset or power-cycle of the Serial API Controller. I.e. the Host should subscribe to the **Serial API started Command** [7.16] to be notified of any Controller restart and re-issue the command accordingly.

Host -> ZW:

7	6	5	4	3	2	1	0
FUNC_ID_SERIAL_API_SETUP							
SERIAL_API_SETUP_CMD_NODEID_BASETYPE_SET							
BaseType							

BaseType

The Node ID Base Type value to be set.

BaseType	Value
8 bits	0x01
16 bits	0x02

ZW->HOST:

7	6	5	4	3	2	1	0
FUNC_ID_SERIAL_API_SETUP							
SERIAL_API_SETUP_CMD_NODEID_BASETYPE_SET							
CmdRes							

CmdRes

Result of the command

CmdRes = 0 – Command Error. The Node ID Base Type is set to default value (8 bit).

CmdRes = 1 – Command OK. Requested Node ID Base Type successfully set.

7.15 Ready Command

The Ready Command is used by the host to inform the Z-Wave module that it is ready to receive a command on the UART.

7	6	5	4	3	2	1	0
FUNC_ID_SERIAL_API_READY							
[SerialLinkState]							

SerialLinkState (8 bit):

Set the Serial link state between HOST and the Serial API Z-Wave module.

SERIAL_LINK_DETACHED – The Serial link state should be DETACHED, or Serial API stops sending data to the HOST until either READY is transmitted again in connected state or any valid Serial API command is received from the HOST.

SERIAL_LINK_CONNECTED – The Serial link state should be CONNECTED, or Serial API sends data to the HOST when needed.

The Serial API Z-Wave module starts up after reset in the Serial link state DETACHED.

SerialLinkState define	Value
SERIAL_LINK_DETACHED	0x00
SERIAL_LINK_CONNECTED	0x01

7.16 Serial API started Command

The Serial API will inform the host that is has been started by issuing the FUNC_ID_SERIAL_API_STARTED command.

ZW->HOST:

7	6	5	4	3	2	1	0
FUNC_ID_SERIAL_API_STARTED							
WakeupReason							
WatchdogStarted							
deviceOptionMask							
GenericNodetype							
SpecificNodetype							
CommandClassLength							
CommandClass 1							
…							
CommandClass x							
Capabilities							

WakeupReason

The reason for starting up the Z-Wave module.

SerialLinkState define	Description	Value
ZW_WAKEUP_RESET	Module was reset	0x00
ZW_WAKEUP_WUT	Module was started by a wake up timer	0x01
ZW_WAKEUP_SENSOR	Module was started because it received a wakeup beam	0x02
ZW_WAKEUP_WATCHDOG	Module was reset by the watchdog timer	0x03
ZW_WAKEUP_EXT_INT	Module was started by external interrupt	0x04
ZW_WAKEUP_POR	Module was reset by loss of power	0x05

WatchdogStarted

0 – Watchdog timer is not started.

1 – Watchdog timer is started and kicked by the Serial API.

deviceOptionMask

The deviceoptionmask set by the SerialAPI_ApplicationNodeInformation command

GenericNodetype

The generic node typ set by the SerialAPI_ApplicationNodeInformation command

SpecificNodetype

The specific node type set by the SerialAPI_ApplicationNodeInformation command

CommandClassLength

The number of command classes in the Node information frame

CommandClass x

The command class number supported by the node

Capabilities

Bitfield with information of supported Serial API Controller features

Value	Description
Bit 0	Controller is Z-Wave Long Range capable
Bit 1 – Bit 7	Unused

7.17 Softreset Command

The host CPU system can make a software reset of the Z-Wave module by using the Softreset Command.

7	6	5	4	3	2	1	0
FUNC_ID_SERIAL_API_SOFT_RESET							

Wait 1.5 seconds after reset to ensure that the module is ready for communication again.

Note: USB modules will disconnect - connect when this command is issued, which means that the module may get a new address on the USB bus. This will make the old file handle to the USB serial interface invalid.

7.18 Watchdog Commands

Some PC-based applications cannot guarantee kicking the watchdog before timeout causing the Watchdog to reset the Z-Wave ASIC unintentionally. The following Watchdog Commands are therefore available to avoid this:

- Stop Watchdog: Disable Watchdog and stop kick Watchdog in ApplicationPoll

Watchdog handling disabled when powered up and Sleep/FLiRS mode will temporary stop Watchdog.

