Instruction

Z-Wave ZW0201/ZW0301 Appl. Prg. Guide v4.55.00

Document No.:	INS11095
Version:	18
Description:	Guideline for developing ZW0201 and ZW0301 based applications using the application programming interface (API) based on Developer's Kit v4.5x
Written By:	JFR;JSI;PSH;DDA;JBU;AES;ABR
Date:	2013-04-16
Reviewed By:	JFR;JSI;PSH;JKA;BBR
Restrictions:	Partners Only

Approved by:				
Date	CET	Initials	Name	Justification
2013-04-16	15:44:43	NTJ	Niels Thybo Johansen	

CONFIDENTIAL

REVISION RECORD

Doc. Rev	Date	By	Pages affected	Brief description of changes
1	20080315	JFR	All	Initial draft
1	20080315	JFR	4.4.3.1	Added missing description of hop failures in TxStatus in ZW_SendData API call
1	20080519	JBU	4.10.1	Names of ZW_RequestNewRouteDestinations() callback function parameters corrected
1	20080519	JBU	4.5.1 4.5.18	Clarified ZW_AddNodeToNetwork() and ZW_RemoveNodeFromNetwork () should use NULL callback when *_NODE_STOP commands are given.
1	20080519	JBU	7.1 7.2	Updated figures and sample code to use recommended ZW_AddNodeToNetwork()/ZW_RemoveNodeFromNetwork () _STOP commands.
1	20080624	JSI	4.5.1	Added description of ADD_NODE_OPTION_NETWORK_WIDE option bit
1	20080825	JSI	5.3.3	Added ZW_SendData_Generic, ZW_SendData_Bridge, ZW_SendDataMeta_Generic and ZW_SendDataMeta_Bridge descriptions
1	20080925	JBU	4.2.1.2 4.2.1.3	Static controller libraries without repeater functionality cannot provide help to or forward Lost requests.
1	20080929	JBU	4.4.1.5	Added promiscuous mode and destNode description to function syntax and SerialAPI description.
1	20080929	JBU	4.4.2.11	Added reference to sec 4.4.1.5.
1	20081201	JSI	4.4.1.5 4.4.1.9	Bridge Controller do not use the ApplicationCommandHandler The ApplicationSlaveCommandHandler and the ApplicationCommandHandler in the Bridge Controller has been unified in the ApplicationCommandHandler_Bridge with full Multicast support
1	20081201	PSH	4.5.24 4.9.1	Changed mode parameter for ZW_SetLearnMode()
1	20081204	JSI	4.4.3 4.4.3.7 4.7	Updated parameter descriptions and SerialAPI definitions. Added ZW_SendDataMulti_Bridge descriptions. Removed ZW_SendSlaveData descriptions.
2	20081216	JFR	4.4.3.1	Clarified transmit options behavior
2	20090114	JFR	4.4.2.18	Watchdog disabled by default
2	20090114	PSH	4.4.1 4.4.2.1	Limitations regarding application execution ZW_Poll description updated
2	20090130	JSI	4.4.2.9	Updated serialapi flow description
2	20090201	JFR	4.5.27 4.10.3 4.5.27 & **Error! Reference source not found.**	Do not recommend to call ZW_SetSUCNodeID using low RF power A non-SUC primary static controller does not respond to a rediscovery needed when there is no SUC present in the network ZW_SetRoutingMAX and ZW_GetRoutingMAX added
2	20090219	JFR	4.4.4.1 & 4.4.4.2 4.4.6	TRIAC_Init and TRIAC_SetDimLevel description updated. PWM API updated.
2	20090223	JBU	4.4.3.1	Discouraged sending to a virtual node from the hosting bridge library.
2	20090225	JSI	4.5.23	Added ZW_SetRoutingInfo description.
2	20090304	JSI	4.4.2.1, 4.4.2.4, 4.4.2.8, 4.9.3, 4.10.1 & 4.10.4	Added SerialAPI descriptions.
2	20090305	JFR	4.4.2.8	Using normal power minus 6dB when doing neighbor search
3	20090310	JFR	4.4.3	Payload length must be minimum one byte.
4	20090928	JFR JSI	4.5.27 & **Error! Reference source not found.** 4.5.17 4 3 4.5.11	Serial API impl. of ZW_SetRoutingMAX/ZW_GetRoutingMAX. Added normal/low RF transmission parm. to ZW_ReplaceFailedNode. TRANSMIT_OPTION_RETURN_ROUTE replaced with TRANSMIT_OPTION_AUTO_ROUTE. Details about routing algorithm added to library descriptions. Sensor capabilities documented in Node Information Frame when calling ZW_GetProtocolInfo
5	20100104	JFR	4.10.1	Missing routing slave supported API call ZW_GetSUCNodeID added
6	20100817	JFR	4.4.1	ApplicationRfNotify discontinued
7	20100819	PSH	4.5.1	Added ADD_NODE_STATUS_NOT_PRIMARY status to ZW_AddNodeToNetwork() call

REVISION RECORD

Doc. Rev	Date	By	Pages affected	Brief description of changes
8	20111004	JFR	3.8	Warning against USING 0 attribute in ISR's
9	20111205	JFR	All	Added RU frequency
10	20111215	JFR	4.4.1.9	Clarified Bridge library based serial API application interface
11	20120326	JFR	4.1 4.4.7 4.5.18	Added API using guidelines 200/300 Series onboard flash write cycles Callback description updated for ZW_RemoveNodeFromNetwork/REMOVE_NODE_STOP
13	20120509	JFR	4.4.1.11 4.4.2.14	Added ApplicationRFNotify to support to external power amplifier (PA) Updated ZW_SetSleepMode description wrt. beamCount and POR
13	20120524	JBU JSI	4.5.3 4.4.2.1 4.5.10	Documented TRANSMIT_COMPLETE_NOROUTE callback. Added ZW_ExploreRequestInclusion serial API description. Added ZW_GetLastWorkingRoute description.
14	20120712	JFR	4.5.1, 4.4.2.9, 4.5.25 & 4.9.2	Updated description of ZW_AddNodeToNetwork(), ZW_SetLearnMode() and ZW_SendNodeInformation() to clarify what mode parameters to use.
14	20120822	JBU	4.5.25, 4.9.2	NWI mode not supported for exclusion.
14	20121031	JFR	3.4 4.4.2	Details about routing principles in the Z-Wave protocol added. ZW_Poll removed.
15	20121102	AES	4.4.3.1, 4.5.1, 4.5.18	State diagrams and timeout comments added.
16	20121109	AES ABR PSH	4.5.1	Add node to/from network revised.
17	20121112	AES	4.5.18	Remove node to/from network revised.
18	20130415	JFR	4.5 4.5.27 4.12 4.5.24	Discontinued ZW_GetRoutingMAX Generalize routing attempts in ZW_SetRoutingMAX Added IL frequency Home ID handling by ZW_SetDefault

CONFIDENTIAL

Table of Contents

CONFIDENTIAL

CONFIDENTIAL

CONFIDENTIAL

CONFIDENTIAL

List of Figures

List of Tables

CONFIDENTIAL

CONFIDENTIAL

1   ABBREVIATIONS

Abbreviation	Explanation
ACK	Acknowledge
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
AODV	Ad hoc On-Demand Distance Vector (AODV) Routing
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
DLL	Dynamic Link Library
DUT	Device Under Test
ERTT	Enhanced Reliability Test tool
EU	Europe
GNU	An organization devoted to the creation and support of Open Source software
HK	Hong Kong
HW	Hardware
IN	India
IL	Israel
ISR	Interrupt Service Routines
LRC	Longitudinal Redundancy Check
LWR	Last Working Route
MY	Malaysia
NAK	Not Acknowledged
NVM	Non-Volatile Memory
NWI	Network Wide Inclusion
PA	Power Amplifier
POR	Power On Reset
PRNG	Pseudo-Random Number Generator
PWM	Pulse Width Modulator
RF	Radio Frequency
RU	Russian Federation
SDK	Software Develooper's Kit
SFR	Special Function Registers
SIS	SUC ID Server
SOF	Start Of Frame
SPI	Serial Peripheral Interface
SUC	Static Update Controller
UPnP	Universal Plug and Play
US	United States
WUT	Wake Up Timer
XML	eXtensible Markup Language

2 INTRODUCTION

2.1 Purpose

The purpose of this document is to guide the Z-Wave application programmer through the very first Z-Wave software system build. This programming guide describes the software components, how to build a complete program and load it on a ZW0201/ZW0301 Z-Wave module. The document is also API reference guide for programmers.

2.2 Audience and Prerequisites

The audience is Z-Wave Partners. The application programmer should be familiar with the Keil Development Tool Kit for 8051 micro controllers and the GNU make utility.

2.1 Terms used in this document

The guidelines outlined in IETF RFC 2119 [37] 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.

CONFIDENTIAL

3   Z-WAVE SOFTWARE ARCHITECTURE

Z-Wave software design relies on polling of functions, command complete callback function calls, and delayed function calls.

The software contains two program modules, the Z-Wave basis software and the Application software. The Z-Wave basis software includes system startup code, low-level poll function, main poll loop, Z-Wave protocol layers, and memory and timer service functions. From the Z-Wave basis point of view the Application software include application hardware and software initialization functions, application state machine (called from the Z-Wave main poll loop), command complete callback functions, and a received command handler function. In addition to that, the application software can include hardware drivers.
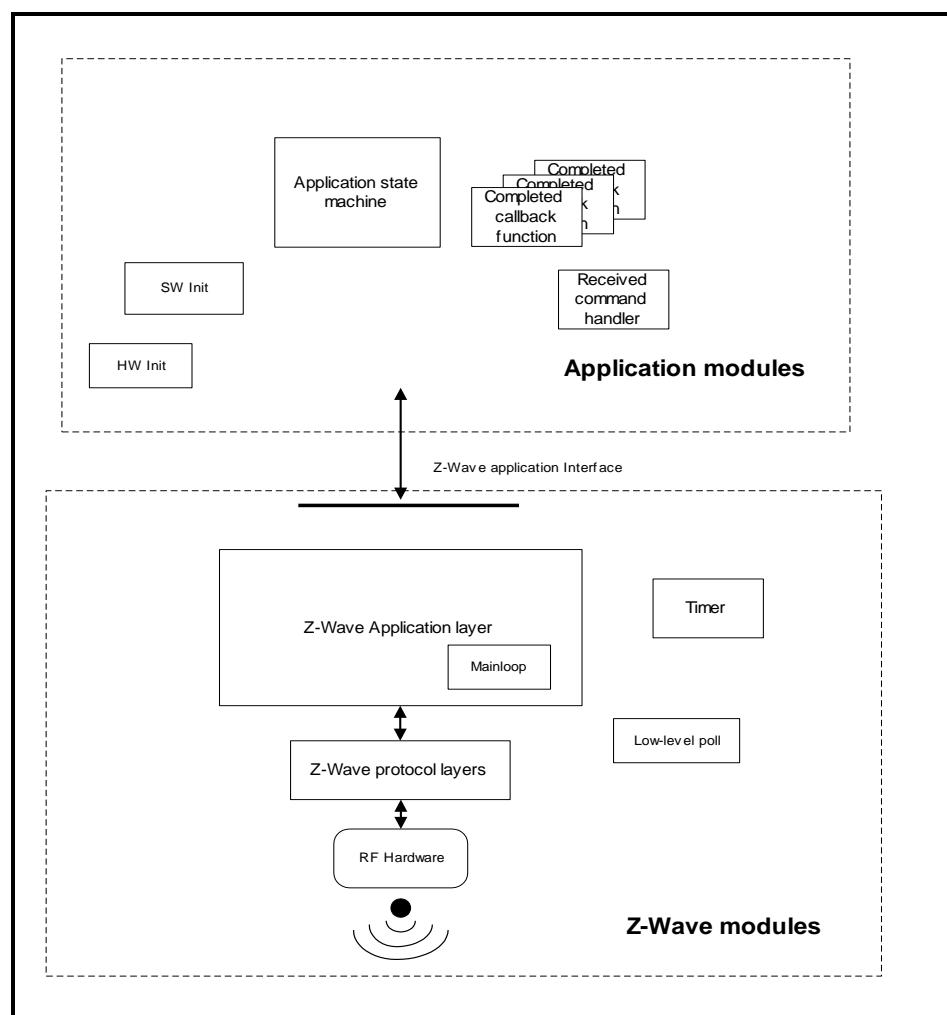
Figure 1. Software architecture

CONFIDENTIAL

3.1   Z-Wave System Startup Code

The Z-Wave modules include the system startup function (main). The Z-Wave system startup function first initializes the Z-Wave hardware and then calls the application hardware initialization function **ApplicationInitHW**. Then the Z-Wave software is initialized (including the software timer used by the timer module) and finally the application software initialization function **ApplicationInitSW** is called. Execution then proceeds in the Z-Wave main loop.

On ZW0201 and ZW0301 are there reserved a small memory area in SRAM that are not initalized by the startup code. These bytes can be used by an application to store information, which should not be cleared by a software reset (or a WUT wakeup). The area is defined by NON_ZERO_START_ADDR and and NON_ZERO_SIZE in the header file ZW_non_zero.h.

3.2   Z-Wave Main Loop

The Z-Wave main loop will call the list of Z-Wave protocol functions, including the **ApplicationPoll** function and the **ApplicationCommandHandler** function (if a frame was received) in round robin order. The functions must therefore be designed to return to the caller as fast as possible to allow the CPU to do other tasks. Busy loops are not allowed. This will make it possible to receive Z-Wave data, transfer data via the UART and check user-activated buttons, etc. "simultaneously". In order not to disrupt the radio communication and the protocol, no application function must execute code for more than 5ms without returning.

For production testing the application can be forced into the **ApplicationTestPoll** function instead of the **ApplicationPoll** function.

3.3   Z-Wave Protocol Layers

When the System layer requests a transmission of data to another node, the Z-Wave protocol layer adds a frame header and a checksum to the data before transmission. The protocol layer also handles frame retransmissions, as well as routing of frames through "repeater" nodes to Z-Wave nodes that are not within direct RF communication reach. When the frame transmission is completed, an application-specified transmit complete callback function is called. The transmission complete callback function includes a parameter that indicates the transmission result. The transmission complete callback function indicate also when the next frame can be send to avoid overwriting the transmit queue.

The Z-Wave frame receiver module (within the MAC layer) can include more than one frame receive buffer, so the upper layers can interpret one frame while the next frame is received.

3.4   Z-Wave Routing Principles

The Z-Wave protocol use source routing, which is a technique whereby the sender of a frame specifies the exact route the frame must take to reach the destination node. Source routing assumes that the sender knows the topology of the network, and can therefore determine a route having a minimum number of hops. The Z-Wave protocol supports up to four repeaters between sender and destination node. Routing can also be used to reach FLiRS destination nodes. Source routing allows implementation of a leightweight protocol by avoiding distributed topologies in all repeaters. Nodes containing the topology can also assign routes to a topology-less node enabling it to communicate with a number of destination nodes using routes.

CONFIDENTIAL

In case sender fails to reach destination node using routes an explorer mechanism can be launched on demand to discover a working route to the destination node in question. The explorer mechanism builds on AODV routing with adjustments for source routing and memory footprint. Explorer frames implement managed multi-hop broadcast forwarding and returns a working route to sender as result. The application payload piggybacks on explorer frame to reduce latency.

The routing algorithm store information about successful attempts to reach a destination node avoiding repetition of previously failed attempts. The last successful route used between sender and destination node are stored in NVM and is called Last Working Route (LWR). The LWR list comprises of 232 destination nodes having up to one route/direct each. The LWR can also contain direct attempts. Updating LWR happens in the following situations:

- When receiving a successful explorer frame route.
- When receiving a successful routed/direct request from another node.
- When receiving a successful acknowledge for a transmitted explorer frame.
- When receiving a successful acknowledge for a transmitted routed/direct frame.

The LWR is removed from the LWR list in case it fails.

The routing attempts depend on the Z-Wave library and transmit options used in the node, for details refer to section 3.9.

The source routing algorithm does not alter the topology due to failed attempts or store any statistics regarding link quality. Only controller-based nodes can return Last Working Route (LWR) via the API call **ZW_GetLastWorkingRoute** enabling application to detect changes in last successful route used.

3.5   Z-Wave Application Layer

The application layer provides the interface to the communications environment which is used by the application process. The application software is located in the hardware initialization function **ApplicationInitHW**, software initialization function **ApplicationInitSW**, application state machine (called from the Z-Wave main poll loop) **ApplicationPoll**, command complete callback functions, and a receive command handler function **ApplicationCommandHandler**.

The application implements communication on application level with other nodes in the network. On application level is a framework defined of Device and Command Classes [1] to obtain interoperability between Z-Wave enabled products from different vendors. The basic structure of these commands provides the capability to set parameters in a node and to request parameters from a node responding with a report containing the requested parameters. The Device and Command Classes are defined in the header file ZW_classcmd.h.

Wireless communication is by nature unreliable because a well defined coverage area simply does not exist since propagation characteristics are dynamic and unpredictable. The Z-Wave protocol minimizes these "noise and distortion" problems by using a transmission mechanisms of the frame there include two re-transmissions to ensure reliable communication. In addition are single casts acknowledged by the receiving node so the application is notified about how the transmission went. All these precautions can unfortunately not prevent that multiple copies of the same frame are passed to the application. Therefore is it very important to implement a robust state machine on application level there can handle multiple copies of the same frame. Below are shown a couple of examples how this can happen:

CONFIDENTIAL
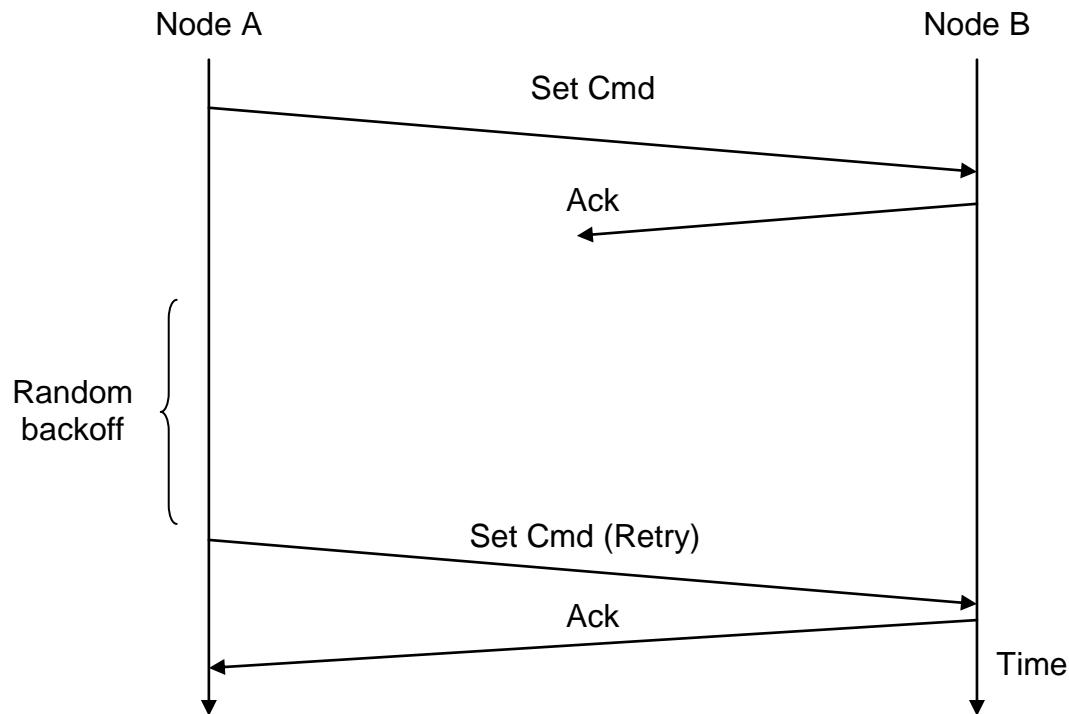
Figure 2. Multiple copies of the same Set frame
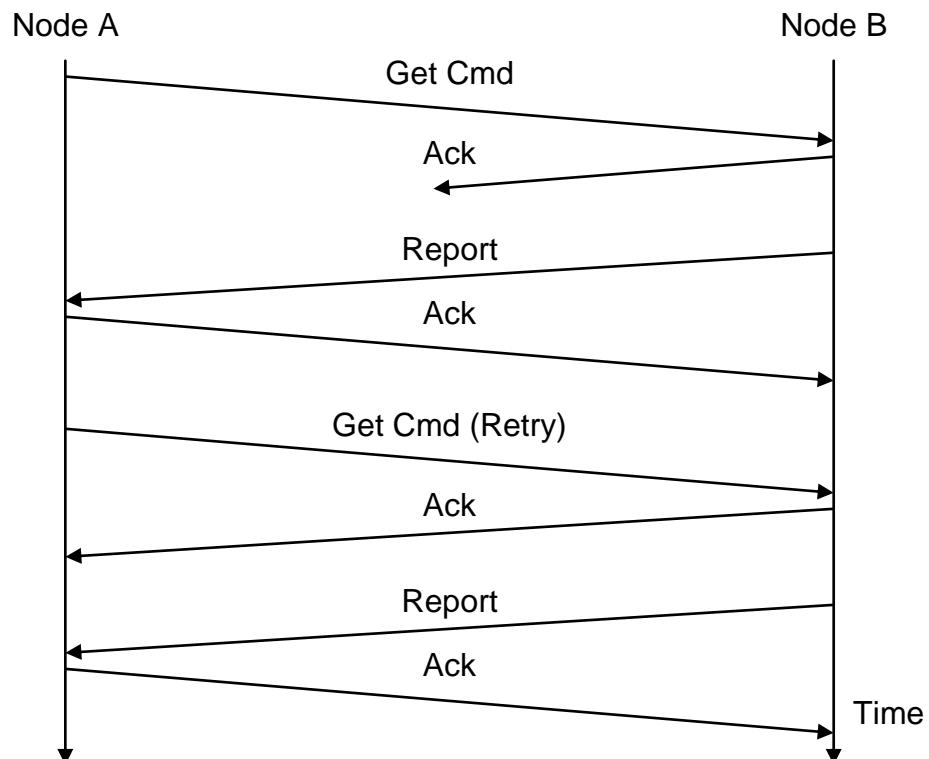
Figure 3. Multiple copies of the same Get/Report frame

A Z-Wave protocol is designed to have low latency on the expense of handling simultaneously communication to a number of nodes in the Z-Wave network. To obtain this is the number of random

backoff values limited to 4 (0, 1 , 2 and 3). The figure below shows how simultaneous communication to even a small number of nodes easily can block the communication completely.
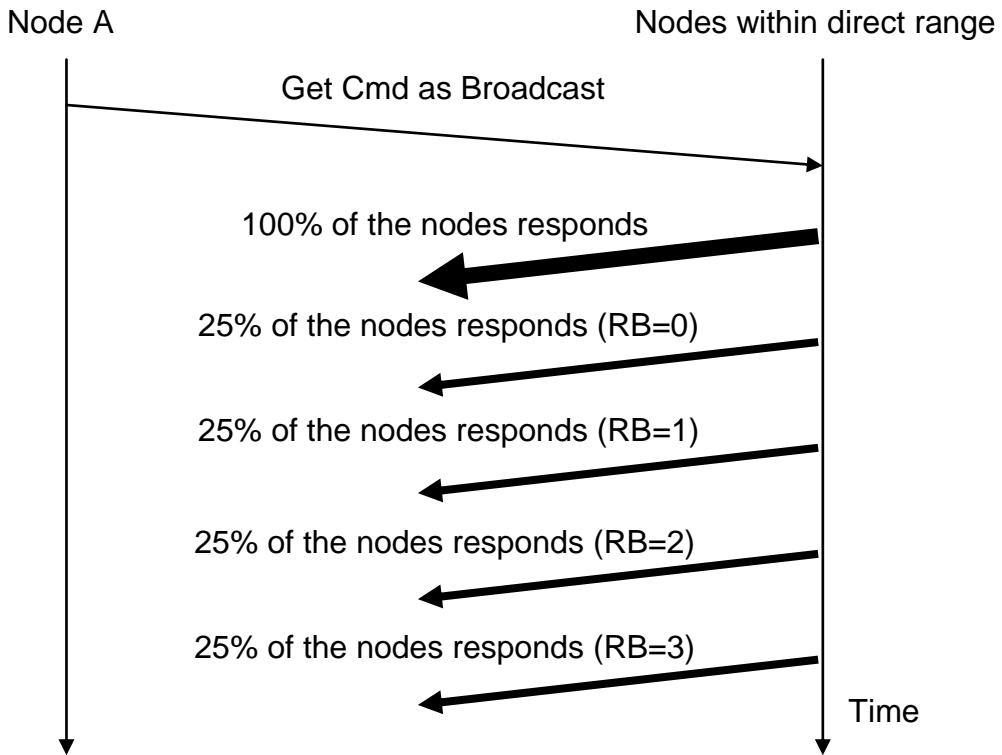
Figure 4. Simultaneous communication to a number of nodes

Avoid simultaneous request to a number of nodes in a Z-Wave network in case the nodes in question respond on the application level.

3.6   Z-Wave Software Timers

The Z-Wave timer module is designed to handle a limited number of simultaneous active software timers. The Z-Wave basis software reserves some of these timers for protocol timeouts.

A delayed function call is initiated by a **TimerStart** API call to the timer module, which saves the function address, sets up the timeout value and returns a timer-handle. The timer-handle can be used to cancel the timeout action e.g. an action completed before the time run out.

The timer can also be used for frequent inspection of special hardware e.g. a keypad. Specifying the time settings to 50 msec and repeating forever will call the timer call back function every 50 msec.

3.7   Z-Wave Hardware Timers

The ZW0102/ZW0201 has a number of hardware timers/counters. Some are reserved by the protocol and others are free to be used by the application as shown in the table below:

Table 1. ZW0102/ZW0201/ZW0301 hardware timer allocation

	ZW0102	ZW0201	ZW0301
TIMER0	Available for the application	Protocol system clock	Protocol system clock
TIMER1	Available for the application in case the UART API is not used	Available for the application	Available for the application
TIMER2	PWM/Timer API	PWM/Timer API	PWM/Timer API
TIMER3	Protocol system clock	Not available	Not available

The TIMER0 and TIMER1 are standard 8051 timers/counters.

3.8   Z-Wave Hardware Interrupts

Application interrupt service routines (ISR) must use 8051 register bank 0. However, do not use USING 0 attribute when declaring ISR's. The Z-Wave protocol uses 8051 register bank 1 for protocol ISR's, see table below regarding application ISR availability:

Table 2. ZW0102/ZW0201/ZW0301 Application ISR avialability

ZW0102	ZW0201	ZW0301
INUM_INT0	INUM_INT0	INUM_INT0
INUM_TIMER0	INUM_INT1	INUM_INT1
INUM_TIMER1	INUM_TIMER1	INUM_TIMER1
INUM_TIMER2	INUM_SERIAL	INUM_SERIAL
INUM_SERIAL0	INUM_SPI	INUM_SPI
INUM_ADC	INUM_TRIAC	INUM_TRIAC
	INUM_GP_TIMER	INUM_GP_TIMER
	INUM_ADC	INUM_ADC

It is not allowed to disable interrupt more than it takes to received 8 bits, which is around 0.8ms at 9.6kbps.

Refer to ZW010x.h ZW020x, and ZW030x.h header files with respect to ISR definitions. For an example, refer to UART ISR in serial API sample application.

CONFIDENTIAL

3.9   Z-Wave Nodes

From a protocol point of view there are seven types of Z-Wave nodes: Portable Controller nodes, Static Controller nodes, Installer Controller nodes, Bridge Controller nodes, Routing Slave nodes and Enhanced Slave nodes. All controller based nodes stores information about other nodes in the Z-Wave network. The node information includes the nodes each of the nodes can communicate with (routing information). The Installation node will present itself as a Controller node, which includes extra functionality to help a professional installer setup, configure and troubleshoot a Z-Wave network. The bridge controller node stores information about the nodes in the Z-Wave network and in addition is it possible to generate up to 128 Virtual Slave nodes.

3.9.1   Z-Wave Portable Controller Node

The software components of a Z-Wave portable controller are split into the controller application and the Z-Wave-Controller basis software, which includes the Z-Wave protocol layers and control of the various data stored into the NVM.

Portable controller nodes include an external EEPROM in which the non-volatile application data area can be placed. The Z-Wave basis software has reserved the first area of the external EEPROM. The physical application memory offset is defined in the header file "ZW_eep_addr.h".

CONFIDENTIAL
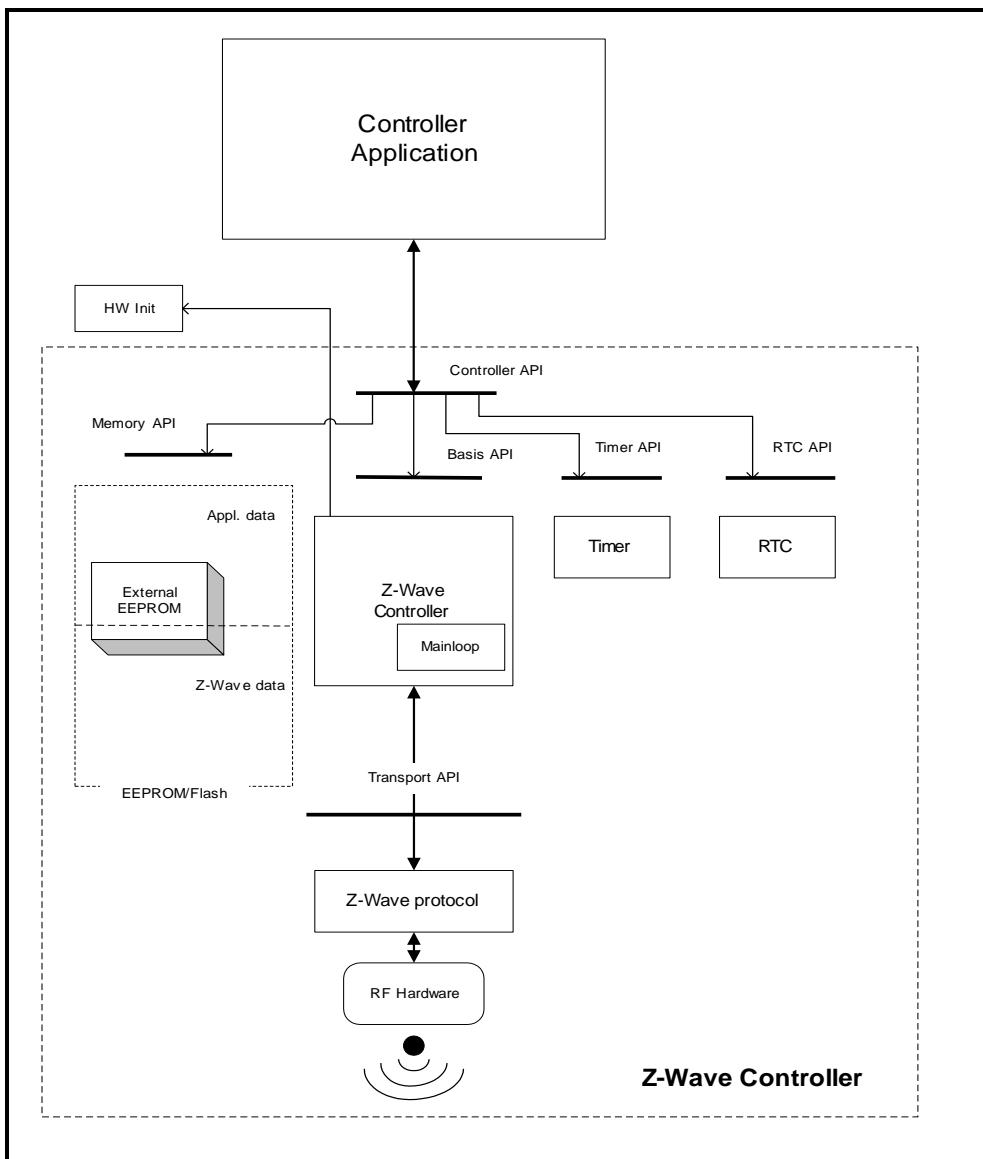
Figure 5. Portable controller node architecture

The Portable Controller node has a unique home ID number assigned, which is stored in the Z-Wave basis area of the external EEPROM. Care must be taken, when reprogramming the external EEPROM, that different controller nodes do not get the same home ID number. Refer to [38] regarding a description of external EEPROM programming.

When new Slave nodes are registered to the Z-Wave network, the Controller node assigns the home ID and a unique node ID to the Slave node. The Slave node stores the home ID and node ID.

When a controller is primary, it will send any networks changes to the SUC node in the network. Controllers can request network topology updates from the SUC node.

The routing attempts done by a portable controller to reach the destination node are as follows:

CONFIDENTIAL

- If LWR do not exist and TRANSMIT_OPTION_ACK set. Try direct with retries.
- If LWR exist and TRANSMIT_OPTION_ACK set. Try direct without retries. In case it fails, try the LWR. In case the LWR also fails, purge it.
- If TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_AUTO_ROUTE are set then calculate up to two routing attempts per entry/repeater node. Defaut maximum number is five to limit number of tries (see ZW_SetRoutingMAX).
- If TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_AUTO_ROUTE are set, then direct with retries.
- If TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_EXPLORE are set then issue an explore frame as last resort.

When developing application software the header file "ZW_controller_api.h" also include the other Z-Wave API header files e.g. ZW_timer_api.h.

The following define must be set when compiling the application: ZW_CONTROLLER.

The application must be linked with ZW_CONTROLLER_PORTABLE_ZW∗S.LIB
(∗ = 030X for ZW0301 modules, etc).

3.9.2   Z-Wave Static Controller Node

The software components of a Z-Wave static controller node are split into a Static Controller application and the Z-Wave Static Controller basis software, which includes the Z-Wave protocol layers and control of the various data stored into the NVM.

The difference between the static controller and the controller described in chapter 3.9.1 is that the static controller cannot be powered down, that is it cannot be used for battery-operated devices. The static controller has the ability to look for neighbors when requested by a controller. This ability makes it possible for a primary controller to assign static routes from a routing slave to a static controller.

The Static Controller can be set as a SUC node, so it can sends network topology updates to any requesting secondary controller. A secondary static controller not functioning as SUC can also request network Topology updates.

The routing attempts done by a static controller to reach the destination node are as follows:

- If LWR do not exist and TRANSMIT_OPTION_ACK set. Try direct when neighbors.
- If LWR exist and TRANSMIT_OPTION_ACK set. Try the LWR. In case the LWR fails, purge it and try direct if neighbor.
- If TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_AUTO_ROUTE are set then calculate up to two routing attempts per entry/repeater node. Defaut maximum number is five to limit number of tries (see ZW_SetRoutingMAX).
- If TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_AUTO_ROUTE are set, then direct with retries.
- If TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_EXPLORE are set then issue an explore frame as last resort.

When developing application software the header file "ZW_controller_static_api.h" also include the other Z-Wave API header files e.g. ZW_timer_api.h.

The following define is being included compiling the application: ZW_CONTROLLER_STATIC.

The application must be linked with ZW_CONTROLLER_STATIC_ZW∗S.LIB
(∗ = 030X for ZW0301 modules, etc).

CONFIDENTIAL

3.9.3   Z-Wave Installer Controller Node

The software components of a Z-Wave Installer Controller are split into an Installer Controller application and the Z-Wave Installer Controller basis software, which includes the Z-Wave protocol layer.

The Installer Controller is essentially a Z-Wave Controller node, which incorporates extra functionality that can be used to implement controllers especially targeted towards professional installers who support and setup a large number of networks.

The routing attempts done by an installer controller to reach the destination node are as follows:

- If LWR do not exist and TRANSMIT_OPTION_ACK set. Try direct with retries.
- If LWR exist and TRANSMIT_OPTION_ACK set. Try direct without retries. In case it fails, try the LWR. In case the LWR also fails, purge it.
- If TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_AUTO_ROUTE are set then calculate up to two routing attempts per entry/repeater node. Defaut maximum number is five to limit number of tries (see ZW_SetRoutingMAX).
- If TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_AUTO_ROUTE are set, then direct with retries.
- If TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_EXPLORE are set then issue an explore frame as last resort.

The following define must be set when compiling the application: ZW_INSTALLER

The application must be linked with ZW_CONTROLLER_INSTALLER_ZW∗S.LIB
(∗ = 030X for ZW0301 modules, etc).

3.9.4   Z-Wave Bridge Controller Node

The software components of a Z-Wave Bridge Controller node are split into a Bridge Controller application and the Z-Wave Bridge Controller basis software, which includes the Z-Wave protocol layer.

The Bridge Controller is essential a Z-Wave Static Controller node, which incorporates extra functionality that can be used to implement controllers, targeted for bridging between the Z-Wave network and others network (ex. UPnP).

The Bridge application interface is an extended Static Controller application interface, which besides the Static Controller application interface functionality gives the application the possibility to manage Virtual Slave nodes. Virtual Slave nodes is a routing slave node without repeater and assign return route functionality, which physically resides in the Bridge Controller. This makes it possible for other Z-Wave nodes to address up to 128 Slave nodes that can be bridged to some functionality or to devices, which resides on a foreign Network type.

The routing attempts done by a bridge controller to reach the destination node are as follows:

- If LWR do not exist and TRANSMIT_OPTION_ACK set. Try direct when neighbors.
- If LWR exist and TRANSMIT_OPTION_ACK set. Try the LWR. In case the LWR fails, purge it and try direct if neighbor.
- If TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_AUTO_ROUTE are set then calculate up to two routing attempts per entry/repeater node. Defaut maximum number is five to limit number of tries (see ZW_SetRoutingMAX).
- If TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_AUTO_ROUTE are set, then direct with retries.
- If TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_EXPLORE are set then issue an explore frame as last resort.

When developing application software the header file "ZW_controller_bridge_api.h" also include the other Z-Wave API header files.

The following define is being included compiling the application: ZW_CONTROLLER_BRIDGE.

The application must be linked with ZW_CONTROLLER_BRIDGE_ZW∗S.LIB
(∗ = 030X for ZW0301 modules, etc).

3.9.5 Z-Wave Routing Slave Node

The software components of a Z-Wave routing slave node are split into a Slave application and the Z-Wave-Slave basis software, which includes the Z-Wave protocol layers.
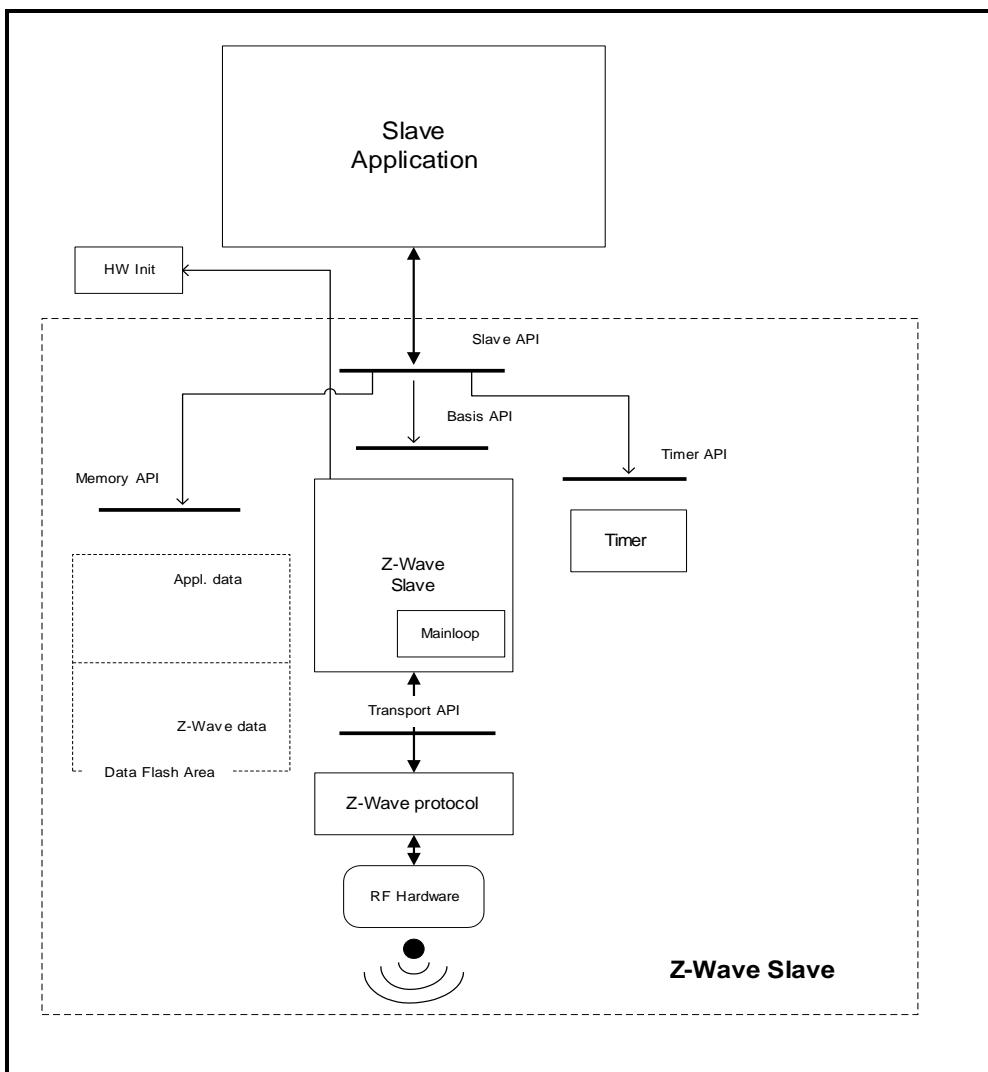
Figure 6. Routing slave node architecture

The routing slave is capable of initiating communication. Examples of a routing slave could be a wall control or temperature sensor. If a user activates the wall control, the routing slave sends an "on" command to a lamp (slave).