The host CPU system can start Watchdog functionality by using the Serial API function FUNC_ID_ZW_WATCHDOG_START:

7	6	5	4	3	2	1	0
FUNC_ID_ZW_WATCHDOG_START							

The host CPU system can stop Watchdog functionality by using the Serial API function FUNC_ID_ZW_WATCHDOG_STOP:

7	6	5	4	3	2	1	0
FUNC_ID_ZW_WATCHDOG_STOP							

7.19 NVM Backup and Restore

The host processor can make a backup or a restore of the Non-Volatile Memory (NVM) in the Z-Wave chip using the serial API.

NOTE: Only supported by the 500 series systems.

There is one command for doing both backup and restore.

Host -> ZW:

7	6	5	4	3	2	1	0
FUNC_ID_NVM_BACKUP_RESTORE							
Operation							
Length							
Offset MSB							
Offset LSB							
Buffer[0]							
…							
..							
Buffer[x]							

Operation (8bit):

The operation to be executed:

Operation	Value
Open	0x00
Read	0x01
Write	0x02
Close	0x03

Read, Write, and Close operations are only valid after an Open operation has been executed.

Length (8bit):

A desired length of the read/write buffer

Offset (16bit)

An offset in the NVM where the write or read should be done.

Buffer (8bit*x):

The write buffer containing the data that should be written to NVM when restoring NVM.

ZW -> Host:

7	6	5	4	3	2	1	0
FUNC_ID_NVM_BACKUP_RESTORE							
Return Value							
Length							
Offset MSB							
Offset LSB							
Buffer[0]							
…							
..							
Buffer[x]							

Return Value (8bit):

The result of the requested operation.

Return Value	Value
Ok	0x00
Error	0x01
ErrorOperationMismatch (Error mixing read and write)	0x02
ErrorOperationDisturbed (Error read operation disturbed by another write)	0x03
End Of File	0xFF

Length (8bit):

An actual length of the read/write buffer.

Offset (16bit)

An offset in the NVM where the write or read was done.

Buffer (8bit*x):

The read buffer containing the data that was read from NVM when backing up NVM.

7.19.1 Backing up NVM

The backup and restore function is session-based because the Z-Wave protocol limits the access to the NVM while the backup and restore is being done. The host application should stop all other activity on the serial API while the backup is being done.

The correct sequence of commands for initiating a backup is the following:

FUNC_ID_NVM_BACKUP_RESTORE (open)

Returns the backup size.

FUNC_ID_NVM_BACKUP_RESTORE (read, read, .)

Returns EOF if no more data or error if the backup is disturbed by other writes to the NVM.

FUNC_ID_NVM_BACKUP_RESTORE (close)

Returns an error if backup was disturbed by other writes. Ok is returned if the backup was done without any writes to the NVM.

If an error was returned, discard backed up data and try again.

7.19.2 Restoring NVM

Restoring the NVM in the Z-Wave protocol requires a few more steps than the backup because the host needs to ensure that all old NVM data is deleted and that the new NVM is taken in use.

The correct sequence of commands for restoring NVM is the following:

FUNC_ID_ZW_SET_DEFAULT

Deletes all old NVM content.

FUNC_ID_NVM_BACKUP_RESTORE (open)

Returns an unused value.

FUNC_ID_NVM_BACKUP_RESTORE (write, write, ...)

Writes the whole NVM image to the NVM in the Z-Wave module.

FUNC_ID_NVM_BACKUP_RESTORE (close)

Returns an unused value.

FUNC_ID_SERIAL_API_SOFT_RESET

Activates the Z-Wave module with the new NVM image.

Note that while the restore is taking place, the node will not be part of the network so all Z-Wave communication to the node will fail.

7.20 Restrictions on Functions Using Buffers

The Serial API is implemented with buffers for queuing requests and responses. This restricts how much data that can be transferred through MemoryGetBuffer() and MemoryPutBuffer() compared to using them directly from the Z-Wave API.

The PC application should not try to get or put buffers larger than approximately 80 bytes.