The routing slave does not have a complete routing table. Frames are sent to destinations configured during association. The association is performed via a controller. If routing is needed for reaching the destinations, it is also up to the controller to calculate the routes.

Routing slave nodes have an area of 4352 bytes in the flash reserved for storing data, but only 125 bytes can be used directly.

The home ID and node ID of a new node is zero. When registering a slave node to a Z-Wave network the slave node receive home and node ID from the networks primary controller node. These ID's are stored in the Z-Wave basis data area in the flash.

CONFIDENTIAL

The routing slave can send unsolicited and non-routed broadcasts, singlecasts and multicasts. Singlecasts can also be routed. Further it can respond with a routed singlecast (response route) in case another node has requested this by sending a routed singlecast to it. A received multicast or broadcast results in a response route without routing.

A temperature sensor based on a routing slave may be battery operated. To improve battery lifetime, the application may bring the node into sleep mode most of the time. Using the wake-up timer (WUT), the application may wake up once per second, measure the temperature and go back to sleep. In case the measurement exceeded some threshold, a command (e.g. "start heating") may be sent to a heating device before going back to sleep.

The routing attempts done by a routing slave to reach the destination node are as follows:

- If TRANSMIT_OPTION_ACK is set and destination is available in response routes, try response route.
- If TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_AUTO_ROUTE are set then try return routes.
- If TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_AUTO_ROUTE are set then try direct.
- If TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_EXPLORE are set, issue an explore frame as last resort.

The return route comprises of five destinations having up to four routes each. Return routes can also contain direct attempts.

The FIFO contains up to two routes. New routes/direct are qualified for return route insertion by checking if the destination exist and route/direct do not exist. In that event the new route/direct entry will be placed either in a free route or the one having lowest priority. The protocol will update response routes in slaves in the following situations:

- When receiving a successful explorer frame route.
- When receiving a successful routed/direct request from another node.
- When receiving a successful acknowledge for a transmitted explorer frame.
- When receiving a successful acknowledge for a transmitted routed/direct frame.

When developing application software the header file "ZW_slave_routing_api.h" also include the other Z-Wave API header files e.g. ZW_timer_api.h.

The following define will be generated by the headerfile, if it does not already exist when when compiling the application: ZW_SLAVE.

The application must be linked with ZW_SLAVE_ROUTING_ZW∗S.LIB
(∗ = 030X for ZW0301 modules, etc).

3.9.6   Z-Wave Enhanced Slave Node

The Z-Wave enhanced slave has the same basic functionality as a Z-Wave routing slave node, but offers more memory that is non-volatile.
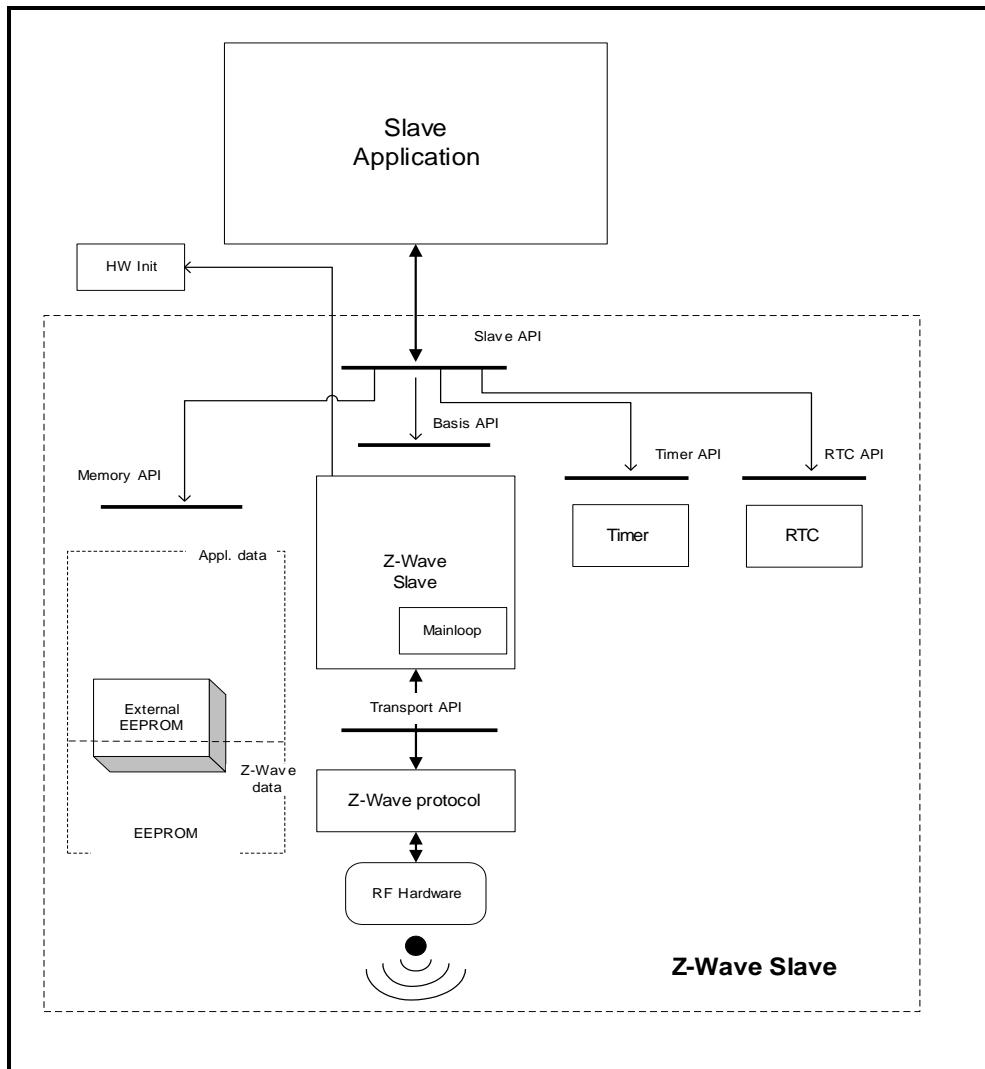
Figure 7. Enhanced slave node architecture

Enhanced slave nodes have an external EEPROM and an WUT. The external EEPROM is used as non volatile memory instead of FLASH. The Z-Wave basis software reserves the first area of the external EEPROM, and the last area of the EEPROM are reserved for the application data. The physical application memory offset is defined in the header file "ZW_eep_addr.h".

The routing attempts done by an enhanced slave to reach the destination node are as follows:

- If TRANSMIT_OPTION_ACK is set and destination is available in response routes, try response route.
- If TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_AUTO_ROUTE are set then try return routes.
- If TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_AUTO_ROUTE are set then try direct.
- If TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_EXPLORE are set, issue an explore frame as last resort.

CONFIDENTIAL

The return route comprises of five destinations having up to four routes each. Return routes can also contain direct attempts.

The FIFO contains up to two routes. New routes/direct are qualified for return route insertion by checking if the destination exist and route/direct do not exist. In that event the new route/direct entry will be placed either in a free route or the one having lowest priority. The protocol will update response routes in slaves in the following situations:

- When receiving a successful explorer frame route.
- When receiving a successful routed/direct request from another node.
- When receiving a successful acknowledge for a transmitted explorer frame.
- When receiving a successful acknowledge for a transmitted routed/direct frame.

When developing application software the header file "ZW_slave_32_api.h" also include the other Z-Wave API header files e.g. ZW_timer_api.h.

The following define will be generated by the headerfile, if it does not already exist when compiling the application: ZW_SLAVE and ZW_SLAVE_32.

The application must be linked with ZW_SLAVE_ENHANCED_ZW∗S.LIB
(∗ = 030X for ZW0301 modules, etc).

3.9.7   Z-Wave Enhanced 232 Slave Node

The Z-Wave enhanced 232 slave has the same basic functionality as a Z-Wave enhanced slave node, but offers return route assignment of up to 232 destination nodes instead of 5.

The routing attempts done by an enhanced 232 slave to reach the destination node are as follows:

- If TRANSMIT_OPTION_ACK is set and destination is available in response routes, try response route.
- If TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_AUTO_ROUTE are set then try return routes.
- If TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_AUTO_ROUTE are set then try direct.
- If TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_EXPLORE are set, issue an explore frame as last resort.

The return route comprises of 232 destinations having up to four routes each. Return routes can also contain direct attempts. Return routes can also contain direct attempts.

The FIFO contains up to one route. New routes/direct are qualified for return route insertion by checking if the destination exist and route/direct do not exist. In that event the new route/direct entry will be placed either in a free route or the one having lowest priority. The protocol will update response routes in slaves in the following situations:

- When receiving a successful explorer frame route.
- When receiving a successful routed/direct request from another node.
- When receiving a successful acknowledge for a transmitted explorer frame.
- When receiving a successful acknowledge for a transmitted routed/direct frame.

When developing application software the header file "ZW_slave_32_api.h" also include the other Z-Wave API header files e.g. ZW_timer_api.h.

The following define will be generated by the headerfile, if it does not already exist when compiling the application: ZW_SLAVE and ZW_SLAVE_32.

CONFIDENTIAL

The application must be linked with ZW_SLAVE_ENHANCED_232_ZW∗S.LIB
(∗ = 030X for ZW0301 modules, etc).

3.9.8   Adding and Removing Nodes to/from the network

Its only controllers that can add new nodes to the Z-Wave network, and reset them again is the primary or inclusion controller. The home ID of the Primary Z-Wave Controller identifies a Z-Wave network.

Information about the result of a learn process is passed to the callback function in a variable with the following structure:

```
typedef struct _LEARN_INFO_
{
      BYTE  bStatus;      /* Status of learn mode                */
      BYTE  bSource;      /* Node id of the node that send node info */
      BYTE  *pCmd;        /* Pointer to Application Node information */
      BYTE  bLen;         /* Node info length                    */
} LEARN_INFO;
```

When adding nodes to the network the controller have a number of choices of how to add and what nodes to add to the network.

Adding a node normally.

The normal way to add a node to the network is to use ZW_AddNodeToNetwork() function on the primary controller, and use the function ZW_SetLearnMode() on the node that should be included into the network.

Adding a new controller and make it the primary controller

A primary controller can add a controller to the network and in the same process give the role as primary controller to the new controller. This is done by using the ZW_ControllerChange() on the primary controller, and use the function ZW_SetLearnMode() on the controller that should be included into the network.. Note that the original primary controller will become a secondary controller when the inclusion is finished.

Create a new primary controller

When there is a Static Update Controller (SUC) in the network then it is possible to create a new primary controller if the original primary controller is lost or broken. This is done by using the ZW_CreateNewPrimary() function on the SUC, and use the function ZW_SetLearnMode() on the controller that should become the new primary controller in the network.

NOTE: A new primary controller will when adding new nodes use the first free node ID starting from 1.

CONFIDENTIAL

The table below lists the options valid on the different types of Controller libraries.

Table 3. Controller functionality

Library used	Node management			
	ZW_AddNodeToNetwork	ZW_RemoveNodeFromNetwork	ZW_ControllerChange	ZW_CreateNewPrimary
Static Controller	Primary	Primary	Primary	When Secondary and only when configured as SUC
(Portable) Controller	Primary	Primary	Primary	Not allowed
Installer Controller	Primary	Primary	Primary	Not allowed
Bridge Controller	Possible but should not be used	Possible but should not be used	Possible but should not be used	Possible but should not be used

Careful considerations should be made as to how the application should implement the process of adding a new controller. Generally speaking the ZW_CreateNewPrimary() option should never be readily available to end-users, since it can be devastating to a network because the user might end up having multiple primary controllers in the network. Another thing to note is that having a Static controller, as a primary controller is only optimal when no portable Controllers exist in the network. A portable Controller offers more flexibility in terms of adding and removing nodes to/from the network since it can be moved around and will report any changes to a Static Controller configured to be a SUC. With these thoughts in mind it is recommended that a network always have one portable controller and if that is not possible, the Primary Static controller should change to secondary when the user wants to include a portable Controller of some sorts.

The most optimal controller setup for networks with several controllers consists of a Static Controller acting as SUC, a portable Primary controller for adding and removing nodes to the network. Controllers besides these two should act as secondary controllers, which from time to time checks with the SUC to get any network updates.

This way the network can be reconfigured and enhanced by using the portable primary controller and all controllers in the network will be able to get the changes from the SUC without user intervention.

SUC ID Server

A SUC with enabled node ID server functionality is called a SUC ID Server (SIS). The SIS becomes the primary controller in the network because it now has the latest update of the network topology and capability to include/exclude nodes in the network. When including a controller to the network it becomes an inclusion controller because it has the capability to include/exclude nodes in the network via the SIS. The inclusion controllers network topology is dated from last time a node was included or it requested a network update from the SIS.

CONFIDENTIAL

3.9.9   The Automatic Network Update

A Z-Wave network consists of slaves, a primary controller and secondary controllers. New nodes can only be added and removed to/from the network by using the primary controller. This could cause secondary controllers and routing slaves to misbehave, if for instance a preferred repeater node is removed. Without automatic network updating a new replication has to be made from the primary controller to all secondary controllers and routing slaves should also be manually updated with the changes. In networks with several controller and routing slave nodes, this process will be cumbersome.

To automate this process, an automatic network update scheme has been introduced to the Z-Wave protocol. To use this scheme a static controller should be available in the network. This static controller should be dedicated to hold a copy of the network topology and the latest changes that have occurred to the network. The static controller used in the Automatic update scheme is called the Static Update Controller (SUC).

Each time a node is added, deleted or a routing change occurs, the primary controller will send the node information to the SUC. Secondary controllers can then ask the SUC if any updates are pending. The SUC will then in turn respond with any changes since last time this controller asked for updates. On the controller requesting an update, **ApplicationControllerUpdate** will be called to notify the application that a new node has been added or removed in the network.

The SUC holds up to 64 changes of the network. If a node requests an update after more than 64 changes occurred, then it will get a complete copy (see **ZW_RequestNetWorkUpdate**).

Routing slaves have the ability to request updates for its known destination nodes. If any changes have occurred to the network, the SUC will send updated route information for the destination nodes to the Routing slave that requested the update. The Routing slave application will be notified when the process is done, but will not get information about any changes to its routes.

If the primary controller sends a new node's node information and its routes to the SUC while it is updating a secondary controller, the updating process will be aborted to process the new nodes information.

CONFIDENTIAL

4 Z-WAVE APPLICATION INTERFACES

The Z-Wave basis software consists of a number of different modules. Time critical functions are written in assembler while the other Z-Wave modules are written in C. The Z-Wave API consists of a number of C functions which give the application programmer direct access to the Z-Wave functionality.

4.1 API usage guidelines

The following guidelines should be followed when making a Z-Wave application.

4.1.1 Buffer protection

Some API calls has one parameter that is a pointer to a buffer in the application SRAM area and another parameter that is a pointer to a callback function. When using these API functions in Z-Wave, it is important that the application does not change the contents of the buffer before the last callback from the API function has been issued. If the content of the buffer is changed before that callback, the Z-Wave protocol might perform the function on invalid data.

4.1.2 Overlapping API calls

In general, it should be avoided to call an API function before the previously started API function is finished and has called the callback function for the last time. Due to the limited resources available for the API not all combinations of API calls will work, some API calls will use the same state machine or the same buffers so if multiple functions is started one or both of the functions might fail.

4.1.3 Error handling.

For purpose of robustness, an application implementation may choose to guard callback API calls whith a timer. In this guide, a timeout value for each API call, which uses a callback, is given. In some functions it is necessarry to to execute some commands in order to recover from a timeout exception. Recovery handling is described for each operation.

CONFIDENTIAL

4.2   Z-Wave Libraries

4.2.1   Library Functionality

Each of the API's provided in the Developer's Kit contains a subset of the full Z-Wave functionality; the table below shows what kind of functionality the API's support independent of the network configuration:

Table 4. Library functionality

	Routing Slave	Enhanced Slave	Portable Controller	Static Controller	Installer Controller	Bridge Controller
Basic Functionality						
Singlecast	X	X	X	X	X	X
Multicast	X	X	X	X	X	X
Broadcast	X	X	X	X	X	X
UART support	X	X	X	X	X	X
SPI support	-	-	-	-	-	-
ADC support	X	X	X	X	X	X
TRIAC control	X	X	X	X	X	X
PWM/HW timer support	X	X	X	X	X	X
Power management	X	X	X	-	X	-
SW timer support	X	X	X	X	X	X
Controller replication	-	-	X	X	X	X
Promiscuous mode	-	-	X	-	X	-
Random number generator	X	X	X	X	X	X
Able to act as NWI center	-	-	X	X	X	X
Able to be included via the NWI mechanism	X	X	X	X	X	X
Able to issue an explorer frame	X	X	X	X	X	X
Able to forward an explorer frame	X	X	-	X	-	-
Memory Location						
Non-volatile RAM in flash	X	-	-	-	-	-
Non-volatile RAM in EEPROM	-	X	X	X	X	X
Network Management						
Network router (repeater)	X	X	-	X	-	-
Assign routes to routing slave	-	-	X	X	X	X
Routing slave functionality	X	X	-	-	-	-
Access to routing table	-	-	X	-	X	-
Maintain virtual slave nodes	-	-	-	-	-	X[†]
Able to be a FLiRS node	X	X	-	-	-	-
Able to beam when repeater	X	X	-	-	-	-
Able to create route containing beam	X[‡]	X[2]	X	X	X	-

[†] Only when secondary controller

[‡] Only when return routes are assigned by a controller capable of creating routes containing beam

CONFIDENTIAL

4.2.1.1   Library Functionality without a SUC/SIS

Some of the API's functionality provided on the Developer's Kit depends on the network configuration. The table below shows what kind of functionality the API's support without a SUC/SIS in the Z-Wave network:

Table 5. Library functionality without a SUC/SIS

	Routing Slave	Enhanced Slave	Portable Controller	Static Controller	Installer Controller	Bridge Controller
Network Management						
Controller replication	-	-	X	X	X	X
Controller shift	-	-	X[§]	X[1]	X[1]	X[1]
Create new primary controller	-	-	-	-	-	-
Request network updates	-	-	-	-	-	-
Request rediscovery of a node	-	-	X[1]	X[1]	X[1]	X[1]
Remove failing nodes	-	-	X[1]	X[1]	X[1]	X[1]
Replace failing nodes	-	-	X[1]	X[1]	X[1]	X[1]
"I'm lost" – cry for help	X	X	-	-	-	-
"I'm lost" – provide help	-	-	-	-	-	-
Provide routing table info	-	-	X	X	X	X

[§] Only when primary controller

CONFIDENTIAL

4.2.1.2   Library Functionality with a SUC

Some of the API's functionality provided on the Developer's Kit depends on the network configuration. The table below shows what kind of functionality the API's support with a Static Update Controller (SUC) in the Z-Wave network:

Table 6. Library functionality with a SUC

	Routing Slave	Enhanced Slave	Portable Controller	Static Controller	Installer Controller	Bridge Controller
Network Management						
Controller replication	-	-	X	X	X	X
Controller shift	-	-	X^1	X^{**}	X^1	X^2
Create new primary controller	-	-	-	$X^{\dagger\dagger}$	-	X^3
Request network updates	X	X	X	X	X	X
Request rediscovery of a node	-	-	X^1	X^1	X^1	X^1
Remove failing nodes	-	-	X^1	X^1	X^1	X^1
Replace failing nodes	-	-	X^1	X^1	X^1	X^1
Set static ctrl. to SUC	-	-	X^1	X^1	X^1	X^1
Work as SUC	-	-	-	X	-	X
Work as primary controller			X	X	X	X
"I'm lost" – cry for help	X	X	-	-	-	-
"I'm lost" – provide help	$X^{\ddagger\ddagger}$	X^3	X^3	$X^{\S\S}$	X^3	X
Provide routing table info	-	-	X	X	X	X

** Only when primary controller and not SUC

†† Only when SUC and not primary controller

‡‡ Only if "always listening"

§§ The library without repeater functionality cannot provide help or forward help requests.

CONFIDENTIAL

4.2.1.3   Library Functionality with a SIS

Some of the API's functionality provided on the Developer's Kit depends on the network configuration. The table below shows what kind of functionality the API's support with a SUC ID Server (SIS) in the Z-Wave network:

Table 7. Library functionality with a SIS

	Routing Slave	Enhanced Slave	Portable Controller	Static Controller	Installer Controller	Bridge Controller
Network Management						
Controller replication	-	-	X	X	X	X
Controller shift	-	-	-	-	-	-
Create new primary controller	-	-	-	-	-	-
Request network updates	X	X	X	X	X	X
Request rediscovery of a node	-	-	X^1	X^1	X^1	X^1
Remove failing nodes	-	-	X^1	X^1	X^1	X^1
Replace failing nodes	-	-	X^1	X^1	X^1	X^1
Set static ctrl. to SIS	-	-	X^2	X^2	X^2	X^2
Work as SIS	-	-	-	X	-	X
Work as inclusion controller			X	X	X	X
"I'm lost" – cry for help	X	X	-	-	-	-
"I'm lost" – provide help	X^3	X^3	X^3	X^4	X^3	X
Provide routing table info	-	-	X	X	X	X

Note that the ability to provide help for "I'm lost" requests is limited to forwarding the request to the SIS. Only the portable controller configured as SIS can actually do the updating of the device.

---

[1] Only when primary/inclusion controller

[2] Only when primary controller

[3] Only if "always listening"

[4] The library without repeater functionality cannot provide help or forward help requests.

CONFIDENTIAL

4.2.1.4   Library Memory Usage

Each API library uses some of the 32KB flash and 2KB RAM available in the ZW0201/ZW0301. Refer to the software release note [20] regarding the minimum amount of flash and RAM that is available for an application build on the library in question. Using the debug functionality of the API will use up to 4K of additional flash and 60 bytes of RAM.

In case an application doesn't have enough flash memory available the following flash usage optimization tips can be used:

1. Use BOOL instead of BYTE for TRUE/FALSE type variables.
2. Try to force the compiler to use registers for local BYTE variables in functions.
3. Avoid using floats because the entire floating point library is linked to the application.
4. Loops are often smallest if they can be done with a do while followed by a decrease of the counter variable.
5. The Keil compiler does not always recognize duplicated code that is used in several different places, so try to move the code to a function and call that instead.
6. Avoid having functions with many parameters, use globals instead.
7. Changing the order of parameters in a function definition will sometimes save code space because the compiler optimization depends on the parameter order.
8. Be aware when using functions from the standard C libraries because the entire library is linked to the application.
9. The dead code elimination in the Keil compiler doesn't always work, so remove all unused code manually.

4.3   Z-Wave Header Files

The C prototypes for the functions in the API's are defined in header files, grouped by functionality:

Protocol releated header files	Description
ZW_controller_api.h	Portable Controller interface. This header should be used together with the Controller Library. Macro defines. Include all necessary header files.
ZW_controller_bridge_api.h	Bridge controller interface. This header should be used together with the Bridge Controller Library. Macro defines. Includes all necessary header files.
ZW_controller_installer_api.h	Installer interface. This header file should be used together with the Installer Controller library. Macro defines. Includes all other necessary header files.
ZW_controller_static_api.h	Static Controller interface. This header should be used together with the Static Controller Library. Macro defines. Includes all necessary header files.
ZW_sensor_api.h	Sensor interface. Macro defines. Includes all other necessary header files.
ZW_slave_32_api.h	Slave interface for ZMXXXX-RE Z-Wave module. Macro defines. Include all header files.
ZW_slave_api.h	Slave interface. Macro defines. Includes all other necessary header files.
ZW_slave_routing_api.h	Routing and Enhanced slave node interface. Macro definitions. Includes all other necessary header files.
ZW_basis_api.h	Z-Wave ⇔ Application general software interface. Interface to common Z-Wave functions.
ZW_transport_api.h	Transfer of data via Z-Wave protocol.
ZW_classcmd.h	Defines for device and command classes used to obtain interoperability between Z-Wave enabled products from different vendors, for a detailed description refer to [1].

CONFIDENTIAL

Various header files	Description
ZW020x.h	Inventra m8051w SFR and ISR defines for the Z-Wave ZW020x RF transceiver.
ZW030x.h	Inventra m8051w SFR and ISR defines for the Z-Wave ZW030x RF transceiver.
ZW_adcdriv_api.h	ADC functionality.
ZW_appltimer.h	PWM/Timer function
ZW_debug_api.h	Debugging functionality via serial port.
ZW_eep_addr.h	Define EEPROM_APPL_OFFSET, which is the offset address to store application data in NVM. Addresses between 0x0 and EEPROM_APPL_OFFSET is used by the protocol.
ZW_mem_api.h	EEPROM interface.
ZW_nodemask_api.h	Routines for manipulation of node ID lists organized as bit masks.
ZW_non_zero.h	Define none zero area of the SRAM (ZW0201/ZW0301 only). See also section 3.1
ZW_power_api.h	ASIC power management functionality.
ZW_RF020x.h	Flash ROM RF table offset for the Z-Wave ZW020x
ZW_RF030x.h	Flash ROM RF table offset for the Z-Wave ZW030x
ZW_SerialAPI.h	Serial API interface with function ID defines etc.
ZW_sysdefs.h	CPU and clock defines.
ZW_timer_api.h	Timer functionality.
ZW_triac_api.h	TRIAC controller functionality.
ZW_typedefs.h	Common used defines (BYTE, WORD…).
ZW_uart_api.h	UART functionality.

4.4   Z-Wave Common API

This section describes interface functions that are implemented within all Z-Wave nodes. The first subsection defines functions that must be implemented within the application modules, while the second subsection defines the functions that are implemented within the Z-Wave basis library.

Functions that do not complete the requested action before returning to the application (e.g. ZW_SEND_DATA) have a callback function pointer as one of the entry parameters. Unless explicitly specified this function pointer can be set to NULL (no action to take on completion).

4.4.1   Required Application Functions

The Z-Wave library requires the functions mentioned here implemented within the Application layer.

4.4.1.1   ApplicationInitHW

BYTE ApplicationInitHW(BYTE bWakeupReason)

ApplicationInitHW should initialize application used hardware. The Z-Wave hardware initialization function set all application IO pins to input mode. The **ApplicationInitHW** function is called by the Z-Wave main function during system startup. At this point of time the Z-Wave timer system is not started so waiting on hardware to get ready may be done by CPU busy loops.

Defined in: ZW_basis_api.h

Return value:

BYTE	TRUE	Application hardware initialized
	FALSE	Application hardware initialization failed. Protocol enters test mode and Calls **ApplicationTestPoll**

Parameters:

bWakeupReason IN	Wakeup flags:	
	ZW_WAKEUP_RESET	Woken up by reset or external interrupt
	ZW_WAKEUP_WUT	Woken up by the WUT timer
	ZW_WAKEUP_SENSOR	Woken up by a wakeup beam

Serial API (Not supported)

CONFIDENTIAL

4.4.1.2   ApplicationInitSW

BYTE ApplicationInitSW(void)

ApplicationInitSW should initialize application used memory and driver software. **ApplicationInitSW** is called from the Z-Wave main function during system startup. At this point of time the Z-Wave timer system is not started, therefore e.g. ZW_MEM_PUT functions cannot be used.

Defined in: ZW_basis_api.h

Return value:

BYTE	TRUE	Application software initialized
	FALSE	Application software initialization failed. (No Z-Wave basis action implemented yet)

Serial API (Not supported)

4.4.1.3   ApplicationTestPoll

void ApplicationTestPoll(void)

The **ApplicationTestPoll** function is the entry point from the Z-Wave basis software to the application software when the production test mode is enabled in the protocol. This will happen when **ApplicationInitHW** returns FALSE. The **ApplicationTestPoll** function will be called indefinitely until the device is reset. The device must be reset and **ApplicationInitHW** must return TRUE in order to exit this mode. When **ApplicationTestPoll** is called the protocol will acknowledge frames sent to home ID 0 and node ID as follows:

Device	Node ID
Slave	`0x00`
Controllers before Dev. Kit v3.40	`0xEF`
Controllers from Dev. Kit v3.40 or later	`0x01`

The following API calls are only available in production test mode:
1. **ZW_EepromInit** is used to initialize the external EEPROM. Remember to initialize controllers with a unique home ID that typically can be transferred via the UART on the production line.
2. **ZW_SendConst** is used to validate RF communication. Remember to enable RF communication when testing products based on a portable controller, routing slave or enhanced slave.

Defined in: ZW_basis_api.h

Serial API (Not supported)

4.4.1.4   ApplicationPoll

void ApplicationPoll(void)

The **ApplicationPoll** function is the entry point from the Z-Wave basis software to the application software modules. The **ApplicationPoll** function is called from the Z-Wave main loop when no low-level time critical actions are active. In order not to disrupt the radio communication and the protocol, no application function must execute code for more than 5ms without returning.

To determine the ApplicationPoll frequency (see table below) is a LED Dimmer application modified to be able to measure how often ApplicationPoll is called via an output pin. The minimum value is measured when the module is idle, i.e. no RF communication, no push button activation etc. The maximum value is measured when the ERTT application at the same time sends Basic Set Commands (value equal 0) as fast as possible to the LED Dimmer (DUT).

Table 8. ApplicationPoll frequency

	ZW0102 LED Dimmer	ZW0201 LED Dimmer	ZW0301 LED Dimmer
Minimum	58 us	7.2 us	7.2 us
Maximum	3.8 ms	2.4 ms	2.4 ms

Defined in: ZW_basis_api.h

Serial API (Not supported)

CONFIDENTIAL

4.4.1.5   ApplicationCommandHandler (Not Bridge Controller library)

In libraries not supporting promiscuous mode (see Table 4):
void ApplicationCommandHandler(BYTE rxStatus,
BYTE sourceNode,
ZW_APPLICATION_TX_BUFFER *pCmd,
BYTE cmdLength)

In libraries supporting promiscuous mode:
void ApplicationCommandHandler(BYTE rxStatus,
BYTE destNode,
BYTE sourceNode,
ZW_APPLICATION_TX_BUFFER *pCmd,
BYTE cmdLength)

The Z-Wave protocol will call the **ApplicationCommandHandler** function when an application command or request has been received from another node. The receive buffer is released when returning from this function. The type of frame used by the request can be determined (single cast, mulitcast or broadcast frame). This is used to avoid flooding the network by responding on a multicast or broadcast. In order not to disrupt the radio communication and the protocol, no application function must execute code for more than 5ms without returning.

All controller libraries (except the Bridge Controller library), requires this function implemented within the Application layer.

Defined in: ZW_basis_api.h

Parameters:

rxStatus IN	Received frame status flags	Refer to ZW_transport_API.h header file
	RECEIVE_STATUS_ROUTED_BUSY xxxxxxx1	A response route is locked by the application
	RECEIVE_STATUS_LOW_POWER xxxxxx1x	Received at low output power level
	RECEIVE_STATUS_TYPE_SINGLE xxxx00xx	Received a single cast frame
	RECEIVE_STATUS_TYPE_BROAD xxxx01xx	Received a broadcast frame
	RECEIVE_STATUS_TYPE_MULTI xxxx10xx	Received a multicast frame
	RECEIVE_STATUS_FOREIGN_FRAME	The received frame is not addressed to this node (Only valid in promiscuous mode)
destNode IN	Command destination Node ID	Only valid in promiscuous mode and for singlecast frames.
sourceNode IN	Command sender Node ID	

CONFIDENTIAL

pCmd IN   Payload from the received frame.   The command class is the very first byte.

cmdLength IN   Number of Command class bytes.

Serial API:

ZW->HOST: REQ | 0x04 | rxStatus | sourceNode | cmdLength | pCmd[ ]

When a foreign frame is received in promiscuous mode:
ZW->HOST: REQ | 0xD1 | rxStatus | sourceNode | cmdLength | pCmd[ ] | destNode

The destNode parameter is only valid for singlecast frames.

CONFIDENTIAL

4.4.1.6   ApplicationNodeInformation

void ApplicationNodeInformation(BYTE *deviceOptionsMask,
**   APPL_NODE_TYPE *nodeType,**
**   BYTE **nodeParm,**
**   BYTE *parmLength)**

The Z-Wave application layer use **ApplicationNodeInformation** to generate the Node Information frame and to save information about node capabilities. Initialize all the Z-Wave application related fields of the Node Information structure in this function. For a description of the Generic Device Classes, Specific Device Classes, and Command Classes refer to [1] and [33]. The deviceOptionsMask is a Bit mask where Listening and Optional functionality flags must be set or cleared accordingly to the nodes capabilities.

 The listening option in the deviceOptionsMask (APPLICATION_NODEINFO_LISTENING) indicates a continuously powered node ready to receive frames. A listening node assists as repeater in the network.

The non-listening option in the deviceOptionsMask (APPLICATION_NODEINFO_NOT_LISTENING) indicates a battery-operated node that power off RF reception when idle (prolongs battery lifetime)..

The optional functionality option in the deviceOptionsMask (APPLICATION_NODEINFO_OPTIONAL_FUNCTIONALITY) indicates that this node supports other command classes than the mandatory classes for the selected generic and specific device class.

Examples:

To set a device as Listening with Optional Functionality:

```
*deviceOptionsMask = APPLICATION_NODEINFO_LISTENING |
                     APPLICATION_NODEINFO_OPTIONAL_FUNCTIONALITY;
```


To set a device as not listening and with no Optional functionality support:

```
*deviceOptionsMask = APPLICATION_NODEINFO_NOT_LISTENING;
```

Note for Controllers: Because controller libraries store some basic information about themselves from ApplicationNodeInformation in nonvolatile memory. ApplicationNodeInformation should be set to the correct values before Application return from **ApplicationInitHW()**, for applications where this cannot be done. The Application must call ZW_SET_DEFAULT() after updating ApplicationNodeInformation in order to force the Z-Wave library to store the correct values.

A way to verify if ApplicationNodeInformation is stored by the protocol is to call **ZW_GetNodeProtocolInfo** to verify that Generic and specific nodetype are correct. If they differ from what is expected, the Application should Set the ApplicationNodeInformation to the correct values and call ZW_SET_DEFAULT() to force the protocol to update its information.

Defined in: ZW_basis_api.h

Parameters:

deviceOptionsMask OUT		Bitmask with options
	APPLICATION_NODEINFO_LISTENING	In case this node is always listening (typically AC powered nodes) and stationary.
	APPLICATION_NODEINFO_NOT_LISTENING	In case this node is non-listening (typically battery powered nodes).
	APPLICATION_NODEINFO_ OPTIONAL_FUNCTIONALITY	If the node supports other command classes than the ones mandatory for this nodes Generic and Specific Device Class
	APPLICATION_FREQ_LISTENING_MODE_250ms	This option bit should be set if the node should act as a Frequently Listening Routing Slave with a wakeup interval of 250ms. This option is only avalaibe on Routing Slaves.
	APPLICATION_FREQ_LISTENING_MODE_1000ms	This option bit should be set if the node should act as a Frequently Listening Routing Slave with a wakeup interval of 250ms. This option is only avalaibe on Routing Slaves.
nodeType OUT	Pointer to structure with the Device Class:	
	(*nodeType).generic	The Generic Device Class [1]. Do not enter zero in this field.
	(*nodeType).specific	The Specific Device Class [1].
nodeParm OUT	Command Class buffer pointer.	Command Classes [1] supported by the device itself and optional Command Classes the device can control in other devices.

CONFIDENTIAL

parmLength OUT   Number of Command Class bytes.

Serial API:

The **ApplicationNodeInformation** is replaced by **SerialAPI_ApplicationNodeInformation**. Used to set information that will be used in subsequent calls to ZW_SendNodeInformation. Replaces the functionality provided by the ApplicationNodeInformation() callback function.

void SerialAPI_ApplicationNodeInformation(BYTE deviceOptionsMask,
**   APPL_NODE_TYPE *nodeType,**
**   BYTE *nodeParm,**
**   BYTE parmLength)**

Information is stored in NVM application area. The define APPL_NODEPARM_MAX in serialappl.h must be modified accordingly to the number of command classes to be notified.

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

The figure below lists the Node Information Frame structure on application level. The Z-Wave Protocol creates this frame via ApplicationNodeInformation. The Node Information Frame structure when transmitted by RF does not include the Basic byte descriptor field. The Basic byte descriptor field on application level is deducted from the Capability and Security byte descriptor fields.

Byte descriptor \ bit number	7	6	5	4	3	2	1	0
Capability	Liste-ning	Z-Wave Protocol Specific Part						
Security	Opt. Func.	Z-Wave Protocol Specific Part						
Reserved	Z-Wave Protocol Specific Part							
Basic	Basic Device Class (Z-Wave Protocol Specific Part)							
Generic	Generic Device Class							
Specific	Specific Device Class							
NodeInfo[0]	Command Class 1							
…	…							
NodeInfo[n-1]	Command Class n							

Figure 8. Node Information frame structure on application level

WARNING: Must use deviceOptionsMask parameter and associated defines to initialize Node Information Frame with respect to listening, non-listening and optional functionality options.

CONFIDENTIAL

4.4.1.7   ApplicationSlaveUpdate (All slave libraries)

void ApplicationSlaveUpdate (BYTE bStatus,
                                BYTE bNodeID,
                                BYTE *pCmd,
                                BYTE bLen)

The Z-Wave protocol also calls **ApplicationSlaveUpdate** when receiving a Node Information Frame and the protocol is not in a state where it needs the node information.

All slave libraries require this function implemented within the Application layer.

Defined in: ZW_slave_api.h

Parameters:

bStatus IN   The status, value could be one of the following:

UPDATE_STATE_NODE_INFO_RECEIVED   A node has sent its node info but the protocol are not in a state where it is needed

bNodeID IN   The updated node's node ID (1..232).

pCmd IN   Pointer of the updated node's node info.

bLen IN   The length of the pCmd parameter.

Serial API:

ZW->HOST: REQ | 0x49 | bStatus | bNodeID | bLen | basic | generic | specific | commandclasses[ ]

CONFIDENTIAL

4.4.1.8   ApplicationControllerUpdate (All controller libraries)

void ApplicationControllerUpdate (BYTE bStatus,
**   BYTE bNodeID,**
**   BYTE *pCmd,**
**   BYTE bLen)**

The Z-Wave protocol in a controller calls **ApplicationControllerUpdate** when a new node has been added or deleted from the controller through the network management features. The Z-Wave protocol calls **ApplicationControllerUpdate** as a result of using the API call **ZW_RequestNodeInfo**. The application can use this functionality to add/delete the node information from any structures used in the Application layer. The Z-Wave protocol also calls **ApplicationControllerUpdate** when a Node Information Frame has been received and the protocol is not in a state where it needs the node information.