If an application requests too much data through MemoryGetBuffer(), the buffer will be truncated, and the application will not be notified.

If an application tries to store too much data with MemoryPutBuffer(), the buffer will be truncated before the data is sent to the Z-Wave module, again without the application being notified.

7.21 Configuration of Z-Wave Long Range channel.

There are 2 rf-channels available for Z-Wave Long Range communication. A controller can only use one frequency at a time. The host can use the commands below to get and set the active Long Range channel.

7.21.1 Get active Long Range channel

Command to get the active Long Range rf-channel. Introduced in Serial API version 9.

HOST->ZW:

7	6	5	4	3	2	1	0
FUNC_ID_GET_LR_CHANNEL							

ZW->HOST:

7	6	5	4	3	2	1	0
FUNC_ID_GET_LR_CHANNEL							
Channel							

Channel

The rf-channel that the controller uses for Long Range communication.

Channel	Value
ZW_LR_CHANNEL_A	0x01
ZW_LR_CHANNEL_B	0x02

7.21.2 Set active Long Range channel

Command to set the active Long Range rf-channel. Introduced in Serial API version 9.

Host -> ZW:

7	6	5	4	3	2	1	0
FUNC_ID_SET_LR_CHANNEL							
Channel							

Channel

The rf-channel to be set.

Channel	Value
ZW_LR_CHANNEL_A	0x01
ZW_LR_CHANNEL_B	0x02

7.22 Configuration of Long Range virtual node IDs

Four Long Range node IDs are reserved for virtual nodes. IDs: 4002, 4003, 4004 and 4005. By default, all frames with virtual node IDs are rejected by Z-wave controllers. To accept application level frames with a virtual node ID, that node ID must be enabled.

Command to enable virtual node IDs. Introduced in Serial API version 9.

Notice: The command is not persistent. Must be re-issued after a reset or power-cycle of the Serial API Controller. I.e. the Host should subscribe to the **Serial API started Command** [7.16] to be notified of any Controller restart and re-issue the command accordingly.

Host -> ZW:

7	6	5	4	3	2	1	0
FUNC_ID_ZW_SET_LR_VIRTUAL_IDS							
NodeIdBitmask							

NodeIdBitmask

Setting bits to 1 will enable node IDs. Setting bits to 0 will disable.

NodeIdBitmask	bit
Ignored	b4-b7
Enable node ID: 4005	b3
Enable node ID: 4004	b2
Enable node ID: 4003	b1
Enable node ID: 4002	b0

7.23 ZW_SendData Function

FUNC_ID_ZW_SEND_DATA:

HOST->ZW: nodeID | dataLength | pData[] | txOptions | funcID

ZW->HOST: RetVal

RetVal == false -> no callback

RetVal == true then callback returns with

ZW->HOST: txStatus | wTransmitTicksMSB | wTransmitTicksLSB | bRepeaters | rssi_values.incoming[0] | rssi_values.incoming[1] | rssi_values.incoming[2] | rssi_values.incoming[3] | rssi_values.incoming[4] | bRouteSchemeState | repeater0 | repeater1 | repeater2 | repeater3 | routespeed | bRouteTries | bLastFailedLink.from | bLastFailedLink.to | bUsedTxpower | bMeasuredNoiseFloor | bAckDestinationUsedTxPower | bDestinationAckMeasuredRSSI | bDestinationckMeasuredNoiseFloor

Fields

- bUsedTxpower
- bMeasuredNoiseFloor
- bAckDestinationUsedTxPower
- bDestinationAckMeasuredRSSI
- bDestinationckMeasuredNoiseFloor

Are applicable for Z-Wave Long Range Network only. Otherwise they are set to RSSI_NOT_AVAILABLE

7.24 ApplicationCommandHandler_Bridge

Function ApplicationCommandHandler_Bridge is triggered after SerialAPI receives the frame

On Long Range Network:

ZW->HOST: REQ | 0xA8 | rxStatus | destNode | sourceNode | cmdLength | pCmd[] | multiDestsOffset_NodeMaskLen | multiDestsNodeMask[] | rssiVal | securityKey | bSourceTxPower | bSourceNoiseFloor

Fields:

- bSourceTxPower
- bSourceNoiseFloor

are applicable for Z-Wave Long Range Network only. Otherwise they are set to RSSI_NOT_AVAILABLE

7.25 Enable PTI Zniffer functionality.

It is possible to enable/disable PTI Zniffer functionality for the 700 SoC as a startup option on SerialAPIControllers. This means that the nodes keep functioning as a normal SerialAPIControllers but in addition also provide Zniffer info via the Ethernet ports on the BRD4001A boards. PTI uses the ZG14 pins #21 and #20, which correspond to PB13 (FRC_DRAME) and PB12 (FRC_DOUT). Other pin configurations are not supported currently. PTI functionality is disabled by default.

7.25.1 Enable/disable PTI Zniffer

Please note that a node must be **soft reset (7.17)** after the command is sent to activate the setting.

Host -> ZW:

7	6	5	4	3	2	1	0
FUNC_ID_ENABLE_RADIO_PTI							
Enable (0x01) / Disable (0x00)							

The node answers the host to verify the setting:

ZW -> Host:

7	6	5	4	3	2	1	0
FUNC_ID_ENABLE_RADIO_PTI							
OK (0x01) / Failure (0x00)							

7.25.2 Get Radio PTI state

To get if the PTI functionality is currently enabled or not.

Host -> ZW:

7	6	5	4	3	2	1	0
FUNC_ID_GET_RADIO_PTI							

The node answers the host to verify the setting:

ZW -> Host:

7	6	5	4	3	2	1	0
FUNC_ID_GET_RADIO_PTI							
Enabled (0x01) / Disabled (0x00)							

APPENDIX ASERIAL API FILES

The Serial API embedded sample code is provided in the Z-Wave Developer's Kit until SDK 6.81.0x. Only binaries are distributed starting with the SDK 7.00.00+. Note that altering the function IDs and frame formats in the Serial API embedded sample code can result in interoperability problems with the Z-Wave DLL supplied on the Developer's Kit as well as commercially available GUI applications. To determine the current version of the Serial API protocol in the embedded sample code, see the API call **ZW_Version**.

The ProductPlus\SerialAPIPlus directory contains sample source code for controller/end device applications on a Z-Wave module. The application also uses several utility functions described in [2], depending on the SDK used.

Appendix A.1 Makefiles

MK.BAT

Make bat file for building the sample application in question. To only build applications using EU frequency enter: **MK "FREQUENCY=EU"** in the command prompt.

Makefile

This is the Makefile for the sample application in question defining the targets built. See [2] for additional details depending on SDK used.

Makefile.common_ZW0x0x_supported_functions

This makefile makes a text file showing the supported serial API functions for the given target.

Appendix A.2 Application

app_version.h

This header file contains defines for the application version.

config_app.h

This header file contains defines for Manufacturer-Specific Command Class and defines for Security settings.

conhandle.h / conhandle.c

Routines for handling Serial API protocol between PC and Z-Wave module.

eeprom.h / eeprom.c

NVM layout.

make-supported-functions-include.bat

Windows batch script for generating Serial API defines for supported functions based on what exists in the library.

Prodtest_vars.c

Critical memory variables used for a production test.

serialapi-supported-func-list.txt

Template file for generating SerialAPI defines for supported functions based on what exists in the library. Enable/disable support of a given Serial API function in serialappl.h header file.

serialappl.h / serialappl.c

This module implements the handling of the Serial API protocol, which involves parsing the frames, calling the appropriate Z-Wave API library functions, returning results, and so on to the PC. Enable/disable support of a given Serial API function in serialappl.h header file.

Supported.bat

Batch file called by Makefile.common_ZW0x0x_supported_function to obtain delayed environment variable expansion when using SET in DOS prompt.

REFERENCES

[1] IETF RFC 2119, Key words for use in RFCs to Indicate Requirement Levels, http://tools.ietf.org/pdf/rfc2119.pdf

[2] SL, INS13933, Instruction, Z-Wave 500 Series SDK Contents v6.81.0x.

[3] SL, INS13954, Instruction, Z-Wave 500 Series Appl. Prg. Guide v6.8x.0x.SL, INS14259, Instruction, Z-Wave Plus V2 Application Framework SDK7.

INDEX

Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support & Community
www.silabs.com/community

SILICON LABS

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

www.silabs.com