ApplicationControllerUpdate is called on the SUC each time a node is added/deleted by the primary controller. **ApplicationControllerUpdate** is called on the SIS each time a node is added/deleted by the inclusion controller. When a node request **ZW_RequestNetWorkUpdate** from the SUC/SIS then the **ApplicationControllerUpdate** is called for each node change (add/delete) on the requesting node. **ApplicationControllerUpdate** is not called on a primary or inclusion controller when a node is added/deleted.

All controller libraries, requires this function implemented within the Application layer.

Defined in: ZW_controller_api.h

Parameters:

bStatus IN   The status of the update process, value could
                     be one of the following:

	UPDATE_STATE_ADD_DONE	A new node has been been given a node ID by the primary or an inclusion controller.
		NOTE: At this point, it is not necessarily possible to route to the node because the range information from the node has not been received yet.
	UPDATE_STATE_DELETE_DONE	A node has been deleted from the network
	UPDATE_STATE_NODE_INFO_RECEIVED	A node has sent its node info either unsolicited or as a response to a ZW_RequestNodeInfo call
	UPDATE_STATE_SUC_ID	The SUC node Id was updated

bNodeID IN   The updated node's node ID (1..232).

pCmd IN   Pointer of the updated node's node info.

bLen IN   The length of the pCmd parameter.

CONFIDENTIAL

Serial API:

ZW->HOST: REQ | 0x49 | bStatus | bNodeID | bLen | basic | generic | specific | commandclasses[ ]

ApplicationControllerUpdate via the Serial API also have the possibility for receiving the status UPDATE_STATE_NODE_INFO_REQ_FAILED, which means that a node did not acknowledge a ZW_RequestNodeInfo call.

CONFIDENTIAL

4.4.1.9   ApplicationCommandHandler_Bridge (Bridge Controller library only)

void ApplicationCommandHandler_Bridge(BYTE rxStatus,
**   BYTE destNode,**
**   BYTE sourceNode,**
**   ZW_MULTI_DEST multi,**
**   ZW_APPLICATION_TX_BUFFER *pCmd,**
**   BYTE cmdLength)**

The Z-Wave protocol will call the **ApplicationCommandHandler_Bridge** function when an application command or request has been received from another node to the Bridge Controller or an existing virtual slave node. The receive buffer is released when returning from this function.

The Z-Wave Bridge Controller library requires this function implemented within the Application layer.

> Defined in: ZW_controller_bridge_api.h

> **Parameters:**

> rxStatus IN   Frame header info:

	RECEIVE_STATUS_ROUTED_BUSY xxxxxxx1	A response route is locked by the application
	RECEIVE_STATUS_LOW_POWER xxxxxx1x	Received at low output power level
	RECEIVE_STATUS_TYPE_SINGLE xxxx00xx	Received a single cast frame
	RECEIVE_STATUS_TYPE_BROAD xxxx01xx	Received a broadcast frame
	RECEIVE_STATUS_TYPE_MULTI xxxx10xx	Received a multicast frame

> destNode IN   Command receiving Node ID. Either Bridge Controller Node ID or virtual slave Node ID.
>
> If received frame is a multicast frame then destNode is not valid and multi points to a multicast structure containing the destination nodes.

> sourceNode IN   Command sender Node ID.

> Multi IN   If received frame is, a multicast frame then multi points at the multicast Structure containing the destination Node IDs.

> pCmd IN   Payload from the received frame. The command class is the very first byte.

> cmdLength IN   Number of Command class bytes.

CONFIDENTIAL

Serial API:

Bridge library based serial API application supporting manual routing (MR) using
FUNC_ID_APPLICATION_COMMAND_HANDLER_BRIDGE for both virtual nodes and the bridge
node itself:
ZW->HOST: REQ | 0xA8 | rxStatus | destNodeID | srcNodeID | cmdLength | pCmd[ ] |
multiDestsOffset_NodeMaskLen | multiDestsNodeMask

Bridge library based serial API application not supporting manual routing using
FUNC_ID_APPLICATION_SLAVE_COMMAND_HANDLER for virtual nodes:
ZW->HOST: REQ | 0xA1 | rxStatus | destNode | sourceNode | cmdLength | pCmd[ ]

Bridge library based serial API application not supporting manual routing using
FUNC_ID_APPLICATION_COMMAND_HANDLER refer to 4.4.1.5 for the bridge node itself:
ZW->HOST: REQ | 0x04 | rxStatus | sourceNode | cmdLength | pCmd[ ]

This target is using old function IDs for backward compatibility with older host implementations,
whereas FUNC_ID_APPLICATION_COMMAND_HANDLER_BRIDGE is the new and improved
interface. The MR target is new, and hence does not have legacy concerns to address.

CONFIDENTIAL

4.4.1.10   ApplicationSlaveNodeInformation (Bridge Controller library only)

void ApplicationSlaveNodeInformation(BYTE destNode,
**   BYTE *listening,**
**   APPL_NODE_TYPE *nodeType,**
**   BYTE **nodeParm,**
**   BYTE *parmLength)**

Request Application Virtual Slave Node information. The Z-Wave protocol layer calls
ApplicationSlaveNodeInformation just before transmitting a "Node Information" frame.

The Z-Wave Bridge Controller library requires this function implemented within the Application layer.

   Defined in: ZW_controller_bridge_api.h

   Parameters:

   destNode IN   Which Virtual Node do we want the node
                                          information from.

   listening OUT   TRUE if this node is always listening and
                                          not moving.

   nodeType OUT   Pointer to structure with the Device Class:

                                          (*nodeType).generic   The Generic Device Class [1].
                                                                                            Do not enter zero in this field.

                                          (*nodeType).specific   The Specific Device Class [1].

   nodeParm OUT   Command Class buffer pointer.   Command Classes [1]
                                                                                            supported by the device itself
                                                                                            and optional Command
                                                                                            Classes the device can control
                                                                                            in other devices.

   parmLength OUT   Number of Command Class bytes.

   Serial API:

   The **ApplicationSlaveNodeInformation** is replaced by
   SerialAPI_ApplicationSlaveNodeInformation. Used to set node information for the Virtual Slave
   Node in the embedded module this node information will then be used in subsequent calls to
   ZW_SendSlaveNodeInformation. Replaces the functionality provided by the
   ApplicationSlaveNodeInformation() callback function.

   void SerialAPI_ApplicationSlaveNodeInformation(BYTE destNode,
   **   BYTE listening,**
   **   APPL_NODE_TYPE * nodeType,**
   **   BYTE *nodeParm,**
   **   BYTE parmLength)**

   HOST->ZW:
   REQ | 0xA0 | destNode | listening | genericType | specificType | parmLength | nodeParm[ ]

CONFIDENTIAL

4.4.1.11 ApplicationRfNotify (ZW0301 only)

void ApplicationRfNotify (BYTE rfState)

This function is used to inform the application about the current state of the radio enabling control of an external power amplifier (PA). The Z-Wave protocol will call the **ApplicationRfNotify** function when the radio changes state as follows:

- From Tx to Rx

- From Rx to Tx

- From powere down to Rx

- From power down to Tx

- When PA is powered up

- When PA is powered down

This enables the application to control an external PA using the appropriate number of I/O pins. For details, refer to [35].

When using an external PA, remember to set the field at FLASH_APPL_PLL_STEPUP_OFFS in App_RFSetup.a51 to 0 (zero) for adjustment of the signal quality. This is necessary to be able to pass a FCC compliance test.

Defined in: ZW_basis_api.h

Parameters:

rfState IN	The current state of the radio.	Refer to ZW_transport_API.h header file
	ZW_RF_TX_MODE	The radio is in Tx state. Previous state is either Rx or power down
	ZW_RF_RX_MODE	The radio in Rx or power down state. Previous state is ether Tx or power down
	ZW_RF_PA_ON	The radio in Tx moode and the PA is powered on
	ZW_RF_PA_OFF	The radio in Tx mode and the PA is powered off

Serial API:

Not implemented

CONFIDENTIAL

4.4.2   Z-Wave Basis API

This section defines functions that are implemented in all Z-Wave nodes.

4.4.2.1   ZW_ExploreRequestInclusion

BYTE ZW_ExploreRequestInclusion()

This function sends out an explorer frame requesting inclusion into a network. If the inclusion request is accepted by a controller in network wide inclusion mode then the application on this node will get notified through the callback from the ZW_SetLearnMode() function. Once a callback is received from ZW_SetLearnMode() saying that the inclusion process has started the application should not make further calls to this function.

NOTE: Recommend not to call this function more than once every 4 seconds.

Defined in: ZW_basis_api.h

Return value:

BYTE	TRUE	Inclusion request queued for transmission
	FALSE	Node is not in learn mode

Serial API

HOST->ZW: REQ | 0x5E

ZW->HOST: RES | 0x5E | retVal

CONFIDENTIAL

4.4.2.2   ZW_GetProtocolStatus

BYTE ZW_GetProtocolStatus(void)

Macro: ZW_GET_PROTOCOL_STATUS()

Report the status of the protocol.

The function return a mask telling which protocol function is currently running

Defined in: ZW_basis_api.h

Return value:

BYTE   Returns the protocol status as one of the
                     following:

          Zero   Protocol is idle.

          ZW_PROTOCOL_STATUS_ROUTING   Protocol is analyzing the routing table.

          ZW_PROTOCOL_STATUS_SUC   SUC sends pending updates.

Serial API

HOST->ZW: REQ | 0xBF

ZW->HOST: RES | 0xBF | retVal

4.4.2.3   ZW_GetRandomWord

**BYTE ZW_GetRandomWord(BYTE *randomWord,
                        BOOL bResetRadio)**

Macro: ZW_GET_RANDOM_WORD(randomWord, bResetRadio)

The API call generates a random word using the ZW0201/ZW0301 builtin random number generator
(RNG). If RF needs to be in Receive then ZW_SetRFReceiveMode should be called afterwards.

NOTE: The ZW0201/ZW0301 RNG is based on the RF transceiver, which must be in powerdown state
(see ZW_SetRFReceiveMode) to assure proper operation of the RNG. Remember to call
ZW_GetRandomWord with bResetRadio = TRUE when the last random word is to be generated. This is
needed for the RF to be reinitialized, so that it can be used to transmit and receive again.

Defined in: ZW_basis_api.h

Return value:

BOOL   TRUE   If possible to generate random number.

                     FALSE   If not possible e.g. RF not powered down.

CONFIDENTIAL

Parameters:

randomWord OUT Pointer to word variable, which
should receive the random word.

bResetRadio IN If TRUE the RF radio is reinitialized
after generating the random word.

Serial API

The Serial API function 0x1C makes use of the ZW_GetRandomWord to generate a specified number
of random bytes and takes care of the handling of the RF:

- Set the RF in powerdown prior to calling the ZW_GetRandomWord the first time, if not
  possible then return result to HOST.

- Call ZW_GetRandomWord until enough random bytes generated or ZW_GetRandomWord
  returns FALSE.

- Call ZW_GetRandomWord with bResetRadio = TRUE to reinitialize the radio.

- Call ZW_SetRFReceiveMode with TRUE if the serialAPI hardware is a listening device or with
  FALSE if it is a non-listening device.

- Return result to HOST.

HOST -> ZW: REQ | 0x1C | [noRandomBytes]

noRandomBytes Number of random bytes needed. Optional if not
present or equal ZERO then 2 random bytes are
returned Range 1...32 random bytes are
supported.

ZW -> HOST: RES | 0x1C | randomGenerationSuccess | noRandomBytesGenerated |
noRandomGenerated[noRandomBytesGenerated]

randomGenerationSuccess TRUE if random bytes could be generated

FALSE if no random bytes could be generated

noRandomBytesGenerated Number of random numbers generated

noRandomBytesGenerated[] Array of generated random bytes

4.4.2.4   ZW_Random

BYTE ZW_Random(void)

Macro: ZW_RANDOM()

A pseudo-random number generator that generates a sequence of numbers, the elements of which are approximately independent of each other. The same sequence of pseudo-random numbers will be repeated in case the module is power cycled. The Z-Wave protocol uses also this function in the random backoff algorithm etc.

Defined in: ZW_basis_api.h

Return value:

BYTE   Random number (0 – 0xFF)

HOST->ZW: REQ | 0x1D

ZW->HOST: RES | 0x1D | rndNo

CONFIDENTIAL

4.4.2.5   ZW_RFPowerLevelSet

BYTE ZW_RFPowerLevelSet(BYTE powerLevel)

Macro: ZW_RF_POWERLEVEL_SET(POWERLEVEL)

Set the power level used in RF transmitting. The actual RF power is dependent on the settings for transmit power level in App_RFSetup.a51. If this value is changed from using the default library value the resulting power levels might differ from the intended values. The returned value is however always the actual one used.

NOTE: This function should only be used in an install/test link situation and the power level should always be set back to normalPower when the testing is done.

Defined in: ZW_basis_api.h

Parameters:

powerLevel IN   Powerlevel to use in RF transmission, valid values:

normalPower	Max power possible
minus1dB	Normal power - 1dB (mapped to minus2dB)
minus2dB	Normal power - 2dB
minus3dB	Normal power - 3dB (mapped to minus4dB)
minus4dB	Normal power - 4dB
minus5dB	Normal power - 5dB (mapped to minus6dB)
minus6dB	Normal power - 6dB
minus7dB	Normal power - 7dB (mapped to minus8dB)
minus8dB	Normal power - 8dB
minus9dB	Normal power - 9dB (mapped to minus10dB)

Return value:

BYTE   The powerlevel set.

Serial API (Serial API protocol version 4):

HOST->ZW: REQ | 0x17 | powerLevel

ZW->HOST: RES | 0x17 | retVal

CONFIDENTIAL

4.4.2.6   ZW_RFPowerLevelGet

BYTE ZW_RFPowerLevelGet(void)

Macro: ZW_RF_POWERLEVEL_GET()

Get the current power level used in RF transmitting.

NOTE: This function should only be used in an install/test link situation.

Defined in: ZW_basis_api.h

Return value:

BYTE   The power level currently in effect during
                    RF transmissions.

Serial API

HOST->ZW: REQ | 0xBA

ZW->HOST: RES | 0xBA | powerlevel

CONFIDENTIAL

4.4.2.7   ZW_RequestNetWorkUpdate

BYTE ZW_RequestNetWorkUpdate (VOID_CALLBACKFUNC (completedFunc)(BYTE txStatus))

Macro: ZW_REQUEST_NETWORK_UPDATE (func)

Used to request network topology updates from the SUC/SIS node. The update is done on protocol level and any changes are notified to the application by calling the **ApplicationControllerUpdate**).

Secondary controllers can only use this call when a SUC is present in the network. All controllers can use this call in case a SUC ID Server (SIS) is available.

Routing Slaves can only use this call, when a SUC is present in the network. In case the Routing Slave has called ZW_RequestNewRouteDestinations prior to ZW_RequestNetWorkUpdate, then Return Routes for the destinations specified by the application in ZW_RequestNewRouteDestinations will be updated along with the SUC Return Route.

NOTE: The SUC can only handle one network update at a time, so care should be taken not to have all the controllers in the network ask for updates at the same time.

WARNING: This API call will generate a lot of network activity that will use bandwidth and stress the SUC in the network. Therefore, network updates should be requested as seldom as possible and never more often that once every hour from a controller.

Defined in: ZW_controller_api.h and ZW_slave_routing_api.h

Return value:

BYTE	TRUE	If the updating process is started.
	FALSE	If the requesting controller is the SUC node or the SUC node is unknown.

Parameters:

completedFunc   Transmit complete call back.
IN

CONFIDENTIAL

Callback function Parameters:

txStatus IN   Status of command:

      ZW_SUC_UPDATE_DONE   The update process succeeded.

      ZW_SUC_UPDATE_ABORT   The update process aborted because of an error.

      ZW_SUC_UPDATE_WAIT   The SUC node is busy.

      ZW_SUC_UPDATE_DISABLED   The SUC functionality is disabled.

      ZW_SUC_UPDATE_OVERFLOW   The controller requested an update after more than 64 changes have occurred in the network. The update information is then out of date in respect to that controller. In this situation the controller have to make a replication before trying to request any new network updates.

Timeout: 65s

Exption recovery: Resume normal operation, no recovery needed

Serial API:

HOST->ZW: REQ | 0x53 | funcID

ZW->HOST: RES | 0x53 | retVal

ZW->HOST: REQ | 0x53 | funcID | txStatus

4.4.2.8   ZW_RFPowerlevelRediscoverySet

void ZW_RFPowerlevelRediscoverySet(BYTE bNewPower)

Macro: ZW_RF_POWERLEVEL_REDISCOVERY_SET(bNewPower)

Set the power level locally in the node when finding neighbors. The default power level is normal power minus 6dB. It is only necessary to call ZW_RFPowerlevelRediscoverySet in case a value different from the default power level is needed. Furthermore is it only necessary to set a new power level once then the new level will be used every time a neighbour discovery is performed. The API call can be called from ApplicationInit or during runtime from ApplicationPoll or ApplicationCommandHandler.

NOTE: Be aware of that weak RF links can be included in the routing table in case the reduce power level is set to 0dB (normalPower). Weak RF links can increase latency in the network due to retries to get through. Finally, will a large reduction in power level result in a reduced range between the nodes in the network, which results in an increased latency due to an increase in the necessary hops to reach the destination.

Defined in: ZW_basis_api.h

Parameters:

bNewPower IN   Powerlevel to use when doing
                   neighbor discovery, valid values:

normalPower	Max power possible
minus1dB	Normal power - 1dB (mapped to minus2dB)
minus2dB	Normal power - 2dB
minus3dB	Normal power - 3dB (mapped to minus4dB)
minus4dB	Normal power - 4dB
minus5dB	Normal power - 5dB (mapped to minus6dB)
minus6dB	Normal power - 6dB
minus7dB	Normal power - 7dB (mapped to minus8dB)
minus8dB	Normal power - 8dB
minus9dB	Normal power - 9dB (mapped to minus10dB)

Serial API:

HOST->ZW: REQ | 0x1E | powerLevel

CONFIDENTIAL

4.4.2.9   ZW_SendNodeInformation

BYTE ZW_SendNodeInformation(BYTE destNode,
                              BYTE txOptions,
                              VOID_CALLBACKFUNC(completedFunc)(BYTE txStatus))

Macro: ZW_SEND_NODE_INFO(node,option,func)

Create and transmit a "Node Information" frame. The Z-Wave transport layer builds a frame, request application node information (see **ApplicationNodeInformation**) and queue the "Node Information" frame for transmission. The completed call back function (**completedFunc**) is called when the transmission is complete.

The Node Information Frame is a protocol frame and will therefore not be directly available to the application on the receiver. The API call ZW_SetLearnMode() can be used to instruct the protocol to pass the Node Information Frame to the application.

When ZW_SendNodeInformation() is used in learn mode for adding or removing the node from the network the transmit option TRANSMIT_OPTION_LOW_POWER should NOT be used.

NOTE: ZW_SendNodeInformation uses the transmit queue in the API, so using other transmit functions before the complete callback has been called by the API is not recommended.

Defined in: ZW_basis_api.h

Return value:

BYTE	TRUE	If frame was put in the transmit queue
	FALSE	If it was not (callback will not be called)

Parameters:

destNode IN   Destination Node ID
                   (NODE_BROADCAST == all nodes)

txOptions IN   Transmit option flags.
                   (see **ZW_SendData**)

completedFunc   Transmit completed call back function
IN

Callback function Parameters:

txStatus IN   (see **ZW_SendData**)

Timeout: 65s

Exception recovery: Resume normal operation, no recovery needed

CONFIDENTIAL

Serial API:

HOST->ZW: REQ | 0x12 | destNode | txOptions | funcID

ZW->HOST: RES | 0x12 | retVal

ZW->HOST: REQ | 0x12 | funcID | txStatus

CONFIDENTIAL

4.4.2.10   ZW_SendTestFrame

BYTE ZW_SendTestFrame(BYTE nodeID,
                        BYTE powerlevel,
                        VOID_CALLBACKFUNC(func)(BYTE txStatus))

Macro: ZW_SEND_TEST_FRAME(nodeID, power, func)

Send a test frame directly to nodeID without any routing, RF transmission power is previously set to powerlevel by calling ZW_RF_POWERLEVEL_SET. The test frame is acknowledged at the RF transmission powerlevel indicated by the parameter powerlevel by nodeID (if the test frame got through). This test will be done using 9600 kbit/s transmission rate.

NOTE: This function should only be used in an install/test link situation.

   Defined in: ZW_basis_api.h

   Parameters:

   nodeID IN   Node ID on the node ID (1..232)
                   the test frame should be
                   transmitted to.

   powerLevel IN   Powerlevel to use in RF
                   transmission, valid values:

                   normalPower   Max power possible

                   minus1dB   Normal power - 1dB (mapped to minus2dB[***])

                   minus2dB   Normal power - 2dB

                   minus3dB   Normal power - 3dB (mapped to minus4dB)

                   minus4dB   Normal power - 4dB

                   minus5dB   Normal power - 5dB (mapped to minus6dB)

                   minus6dB   Normal power - 6dB

                   minus7dB   Normal power - 7dB (mapped to minus8dB)

                   minus8dB   Normal power - 8dB

                   minus9dB   Normal power - 9dB (mapped to minus10dB)

   func IN   Call back function called when
                   done.

   Callback function Parameters:

[***] 200/300 Series support only -2dB power level steps

CONFIDENTIAL

txStatus IN   (see **ZW_SendData**)

Return value:

BYTE   FALSE   If transmit queue overflow.

Timeout: 200ms

Exception recovery: Resume normal operation, no recovery needed

CONFIDENTIAL

Serial API

HOST->ZW: REQ | 0xBE | nodeID| powerlevel | funcID

ZW->HOST: REQ | 0xBE | retVal

ZW->HOST: REQ | 0xBE | funcID | txStatus

4.4.2.11   ZW_SetExtIntLevel

void ZW_SetExtIntLevel(BYTE intSrc, BOOL triggerLevel)

Macro: ZW_SET_EXT_INT_LEVEL(SRC, TRIGGER_LEVEL)

Set the trigger level for external interrupt 0 or 1. Level or edge triggered is selected as follows:

	Level Triggered	Edge Triggered
External interrupt 0	IT0 = 0;	IT0 = 1;
External interrupt 1	IT1 = 0;	IT1 = 1;

Defined in: ZW_basis_api.h

Parameters:

intSrc IN   The external interrupt valid values:

      ZW_INT0   External interrupt 0 (PIN P1_6)

      ZW_INT1   External interrupt 1 (PIN P1_7)

triggerLevel IN   The external interrupt trigger level:

      TRUE   Set the interrupt trigger to high level
                                   /Rising edge

      FALSE   Set the the interrupt trigger to low level
                                   /Faling edge

Serial API

HOST->ZW: REQ | 0xB9 | intSrc | triggerLevel

4.4.2.12   ZW_SetPromiscuousMode (Not Bridge Controller library)

void ZW_SetPromiscuousMode(BOOL state)
Macro: ZW_SET_PROMISCUOUS_MODE(state)

ZW_SetPromiscuousMode Enable / disable the promiscuous mode.

When promiscuous mode is enabled, all application layer frames will be passed to the application layer regardless if the frames are addressed to another node. When promiscuous mode is disabled, only the frames addressed to the node will be passed to the application layer.

Promiscuously received frames are delivered to the application via the ApplicationCommandHandler callback function (see section 4.4.1.5).

    Defined in: ZW_basis_api.h

    Parameters:

    state IN   TRUE to enable the promiscuous mode,
                        FALSE to disable it.

    Serial API:

    HOST->ZW: REQ | 0xD0 | state

    See section 4.4.1.5 for callback syntax when a frame has been promiscuously received.

CONFIDENTIAL

4.4.2.13   ZW_SetRFReceiveMode

BYTE ZW_SetRFReceiveMode(BYTE mode)

Macro: ZW_SET_RX_MODE(mode)

ZW_SetRFReceiveMode is used to power down the RF when not in use e.g. expects nothing to be received. **ZW_SetRFReceiveMode** can also be used to set the RF into receive mode. This functionality is useful in battery powered Z-Wave nodes e.g. the Z-Wave Remote Controller. The RF is automatic powered up when transmitting data.

Defined in:	ZW_basis_api.h

Return value:

BYTE	TRUE	If operation was successful
	FALSE	If operation was none successful

Parameters:

mode IN	TRUE	On: Set the RF in receive mode and starts the receive data sampling
	FALSE	Off: Set the RF in power down mode (for battery power save).

Serial API

HOST->ZW: REQ | 0x10 | mode

ZW->HOST: RES | 0x10 | retVal

CONFIDENTIAL

4.4.2.14   ZW_SetSleepMode

BOOL ZW_SetSleepMode(BYTE mode,
                                       BYTE intEnable,
                                       BYTE beamCount)

Macro: ZW_SET_SLEEP_MODE(MODE,MASK_INT)

Set the CPU in a specified power down mode. Everything shut down except RAM (IDATA and XDATA), brown-out detection and an optional low power timer (WUT).

Battery-operated devices use this function in order to save power when idle. Notice that ZW_SetSleepMode() doesn't go into sleep mode immediately, it sets a sleep state flag and return. Then at a later point when the protocol is idle (stopped RF transmission etc.) the CPU will power down.

The RF transceiver is turned off so nothing can be received while in WUT or STOP mode. The ADC is also disabled when in STOP or WUT mode. The Z-Wave main poll loop is stopped until the CPU is awake again. Refer to the mode parameter description regarding how the CPU can be wakened up from sleep mode. In STOP and WUT modes can the interrupt(s) be masked out so they cannot wake up the ASIC.

Any external hardware controlled by the application should be turned off before returning from the application poll function.

For more information on the best way to use this API call see section 4.4.9.

The Z-Wave main poll loop is stopped until the CPU is wakened.

Defined in: ZW_power_api.h

Return values

BOOL	TRUE	The chip will power down when the protocol is ready
	FALSE	The protocol can not power down because a wakeup beam is being received, try again later.

CONFIDENTIAL

Parameters:

mode IN   Specify the type of power save mode:

ZW_STOP_MODE   The whole ASIC is turned down. The
                                    ASIC can be wakened up again by
                                    Hardware reset or by the external
                                    interrupt INT1.

ZW_WUT_MODE   The ASIC is powered down, and it can
                                    only be waked by the timer timeout or by
                                    the external interrupt INT1. The time out
                                    value of the WUT can be set by the API
                                    call **ZW_SetWutTimeout**. When the
                                    ASIC is waked from the WUT_MODE,
                                    the API call **ZW_IsWutFired** can be used
                                    to test if the ASIC is waked up by timeout
                                    or INT1. The ASIC wake up from WUT
                                    mode from the reset state. The timer
                                    resolution in this mode is one second.
                                    The maximum timeout value is 256 secs.

ZW_WUT_FAST_MODE   This mode has the same functionality as
                                    ZW_WUT_MODE, except that the timer
                                    resolution is 1/128 sec. The maximum
                                    timeout value is 2 secs. This mode is only
                                    available in ZW0301.

ZW_FREQUENTLY_LISTENING_MODE   This mode make the module enter a
                                    Frequently Listening mode where the
                                    module will wakeup for a few
                                    milliseconds every 1000ms or 250ms and
                                    check for radio transmissions to the
                                    module (See 4.4.1.6 for details about
                                    selecting wakeup speed). The application
                                    will only wakeup if there is incoming RF
                                    traffic or if the intEnable or beamCount
                                    parameters are used.

CONFIDENTIAL

intEnable IN	Interrupt enable bit mask. If a bit mask is 1, the corresponding interrupt is enabled and this interrupt will wakeup the ASIC from power down. Valid bit masks are:	
	ZW_INT_MASK_EXT1	External interrupt 1 (PIN P1_7) is enabled as interrupt source
	0x00	No external Interrupts will wakeup. Usefull in WUT mode
beamCount IN	Frequently listening WUT wakeups	
	0x00	No WUT wakeups in Frequently listening mode. Both macro and serial API call use this value when called.
	0x01-0xFF	Number of frequently listening wakeup interval between the module does a normal WUT wakeup. This parameter is only used if mode is set to ZW_FREQUENTLY_LISTENING_MODE.

Serial API

HOST->ZW: REQ | 0x11 | mode | intEnable

CONFIDENTIAL

4.4.2.15   ZW_Type_Library

BYTE ZW_Type_Library(void)

Macro: ZW_TYPE_LIBRARY()

Get the Z-Wave library type.

Defined in: ZW_basis_api.h

Return value:

BYTE   Returns the library type as one of the following:

ZW_LIB_CONTROLLER_STATIC	Static controller library
ZW_LIB_CONTROLLER_BRIDGE	Bridge controller library
ZW_LIB_CONTROLLER	Portable controller library
ZW_LIB_SLAVE_ENHANCED	Enhanced slave library
ZW_LIB_SLAVE_ROUTING	Routing slave library
ZW_LIB_SLAVE	Slave library
ZW_LIB_INSTALLER	Installer library

Serial API

HOST->ZW: REQ | 0xBD

ZW->HOST: RES | 0xBD | retVal

CONFIDENTIAL

4.4.2.16 ZW_Version

BYTE ZW_Version(BYTE *buffer)

Macro: ZW_VERSION(buffer)

Get the Z-Wave basis API library version.

Defined in: ZW_basis_api.h

Parameters:

buffer OUT Returns the API library version in text
using the format:

Z-Wave x.yy

where x.yy is the library version.

Return value:

BYTE Returns the library type as one of the
following:

ZW_LIB_CONTROLLER_STATIC	Static controller library
ZW_LIB_CONTROLLER_BRIDGE	Bridge controller library
ZW_LIB_CONTROLLER	Portable controller library
ZW_LIB_SLAVE_ENHANCED	Enhanced slave library
ZW_LIB_SLAVE_ROUTING	Routing slave library
ZW_LIB_SLAVE	Slave library
ZW_LIB_INSTALLER	Installer library

Serial API:

HOST->ZW: REQ | 0x15

ZW->HOST: RES | 0x15 | buffer (12 bytes) | library type

CONFIDENTIAL

An additional call is offered capable of returning Serial API version number, Serial API capabilities, nodes currently stored in the EEPROM (only controllers) and chip used.

HOST->ZW: REQ | 0x02

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

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

Nodes[29] is a node bitmask.

Capabilities flag:
Bit 0: 0 = Controller API; 1 = Slave API
Bit 1: 0 = Timer functions not supported; 1 = Timer functions supported.
Bit 2: 0 = Primary Controller; 1 = Secondary Controller
Bit 3-7: reserved

The chip used can be determined as follows:

Z-Wave Chip	Chip_type	Chip_version
ZW0102	0x01	0x02
ZW0201	0x02	0x01
ZW0301	0x03	0x01

Timer functions are: TimerStart, TimerRestart and TimerCancel.

4.4.2.17   ZW_VERSION_MAJOR / ZW_VERSION_MINOR / ZW_VERSION_BETA

Macro: ZW_VERSION_MAJOR/ZW_VERSION_MINOR/ ZW_VERSION_BETA

These #defines can be used to get a decimal value of the used Z-Wave library. ZW_VERSION_MINOR should be 0 padded when displayed to users EG: ZW_VERSION_MAJOR = 1 ZW_VERSION_MINOR =2 should be shown as: 1.02 to the user where as ZW_VERSION_MAJOR = 1 ZW_VERSION_MINOR =20 should be shown as 1.20.

ZW_VERSION_BETA is only defined for beta releases of the Z-Wave Library. In which case it is defined as a single char for instance: 'b'

Defined in: ZW_basis_api.h

Serial API (Not supported)

CONFIDENTIAL

4.4.2.18   ZW_WatchDogEnable

void   ZW_WatchDogEnable(void)

Macro: ZW_WATCHDOG_ENABLE()

Enables the ASIC's built in watchdog , which by default is disabled.
The watchdog timeout interval depends on the hardware platform.

Z-Wave Chip	Watchdog interval
ZW0102	0.56 s
ZW0201	1.05 s
ZW0301	1.05 s

The watchdog must be kicked at least one time per interval. Failing to do so will cause the ASIC to be reset.

To avoid unintentional reset of the application during initialization, the watchdog may be kicked one or more times in the function **ApplicationInitSW**.

Some software defects can be difficult to diagnose when the watchdog is enabled because the application will reboot when the watchdog resets the ASIC. Therefore, it is recommended to also test the device with the watchdog disabled.

   Defined in: ZW_basis_api.h

   Serial API

   HOST->ZW: REQ | 0xB6

4.4.2.19   ZW_WatchDogDisable

void   ZW_WatchDogDisable(void)

Macro: ZW_WATCHDOG_DISABLE ()

Disable the ASIC's built in watchdog.

   Defined in: ZW_basis_api.h

   Serial API

   HOST->ZW: REQ | 0xB7

CONFIDENTIAL

4.4.2.20   ZW_WatchDogKick

void   ZW_WatchDogKick(void)

Macro: ZW_WATCHDOG_KICK ()

To keep the watchdog timer from resetting the ASIC, you've got to kick it regularly. The ZW_WatchDogKick API call must be called in the function **ApplicationPoll** to assure correct detection of any software anomalies etc.

   Defined in: ZW_basis_api.h

   Serial API

   HOST->ZW: REQ | 0xB8

CONFIDENTIAL

4.4.3   Z-Wave Transport API

The Z-Wave transport layer controls transfer of data between Z-Wave nodes including retransmission, frame check and acknowledgement. The Z-Wave transport interface includes functions for transfer of data to other Z-Wave nodes. Application data received from other nodes is handed over to the application via the **ApplicationCommandHandler** function. The ZW_MAX_NODES define defines the maximum of nodes possible in a Z-Wave network.

4.4.3.1   ZW_SendData

BYTE ZW_SendData(BYTE nodeID,
**   BYTE *pData,**
**   BYTE dataLength,**
**   BYTE txOptions,**
**   Void (*completedFunc)(BYTE txStatus))**

NOTE: Only libraries without manual routing functionality support ZW_SendData.

Macro: ZW_SEND_DATA(node,data,length,options,func)

The SendData function is used to transmit contents of a data buffer to a single Z-Wave Node or all Z-Wave Nodes (broadcast). The data buffer contents are encapsulated in a Z-Wave transport frame by adding a protocol header and a checksum trailer. The frame is appended to the end of the transmit queue (first in; first out) and transmitted whenever possible.

When communicating to a Frequently Listening Routing Slave (FLiRS) the API call automatically generates a wakeup beam to awake the FLiRS.

A bridge controller library MUST NOT send to a virtual node belonging to the bridge itself.

The following parameters MUST be specified for the SendData function.

4.4.3.1.1   nodeID parameter
The nodeID parameter MUST specify the destination nodeID.

The nodeID parameter MAY specify the broadcast nodeID (0xFF).

4.4.3.1.2   *pData parameter
The *pData parameter MUST specify a pointer to a data buffer containing a valid Z-Wave command.
The data buffer referenced by the *pData parameter MUST contain the number of bytes indicated by the dataLength parameter.

4.4.3.1.3   dataLength parameter
The data buffer referenced by the *pData parameter is used to hold a valid Z-Wave command.
The dataLength parameter MUST specify the length of the Z-Wave command.

4.4.3.1.4   txOptions parameter
The calling application MUST compose the txOptions parameter value by combining relevant options chosen from the table below.

One or more callbacks to the completedFunc pointer indicate the status of the operation.

CONFIDENTIAL

Table 9. SendData :: txOptions

TRANSMIT_OPTION_	Description	Priority
ACK	Request acknowledged transmission.	If ACK is disabled (0), all other options are ignored by the SendData function
NO_ROUTE	Request acknowledged transmission and explicitly disable routing.	ACK MUST be enabled (1)
AUTO_ROUTE	Request acknowledged transmission and allow routing. If TRANSMIT_OPTION_AUTO_ROUTE == 0, only the Last Working Route is used for routing if direct range transmission fails. If TRANSMIT_OPTION_AUTO_ROUTE == 1, routed transmission uses the Last Working Route and routing table if direct range transmission fails	ACK MUST be enabled (1) NO_ROUTE MUST be disabled (0)
EXPLORER	Request acknowledged transmission and allow routing. Allow reactive route recovery if Last Working Route, routing table and direct range transmission fails.	ACK MUST be enabled (1) NO_ROUTE MUST be disabled (0) AUTO_ROUTE SHOULD be enabled (1)

If the broadcast nodeID (0xFF) is specified, the txOptions parameter SHOULD carry the following option values

- TRANSMIT_OPTION_ACK = 0
- TRANSMIT_OPTION_NO_ROUTE = 1
- TRANSMIT_OPTION_AUTO_ROUTE = 0
- TRANSMIT_OPTION_EXPLORER = 0

Table 10. Use of transmit options for controller libraries

TRANSMIT_OPTION_			Protocol behaviour
NO_ROUTE	ACK	AUTO_ROUTE	
1	0	(ignore)	Transmit frame with no routing, nor retransmission; just as if it was a broadcast frame.
1	1	(ignore)	Frame will be transmitted with direct communication i.e. no routing regardless whether a LWR exist or not.
0	1	0	In case direct transmission fails, the frame will be transmitted using LWR if one exists to the destination in question.
0	1	1	If direct communication fails, then attempt with LWR. If LWR also fails or simply do not exist to the destination, then routes from the routing table will be used.

4.4.3.1.4.1 TRANSMIT_OPTION_ACK

The transmit option TRANSMIT_OPTION_ACK MAY be used to request the destination node to return a transfer acknowledgement. The Z-Wave protocol layer will retry the transmission if no acknowledgement is received.

The transmit option TRANSMIT_OPTION_ACK SHOULD be specified for all normal application communication.

If the nodeID parameter specifies the broadcast nodeID (0xFF), the Z-Wave protocol layer ignores the transmit option TRANSMIT_OPTION_ACK.

4.4.3.1.4.2 TRANSMIT_OPTION_NO_ROUTE

The transmit option TRANSMIT_OPTION_NO_ROUTE MAY be used to force the protocol to send the frame without routing. All available routing information is ignored.

The transmit option TRANSMIT_OPTION_NO_ROUTE SHOULD NOT be specified for normal application communication.

If the nodeID parameter specifies the broadcast nodeID (0xFF), the Z-Wave protocol layer ignores the transmit option TRANSMIT_OPTION_NO_ROUTE.

4.4.3.1.4.3 TRANSMIT_OPTION_AUTO_ROUTE

The transmit option TRANSMIT_OPTION_AUTO_ROUTE MAY be used to enable routing.

The Z-Wave protocol layer will then try transmitting the frame via repeater nodes in case destination node is out of direct range.

Controller nodes MAY use the TRANSMIT_OPTION_AUTO_ROUTE to enable routing via Last Working Routes, calculated routes and routes discovered via explorer route recovery. See **Error! Reference source not found.** For details.

Routing Slave and Enhanced Slave nodes MAY use the TRANSMIT_OPTION_AUTO_ROUTE to enable routing via return routes for the actual destination nodeID (if any exist).

CONFIDENTIAL

If the nodeID parameter specifies the broadcast nodeID (0xFF), the Z-Wave protocol layer ignores the transmit option TRANSMIT_OPTION_AUTO_ROUTE.

4.4.3.1.4.4 TRANSMIT_OPTION_EXPLORER

The transmit option TRANSMIT_OPTION_EXPLORER MAY be used to enable reactive route recovery. Reactive route recovery allows a node to discover new routes if all known routes are failing.
An explorer frame cannot wake up FLiRS nodes.

An explorer frame uses normal RF power level minus 6dB. This is also the power level used by a node finding its neighbors.

The API function ZW_SetRoutingMAX MAY be used to specify the maximum number of routing attempts based on routing table lookups to use before the Z-Wave protocol layer resorts to reactive route recovery.
A default value of five routing attempts SHOULD be used.

For backwards compatibility reasons will SDK 4.5x nodes communicating with nodes, which do not support on-demand route resolution ignore transmit option flag TRANSMIT_OPTION_EXPLORE.

4.4.3.1.4.5 TRANSMIT_OPTION_LOW_POWER

The TRANSMIT_OPTION_LOW_POWER option should only be used when the two nodes that are communicating are close to each other (<2 meter). In all other cases, this option SHOULD NOT be used.

4.4.3.1.4.6 completedFunc

The **completedFunc** parameter MUST specify the calling address of a function that can be called when the SendData frame transmission completes. Completion includes a range of possible situations:

- Direct range frame was successfully transmitted (as requested) without acknowledgement
- Direct range frame was successfully acknowledged
- Routed frame was successfully acknowledged

The transmit status txStatus indicates how the transmission operation was completed.

Table 11. txStatus values

txStatus	Description
TRANSMIT_COMPLETE_OK	The operation was successful.
TRANSMIT_COMPLETE_NO_ACK	No acknowledgement was received from the destination node.
TRANSMIT_COMPLETE_FAIL	Indicates that the network is busy (jammed).

CONFIDENTIAL

WARNING: Allways use the completeFunc callback to determine when the next frame can be send. Calling the ZW_SendData or ZW_SendDataMulti in a loop without checking the completeFunc callback will overflow the transmit queue and eventually fail. The data buffer in the application must not be changed before completeFunc callback is received because it is only the pointer there is passed to the transmit queue.

4.4.3.1.5 Payload size

The maximum size of a frame is 64 bytes. The protocol header and checksum takes 10 bytes in a single cast or broadcast frame leaving 54 bytes for the payload. A S0 security enabled single cast takes 20 bytes as overhead. The maximum dataLength field depends on the transmit options and whether a non-secure/secure frame is used.

Table 12. Maximum payload size

Transmit option	Maximum dataLength	
Notice: Always use lowest maximum dataLength depending on options used.	Non-secure	Secure
TRANSMIT_OPTION_EXPLORE	46 bytes	26 bytes
TRANSMIT_OPTION_AUTO_ROUTE	48 bytes	28 bytes
TRANSMIT_OPTION_NO_ROUTE	54 bytes	34 bytes

4.4.3.1.6 Embedded API function prototypes

Defined in: ZW_transport_api.h

Return value:

BYTE   FALSE   If transmit queue overflow

Parameters:

nodeID IN   Destination node ID   The frame will also be transmitted in case
                (NODE_BROADCAST == all nodes)   the source node ID is equal destination
                                                       node ID

pData IN   Data buffer pointer

dataLength IN   Data buffer length   The maximum dataLength field depends
                                                       on the transmit options and whether a
                                                       non-secure/secure frame is used. For
                                                       details, see section 3.4. The payload
                                                       must be minimum one byte.

txOptions IN   Transmit option flags:

               TRANSMIT_OPTION_LOW_POWER   Transmit at low output power level (1/3 of
                                                  normal RF range).

               TRANSMIT_OPTION_NO_ROUTE   Only send this frame directly, even if a
                                                  response route exist

               TRANSMIT_OPTION_ACK   Request acknowledge from destination
                                                  node.

               TRANSMIT_OPTION_AUTO_ROUTE   <u>Controllers:</u>
                                                  Request retransmission via repeater
                                                  nodes (at normal output power level).
                                                  Number of max routes can be set using
                                                  ZW_SetRoutingMax

                                                  <u>Routing and Enhanced Slaves:</u>
                                                  Send the frame to nodeID using the
                                                  return routes assigned for nodeID to the
                                                  routing/enhanced slave, if no routes are
                                                  valid then transmit directly to nodeID (if
                                                  nodeID = NODE_BROADCAST then the
                                                  frame will be a BROADCAST).
                                                  If return routes exists and the nodeID =
                                                  NODE_BROADCAST then the frame will
                                                  be transmitted to all assigned return route
                                                  destinations. If nodeID !=
                                                  NODE_BROADCAST then the frame will
                                                  be transmitted via the assigned return
                                                  routes for nodeID.

               TRANSMIT_OPTION_EXPLORE   Transmit frame as an explore frame if
                                                  everything else fails.

completedFunc   Transmit completed call back function

Callback function Parameters:

txStatus   Transmit completion status:

               TRANSMIT_COMPLETE_OK   Successfully

               TRANSMIT_COMPLETE_NO_ACK   No acknowledge is received before timeout
                                                  from the destination node. Acknowledge is
                                                  discarded in case it is received after the
                                                  timeout.

               TRANSMIT_COMPLETE_FAIL   Not possible to transmit data because the
                                                  Z-Wave network is busy (jammed).

Timeout: 65s

Exception recovery: If a timeout occurs, it is important to call ZW_SendDataAbort to stop the sending of the frame.
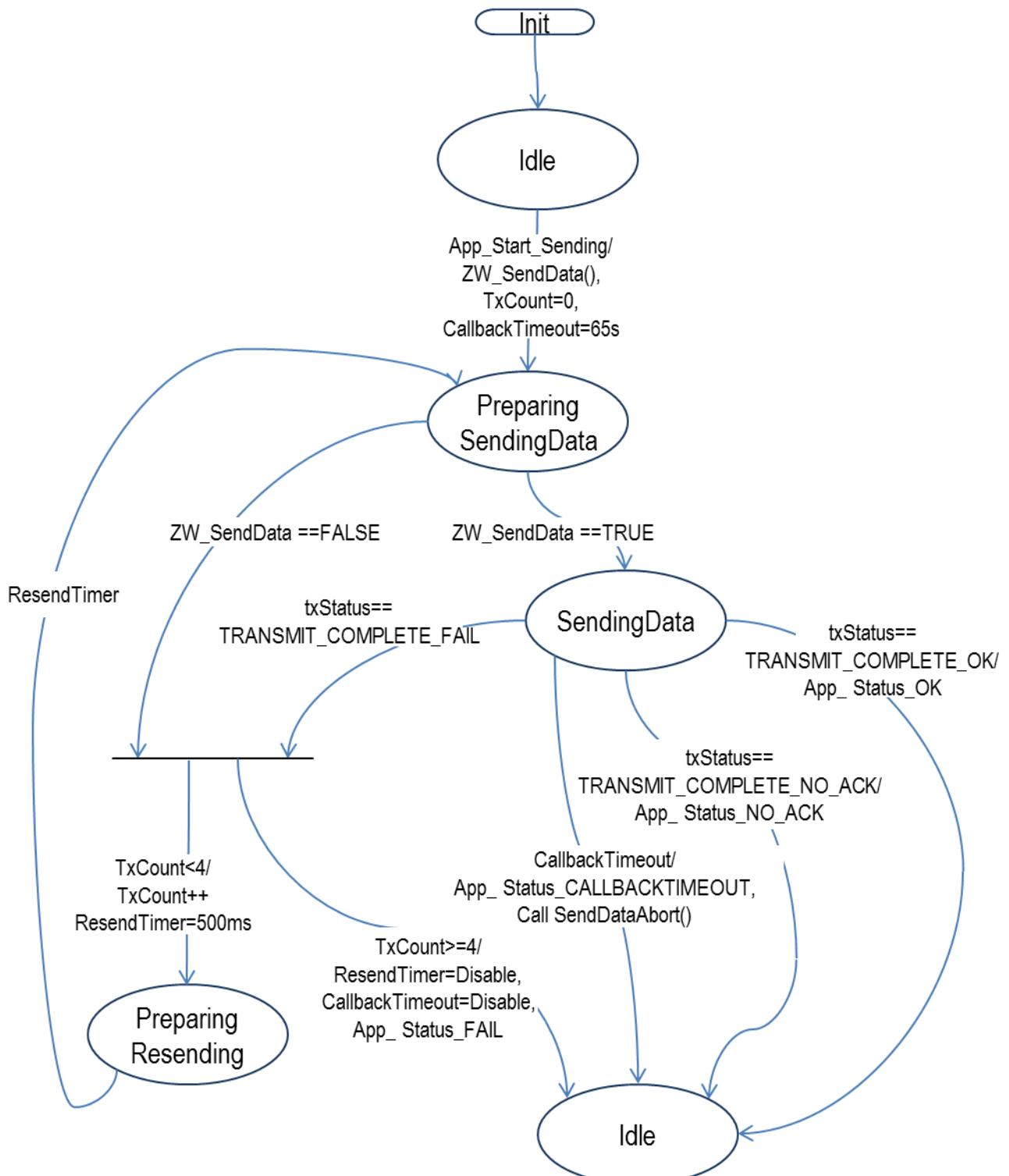
Figure 9. Application state machine for ZW_SendData

CONFIDENTIAL

Table 13. ZW_SendData : State/Event processing

Idle	Waiting for events
	
	Event: (Init) => // Initialize timers, etc.
	
	Event: App_Start_Sending => // Higher layer application event calls for data to be sent
	        New state: <PreparingSendingData>
	        Actions: Call ZW_SendData()
	                Reset retransmission counter TxCount=0
	                Set CallbackTimeout=65s

PreparingSendingData	Waiting for events
	
	Event: ZW_SendData()==FALSE => // Transmitter queue is full. Transmission is not attempted
	        New state: <PreparingResending>
	        Actions: IF (TxCount<4) THEN
	                Increment TxCount retransmission counter
	                Preset retransmission delay timer ResendTimer to 500ms
	            ELSE
	                Disable retransmission delay timer ResendTimer
	                Disable callback timeout timer CallbackTimeout
	                Generate App_Status_FAIL event for application
	            ENDIF
	        // App_Status_FAIL SHOULD cause application to report to user that
	        // that transmission was not possible, RF media may be jammed
	
	Event: ZW_SendData()==TRUE => // Transmitter is starting; await callback events
	        New state: <SendingData>
	        Actions: (none)

SendingData	Waiting for events
	
	Event: txStatus==TRANSMIT_COMPLETE_OK => // Callback
	        New state: <Idle>
	        Actions: Generate App_Status_OK event for application
	
	Event: txStatus==TRANSMIT_COMPLETE_NO_ACK => // Callback
	        New state: <Idle>
	        Actions: Generate App_Status_NO_ACK event for application
	        // App_Status_NO_ACK SHOULD cause application to report to user that
	        // that transmission failed, Destination may be unreachable.
	
	Event: CALLBACKTIMEOUT => // Timer event
	        // This is an exception that should never happen.
	        New state: <Idle>
	        Actions: Generate App_Status_CALLBACKTIMEOUT event for application
	                Call ZW_SendDataAbort()
	        // Recommended application response is hard reset target via watchdog timeout.
	        // The state machine may receive one or more SendDataAbort callback events
	        // after entering the <Idle> state. These events must be ignored.

PreparingResending	Waiting for events
	
	Event: ResendTimer => // Timer event
	        New state: <PreparingSendingData>
	        Actions: (none)

CONFIDENTIAL

4.4.3.1.7   Serial API function prototypes

HOST->ZW: REQ | 0x13 | nodeID | dataLength | pData[ ] | txOptions | funcID

ZW->HOST: RES | 0x13 | RetVal

ZW->HOST: REQ | 0x13 | funcID | txStatus

4.4.3.2   ZW_SendData_Generic

BYTE ZW_SendData_Generic(BYTE nodeID,
**   BYTE *pData,**
**   BYTE dataLength,**
**   BYTE txOptions,**
**   BYTE_P pRoute,**
**   Void (*completedFunc)(BYTE txStatus))**

NOTE: Not supported by the Bridge Controller library (See ZW_SendData_Bridge). For backward compatibility macros for the supporting libraries has been made for
ZW_SendData(node,data,length,options,func) and
ZW_SEND_DATA(node,data,length,options,func)

Macro: ZW_SEND_DATA_GENERIC(node,data,length,options,pRoute,func)

Transmit the data buffer to a single Z-Wave Node or all Z-Wave Nodes (broadcast). The data buffer is queued to the end of the transmit queue (first in; first out) and when ready for transmission the Z-Wave protocol layer frames the data with a protocol header in front and a checksum at the end.

The transmit option TRANSMIT_OPTION_ACK requests the destination node to return a transfer acknowledge to ensure proper transmission. The transmitting node will retry the transmission if no acknowledge received. The Controller nodes can add the TRANSMIT_OPTION_AUTO_ROUTE flag to the transmit option parameter. The Controller will then try transmitting the frame via repeater nodes if the direct transmission failed.

The transmit option TRANSMIT_OPTION_NO_ROUTE force the protocol to send the frame without routing, even if a response route exist.

The Routing Slave and Enhanced Slave nodes can add the TRANSMIT_OPTION_AUTO_ROUTE flag to the transmit option parameter. This flag informs the Enhanced/Routing Slave protocol that the frame about to be transmitted should use the assigned return routes for the concerned nodeID (if any). The node will then try to use one of the return routes assigned (if a route is unsuccessful the next route is used and so on), if no routes are valid then transmission will try direct (no route) to nodeID. If the nodeID = NODE_BROADCAST then the frame will be transmitted to all assigned return route destinations. If nodeID != NODE_BROADCAST then the frame will be transmitted to nodeID using the assigned return routes for nodeID.

To enable on-demand route resolution a new transmit option TRANSMIT_OPTION_EXPLORE must be appended to the well known send API calls. This instruct the protocol to transmit the frame as an explore frame to the destination node if source routing fails. An explore frame uses normal RF power level minus 6dB similar to a node finding neighbors. It is also possible to specify the maximum number of source routing attempts before the explorer frame kicks in using the API call ZW_SetRoutingMAX. Default value is five with respect to maximum number of source routing attempts. A SDK 4.5 controller uses the routing

CONFIDENTIAL

algorithm from 5.02 to address nodes from SDK's not supporting explorer frame. When communicating with nodes, which do not support on-demand route resolution the transmit option flag TRANSMIT_OPTION_EXPLORE is ignored. Notice that an explorer frame cannot wake up FLiRS nodes.

The **completedFunc** is called when the frame transmission completes, that is when transmitted if ACK is not requested; when acknowledge received from the destination node, or when routed acknowledge completed if the frame was transmitted via one or more repeater nodes. The transmit status TRANSMIT_COMPLETE_NO_ACK indicate that no acknowledge is received from the destination node. The transmit status TRANSMIT_COMPLETE_FAIL indicate that the Z-Wave network is busy (jammed).

The TRANSMIT_OPTION_LOW_POWER option should only be used when the two nodes that are communicating are close to each other (<2 meter). In all other cases this option should **not** be used.

NOTE: Allways use the completeFunc callback to determine when the transmit is done. The completeFunc should flag the application state machine that the transmit has been done and next state/action can be started. A frame transmit should always be started through the application state machine in order to be sure that the transmit buffer is ready for sending next frame. Calling the ZW_SendData_Generic in a loop without using the completeFunc callback will overflow the transmit queue and eventually fail. The payload data buffer in the application must not be changed before completeFunc callback is received because it is only the pointer that is passed to the transmit queue.

CONFIDENTIAL

Defined in: ZW_transport_api.h

Return value:

BYTE	FALSE	If transmit queue overflow

Parameters:

nodeID IN	Destination node ID (NODE_BROADCAST == all nodes)	The frame will also be transmitted in case the source node ID is equal destination node ID
pData IN	Data buffer pointer	
dataLength IN	Data buffer length	The maximum dataLength field depends on the transmit options and whether a non-secure/secure frame is used. For details, see section 3.4. The payload must be minimum one byte.
txOptions IN	Transmit option flags:	
	TRANSMIT_OPTION_LOW_POWER	Transmit at low output power level (1/3 of normal RF range).
	TRANSMIT_OPTION_NO_ROUTE	Only send this frame directly, even if a response route exist
	TRANSMIT_OPTION_EXPLORE	Transmit frame as an Explore frame if all else fails.
	TRANSMIT_OPTION_ACK	Request acknowledge from destination node.
	TRANSMIT_OPTION_AUTO_ROUTE	Controllers: Request retransmission via repeater nodes (at normal output power level). Number of max routes can be set using **ZW_SetRoutingMax** Routing and Enhanced Slaves: Send the frame to nodeID using the return routes assigned for nodeID to the routing/enhanced slave, if no routes are valid then transmit directly to nodeID (if nodeID = NODE_BROADCAST then the frame will be a BROADCAST). If return routes exists and the nodeID = NODE_BROADCAST then the frame will be transmitted to all assigned return route destinations. If nodeID != NODE_BROADCAST then the frame will be transmitted via the assigned return routes for nodeID.

CONFIDENTIAL

pRoute IN   Pointer to byte array (4 bytes) containing up to 4 hop node IDs. Zero indicates end of source route if the route is shorter than 4 hops. NULL indicates that the internal source routing mechanism is used.

The call will only try the proposed source route when pRoute is different from NULL. Ignores the transmit options TRANSMIT_OPTION_AUTO_ROUTE if pRoute is different from NULL.

Format of data:

| Repeater 1 |
| Repeater 2 |
| Repeater 3 |
| Repeater 4 |

NOTE: On controllers the protocol will calculate the speed used for the route and in routing/enhanced slaves the route will always use 9.6kbps

completedFunc   Transmit completed call back function

Callback function Parameters:

txStatus   Transmit completion status:

TRANSMIT_COMPLETE_OK	Successfully
TRANSMIT_COMPLETE_NO_ACK	No acknowledge is received before timeout from the destination node. Acknowledge is discarded in case it is received after the timeout. Also used when transmission of a routed frame fails between Source node and hop 1.
TRANSMIT_COMPLETE_FAIL	Not possible to transmit data because the Z-Wave network is busy (jammed).
TRANSMIT_COMPLETE_HOP_1_FAIL	Transmission between hop 1 and hop 2 failed. Only detected in case a Routed Error is returned to source node.
TRANSMIT_COMPLETE_HOP_2_FAIL	Transmission between hop 2 and hop 3 failed. Only detected in case a Routed Error is returned to source node.
TRANSMIT_COMPLETE_HOP_3_FAIL	Transmission between hop 3 and hop 4 failed. Only detected in case a Routed Error is returned to source node.
TRANSMIT_COMPLETE_HOP_4_FAIL	Transmission between hop 4 and destination node failed. Only detected in case a Routed Error is returned to source node.

Timeout: 65s

Exception recovery: If a timeout occurs, it is important to call ZW_SendDataAbort to stop the seding of the frame.

CONFIDENTIAL

Serial API:

HOST->ZW: REQ | 0x19 | nodeID | dataLength | pData[ ] | txOptions | pRoute[4] | funcID

ZW->HOST: RES | 0x19 | RetVal

ZW->HOST: REQ | 0x19 | funcID | txStatus

CONFIDENTIAL

4.4.3.3   ZW_SendData_Bridge

BYTE ZW_SendData_Bridge(BYTE srcNodeID,
**   BYTE destNodeID,**
**   BYTE *pData,**
**   BYTE dataLength,**
**   BYTE txOptions,**
**   BYTE_P pRoute,**
**   Void (*completedFunc)(BYTE txStatus))**

NOTE: Only supported by the Bridge Controller library. For backward compatibility macros for the Bridge Controller library has been made for ZW_SendData(node,data,length,options,func) and ZW_SEND_DATA(node,data,length,options,func)

Macro: ZW_SEND_DATA_BRIDGE(srcnodeid,destnodeid,data,length,options,pRoute,func)

Transmit the data buffer to a single Z-Wave Node or all Z-Wave Nodes (broadcast). The data buffer is queued to the end of the transmit queue (first in; first out) and when ready for transmission the Z-Wave protocol layer frames the data with a protocol header in front and a checksum at the end.

The transmit option TRANSMIT_OPTION_ACK requests the destination node to return a transfer acknowledge to ensure proper transmission. The transmitting node will retry the transmission if no acknowledge received. The Controller nodes can add the TRANSMIT_OPTION_AUTO_ROUTE flag to the transmit option parameter. The Controller will then try transmitting the frame via repeater nodes if the direct transmission failed.

The transmit option TRANSMIT_OPTION_NO_ROUTE force the protocol to send the frame without routing, even if a response route exist.

To enable on-demand route resolution a new transmit option TRANSMIT_OPTION_EXPLORE must be appended to the well known send API calls. This instruct the protocol to transmit the frame as an explore frame to the destination node if source routing fails. An explore frame uses normal RF power level minus 6dB similar to a node finding neighbors. It is also possible to specify the maximum number of source routing attempts before the explorer frame kicks in using the API call ZW_SetRoutingMAX. Default value is five with respect to maximum number of source routing attempts. A SDK 4.5 controller uses the routing algorithm from 5.02 to address nodes from SDK's not supporting explorer frame. When communicating with nodes, which do not support on-demand route resolution the transmit option flag TRANSMIT_OPTION_EXPLORE is ignored. Notice that an explorer frame cannot wake up FLiRS nodes.

The **completedFunc** is called when the frame transmission completes, that is when transmitted if ACK is not requested; when acknowledge received from the destination node, or when routed acknowledge completed if the frame was transmitted via one or more repeater nodes. The transmit status TRANSMIT_COMPLETE_NO_ACK indicate that no acknowledge is received from the destination node. The transmit status TRANSMIT_COMPLETE_FAIL indicate that the Z-Wave network is busy (jammed).

The TRANSMIT_OPTION_LOW_POWER option should only be used when the two nodes that are communicating are close to each other (<2 meter). In all other cases this option should **not** be used.

NOTE: Allways use the completeFunc callback to determine when the transmit is done. The completeFunc should flag the application state machine that the transmit has been done and next state/action can be started. A frame transmit should always be started through the application state machine in order to be sure that the transmit buffer is ready for sending next frame. Calling the ZW_SendData_Bridge in a loop without using the completeFunc callback will overflow the transmit queue and eventually fail. The payload data buffer in the application must not be changed before completeFunc callback is received because it is only the pointer that is passed to the transmit queue.

CONFIDENTIAL

Defined in: ZW_transport_api.h

Return value:

BYTE	FALSE	If transmit queue overflow

Parameters:

srcNodeID IN	Source node ID. Valid values:	
	NODE_BROADCAST = Bridge Controller NodeID.	
	Bridge Controller NodeID.	
	Virtual Slave NodeID (only existing Virtual Slave NodeIDs).	
destNodeID IN	Destination node ID (NODE_BROADCAST == all nodes)	The frame will also be transmitted in case the source node ID is equal destination node ID
pData IN	Data buffer pointer	
dataLength IN	Data buffer length	The maximum dataLength field depends on the transmit options and whether a non-secure/secure frame is used. For details, see section 3.4. The payload must be minimum one byte.
txOptions IN	Transmit option flags:	
	TRANSMIT_OPTION_LOW_POWER	Transmit at low output power level (1/3 of normal RF range).
	TRANSMIT_OPTION_NO_ROUTE	Only send this frame directly, even if a response route exist
	TRANSMIT_OPTION_EXPLORE	Transmit frame as an Explore frame if all else fails
	TRANSMIT_OPTION_ACK	Request acknowledge from destination node.
	TRANSMIT_OPTION_AUTO_ROUTE	Request retransmission via repeater nodes (at normal output power level).
pRoute IN	Pointer to byte array (4 bytes) containing up to 4 hop node IDs. Zero indicates end of source route. NULL indicates that the internal source routing mechanism is used.	The call will only try the proposed source route when pRoute is different from NULL. Ignores the transmit options TRANSMIT_OPTION_AUTO_ROUTE if pRoute is different form NULL.
completedFunc	Transmit completed call back function	

CONFIDENTIAL

Callback function Parameters:

txStatus   Transmit completion status:

TRANSMIT_COMPLETE_OK	Successfully
TRANSMIT_COMPLETE_NO_ACK	No acknowledge is received before timeout from the destination node. Acknowledge is discarded in case it is received after the timeout.
TRANSMIT_COMPLETE_FAIL	Not possible to transmit data because the Z-Wave network is busy (jammed).
TRANSMIT_COMPLETE_HOP_0_FAIL	Transmission between Source node and hop 1 failed.
TRANSMIT_COMPLETE_HOP_1_FAIL	Transmission between hop 1 and hop 2 failed. Only detected in case a Routed Error is returned to source node.
TRANSMIT_COMPLETE_HOP_2_FAIL	Transmission between hop 2 and hop 3 failed. Only detected in case a Routed Error is returned to source node.
TRANSMIT_COMPLETE_HOP_3_FAIL	Transmission between hop 3 and hop 4 failed. Only detected in case a Routed Error is returned to source node.
TRANSMIT_COMPLETE_HOP_4_FAIL	Transmission between hop 4 and destination node failed. Only detected in case a Routed Error is returned to source node.

Timeout: 65s

Exception recovery: If a timeout occur, it is important to call ZW_SendDataAbort to stop the seding of the frame.

Serial API:

HOST->ZW: REQ | 0xA9 | srcNodeID | destNodeID | dataLength | pData[ ] | txOptions | pRoute[4] | funcID

ZW->HOST: RES | 0xA9 | RetVal

ZW->HOST: REQ | 0xA9 | funcID | txStatus

4.4.3.4   ZW_SendDataMeta_Generic

BYTE ZW_SendDataMeta_Generic(BYTE destNodeID,
**   BYTE *pData,**
**   BYTE dataLength,**
**   BYTE txOptions,**
**   BYTE *pRoute,**
**   Void (*completedFunc)(BYTE txStatus))**

Macro: ZW_SEND_DATA_META_GENERIC(destnodeid,data,length,options,proute,func)

NOTE: This function is not available in the Bridge Controller libraries.

Transmit streaming or bulk data in the Z-Wave network. The application must implement a delay of minimum 35ms after each ZW_SendDataMeta_Generic call to ensure that streaming data traffic do not prevent control data from getting through in the network. Both slaves and controllers supporting 40kbps can use the API call ZW_SendDataMeta_Generic. The call also checks that the destination supports 40kbps except if it is a source based on a slave. Routing can use both 40kbps and 9.6kbps hops.

NOTE: The **completedFunc** is called when the frame transmission completes in the case that ACK is not requested; When TRANSMIT_OPTION_ACK is requested the callback function is called when frame has been acknowledged or all transmission attempts are exausted.

The transmit status TRANSMIT_COMPLETE_NO_ACK indicate that no acknowledge is received from the destination node. The transmit status TRANSMIT_COMPLETE_FAIL indicate that the Z-Wave network is busy (jammed).

The Routing Slave and Enhanced Slave nodes can add the TRANSMIT_OPTION_AUTO_ROUTE flag to the transmit option parameter. This flag informs the Enhanced/Routing Slave protocol that the frame about to be transmitted should use the assigned return routes for the concerned nodeID (if any). The node will then try to use one of the return routes assigned (if a route is unsuccessful the next route is used and so on), if no routes are valid then transmission will try direct (no route) to nodeID. If the nodeID = NODE_BROADCAST then the frame will be transmitted to all assigned return route destinations. If nodeID != NODE_BROADCAST then the frame will be transmitted to nodeID using the assigned return routes for nodeID.

NOTE: Allways use the completeFunc callback to determine when the transmit is done. The completeFunc should flag the application state machine that the transmit has been done and next state/action can be started. A frame transmit should always be started through the application state machine in order to be sure that the transmit buffer is ready for sending next frame. Calling the ZW_SendDataMeta_Generic in a loop without using the completeFunc callback will overflow the transmit queue and eventually fail. The payload data buffer in the application must not be changed before completeFunc callback is received because it is only the pointer that is passed to the transmit queue.

   Defined in: ZW_transport_api.h

   Return value:

   BYTE   FALSE   If transmit queue overflow or if
                                                                                              destination node is not 40kbit/s
                                                                                              compatible

CONFIDENTIAL

Parameters:

destNodeID	IN Destination node ID	Node to send Meta data to. Should be 40kbit/s capable
pData	IN Data buffer pointer	Pointer to data buffer.
dataLength	IN Data buffer length	Length of buffer
txOptions	IN Transmit option flags:	The maximum dataLength field depends on the transmit options and whether a non-secure/secure frame is used. For details, see section 3.4. The payload must be minimum one byte.
	TRANSMIT_OPTION_LOW_POWER	Transmit at low output power level (1/3 of normal RF range).
	TRANSMIT_OPTION_EXPLORE	Transmit frame as an Explore frame if all else fails
	TRANSMIT_OPTION_ACK	Request the destination node to acknowledge the frame
	TRANSMIT_OPTION_AUTO_ROUTE	<u>Controllers:</u> Request retransmission via repeater nodes (at normal output power level). Number of max routes can be set using **ZW_SetRoutingMax** <u>Routing and Enhanced Slaves:</u> Send the frame to nodeID using the return routes assigned for nodeID to the routing/enhanced slave, if no routes are valid then transmit directly to nodeID (if nodeID = NODE_BROADCAST then the frame will be a BROADCAST). If return routes exists and the nodeID = NODE_BROADCAST then the frame will be transmitted to all assigned return route destinations. If nodeID != NODE_BROADCAST then the frame will be transmitted via the assigned return routes for nodeID.
pRoute IN	Pointer to byte array (4 bytes) containing up to 4 hop node IDs. Zero indicates end of source route. NULL indicates that the internal source routing mechanism is used.	The call will only try the proposed source route when pRoute is different from NULL. Ignores the transmit options TRANSMIT_OPTION_AUTO_ROUTE if pRoute is different form NULL.
completedFunc	Transmit completed call back function	

CONFIDENTIAL

Callback function Parameters:

txStatus IN   (see **ZW_SendData**)

Timeout: 65s

Exception recovery: If a timeout occurs, it is important to call ZW_SendDataAbort to stop the seding of the frame.

Serial API (Serial API protocol version 4):

HOST->ZW: REQ | 0x1A | destNodeID | dataLength | pData[ ] | txOptions | pRoute[4] | funcID

ZW->HOST: RES | 0x1A | RetVal

ZW->HOST: REQ | 0x1A | funcID | txStatus

4.4.3.5   ZW_SendDataMeta_Bridge

BYTE ZW_SendDataMeta_Bridge(BYTE srcNodeID,
**   BYTE destNodeID,**
**   BYTE *pData,**
**   BYTE dataLength,**
**   BYTE txOptions,**
**   BYTE *pRoute,**
**   Void (*completedFunc)(BYTE txStatus))**

Macro: ZW_SEND_DATA_META_BRIDGE(srcnodeid, nodeid, data, length, options, proute, func)

NOTE: This function is only available in the Bridge Controller library.

Transmit streaming or bulk data in the Z-Wave network. The application must implement a delay of minimum 35ms after each ZW_SendDataMeta_Bridge call to ensure that streaming data traffic do not prevent control data from getting through in the network. Both virtual slaves and the bridge controller id can use the API call ZW_SendDataMeta_Bridge. The call checks that the destination supports 40kbps and denies transmission if destination is 9.6kbps only. Both 40kbps and 9.6kbps hops are allowed in case routing is necessary.

NOTE: The **completedFunc** is called when the frame transmission completes in the case that ACK is not requested; When TRANSMIT_OPTION_ACK is requested the callback function is called when frame has been acknowledged or all transmission attempts are exausted.

The transmit status TRANSMIT_COMPLETE_NO_ACK indicate that no acknowledge is received from the destination node. The transmit status TRANSMIT_COMPLETE_FAIL indicate that the Z-Wave network is busy (jammed).

NOTE: Allways use the completeFunc callback to determine when the transmit is done. The completeFunc should flag the application state machine that the transmit has been done and next state/action can be started. A frame transmit should always be started through the application state

CONFIDENTIAL

machine in order to be sure that the transmit buffer is ready for sending next frame. Calling the ZW_SendDataMeta_Bridge in a loop without using the completeFunc callback will overflow the transmit queue and eventually fail. The payload data buffer in the application must not be changed before completeFunc callback is received because it is only the pointer that is passed to the transmit queue.

Defined in: ZW_transport_api.h

Return value:

BYTE	FALSE	If transmit queue overflow or if destination node is not 40kbit/s compatible

CONFIDENTIAL

Parameters:

srcNodeID IN   Source node ID. Valid values:

NODE_BROADCAST = Bridge
Controller NodeID.

Bridge Controller NodeID.

Virtual Slave NodeID (only existing
Virtual Slave NodeIDs).

destNodeID	IN Destination node ID	Node to send Meta data to. Should be 40kbit/s capable
pData	IN Data buffer pointer	Pointer to data buffer.
dataLength	IN Data buffer length	Length of buffer
txOptions	IN Transmit option flags:	The maximum dataLength field depends on the transmit options and whether a non-secure/secure frame is used. For details, see section 3.4. The payload must be minimum one byte.
	TRANSMIT_OPTION_LOW_POWER	Transmit at low output power level (1/3 of normal RF range).
	TRANSMIT_OPTION_EXPLORE	Transmit frame as an Explore frame if all else fails
	TRANSMIT_OPTION_ACK	Request the destination node to acknowledge the frame
	TRANSMIT_OPTION_AUTO_ROUTE	Request retransmission on single cast frames via repeater nodes (at normal output power level)
pRoute IN	Pointer to byte array (4 bytes) containing up to 4 hop node IDs. Zero indicates end of source route. NULL indicates that the internal source routing mechanism is used.	The call will only try the proposed source route when pRoute is different from NULL. Ignores the transmit options TRANSMIT_OPTION_AUTO_ROUTE if pRoute is different form NULL.
completedFunc	Transmit completed call back function	

Callback function Parameters:

txStatus IN   (see **ZW_SendData**)

Timeout: 65s

Exception recovery: If a timeout occurs, it is important to call ZW_SendDataAbort to stop the seding of the frame.

Serial API (Serial API protocol version 4):

HOST->ZW: REQ | 0xAA | srcNodeID | destNodeID | dataLength | pData[ ] | txOptions | pRoute[4] | funcID

ZW->HOST: RES | 0xAA | RetVal

ZW->HOST: REQ | 0xAA | funcID | txStatus

4.4.3.6   ZW_SendDataMulti

BYTE ZW_SendDataMulti(BYTE *pNodeIDList,
**   BYTE *pData,**
**   BYTE dataLength,**
**   BYTE txOptions,**
**   Void (*completedFunc)(BYTE txStatus))**

Macro: ZW_SEND_DATA_MULTI(nodelist,data,length,options,func)

NOTE: This function is not available in the Bridge Controller library (See ZW_SendDataMulti_Bridge).

Transmit the data buffer to a list of Z-Wave Nodes (multicast frame). If the transmit optionflag TRANSMIT_OPTION_ACK is set the data buffer is also sent as a singlecast frame to each of the Z-Wave Nodes in the node list.

The **completedFunc** is called when the frame transmission completes in the case that ACK is not requested; When TRANSMIT_OPTION_ACK is requested the callback function is called when all single casts have been transmitted and acknowledged.

 The transmit status TRANSMIT_COMPLETE_NO_ACK indicate that no acknowledge is received from the destination node. The transmit status TRANSMIT_COMPLETE_FAIL indicate that the Z-Wave network is busy (jammed). The data pointed to by pNodeIDList should not be changed before the callback is called.

NOTE: Allways use the completeFunc callback to determine when the next frame can be send. Calling the ZW_SendData or ZW_SendDataMulti in a loop without checking the completeFunc callback will overflow the transmit queue and eventually fail. The data buffer in the application must not be changed before completeFunc callback is received because it is only the pointer there is passed to the transmit queue.

Defined in: ZW_transport_api.h

Return value:

BYTE   FALSE   If transmit queue overflow

CONFIDENTIAL

Parameters:

pNodeIDList	IN	List of destination node ID's	This is a fixed length bit-mask.
Pdata	IN	Data buffer pointer	
DataLength	IN	Data buffer length	The maximum size of a packet is 64 bytes. The protocol header, multicast addresses and checksum takes 39 bytes in a multicast frame leaving 25 bytes for the payload. In case routed single casts follow multicast the source routing info takes up to 6 bytes depending on the number of hops leaving minimum 19 bytes for the payload. In case it is a singlecast, which piggyback on an explorer frame overhead is 8 bytes leaving minimum 17 bytes for the payload. The payload must be minimum one byte.

TxOptions	IN Transmit option flags:	
	TRANSMIT_OPTION_LOW_POWER	Transmit at low output power level (1/3 of normal RF range).
	TRANSMIT_OPTION_EXPLORE	If TRANSMIT_OPTION_ACK is set the will make the node try sending as an Explore frame if all else fails when doing the single cast transmits
	TRANSMIT_OPTION_ACK	The multicast frame will be followed by a number of single cast frames to each of the destination nodes and request acknowledge from each destination node.
	TRANSMIT_OPTION_AUTO_ROUTE **(Controller API only)**	Request retransmission on single cast frames via repeater nodes (at normal output power level)

completedFunc	Transmit completed call back function

Callback function Parameters:

txStatus IN	(see **ZW_SendData**)

Timeout: 65s*numberOfDestinations

Exception recovery: If a timeout occurs, it is important to call ZW_SendDataAbort to stop the seding of the frame.

CONFIDENTIAL

Serial API:

HOST->ZW: REQ | 0x14 | numberNodes | pNodeIDList[ ] | dataLength | pData[ ] | txOptions | funcId

ZW->HOST: RES | 0x14 | RetVal

ZW->HOST: REQ | 0x14 | funcId | txStatus

4.4.3.7   ZW_SendDataMulti_Bridge

BYTE ZW_SendDataMulti_Bridge(BYTE srcNodeID,
**   BYTE *pNodeIDList,**
**   BYTE *pData,**
**   BYTE dataLength,**
**   BYTE txOptions,**
**   Void (*completedFunc)(BYTE txStatus))**

Macro: ZW_SEND_DATA_MULTI_BRIDGE(srcnodid,nodelist,data,length,options,func)

NOTE: This function is only available in the Bridge Controller library.

Transmit the data buffer to a list of Z-Wave Nodes (multicast frame). If the transmit optionflag TRANSMIT_OPTION_ACK is set the data buffer is also sent as a singlecast frame to each of the Z-Wave Nodes in the node list.

The **completedFunc** is called when the frame transmission completes in the case that ACK is not requested; When TRANSMIT_OPTION_ACK is requested the callback function is called when all single casts have been transmitted and acknowledged.

 The transmit status TRANSMIT_COMPLETE_NO_ACK indicate that no acknowledge is received from the destination node. The transmit status TRANSMIT_COMPLETE_FAIL indicate that the Z-Wave network is busy (jammed). The data pointed to by pNodeIDList should not be changed before the callback is called.

NOTE: Allways use the completeFunc callback to determine when the next frame can be send. Calling the ZW_SendData_Bridge or ZW_SendDataMulti_Bridge in a loop without checking the completeFunc callback will overflow the transmit queue and eventually fail. The data buffer in the application must not be changed before completeFunc callback is received because it's only the pointer there is passed to the transmit queue.

    Defined in: ZW_transport_api.h

    Return value:

    BYTE   FALSE   If transmit queue overflow

CONFIDENTIAL

Parameters:

srcNodeID IN   Source node ID. Valid values:

NODE_BROADCAST = Bridge
Controller NodeID.

Bridge Controller NodeID.

Virtual Slave NodeID (only existing
Virtual Slave NodeIDs).

pNodeIDList	IN List of destination node ID's	This is a fixed length bit-mask.
Pdata	IN Data buffer pointer	
DataLength	IN Data buffer length	The maximum size of a packet is 64 bytes. The protocol header, multicast addresses and checksum takes 39 bytes in a multicast frame leaving 25 bytes for the payload. In case routed single casts follow multicast the source routing info takes up to 6 bytes depending on the number of hops leaving minimum 19 bytes for the payload. In case it is a singlecast, which piggyback on an explorer frame overhead is 8 bytes leaving minimum 17 bytes for the payload. The payload must be minimum one byte.

TxOptions   IN Transmit option flags:

TRANSMIT_OPTION_LOW_POWER	Transmit at low output power level (1/3 of normal RF range).
TRANSMIT_OPTION_EXPLORE	If TRANSMIT_OPTION_ACK is set the will make the node try sending as an Explore frame if all else fails when doing the single cast transmits
TRANSMIT_OPTION_ACK	The multicast frame will be followed by a number of single cast frames to each of the destination nodes and request acknowledge from each destination node.
TRANSMIT_OPTION_AUTO_ROUTE	Request retransmission on single cast frames via repeater nodes (at normal output power level)

completedFunc   Transmit completed call back function

Callback function Parameters:

txStatus IN   (see **ZW_SendData**)

CONFIDENTIAL

Timeout: 65s

Exception recovery: If a timeout occurs, it is important to call ZW_SendDataAbort to stop the seding of the frame.

Serial API:

HOST->ZW: REQ | 0xAB | srcNodeID | numberNodes | pNodeIDList[ ] | dataLength | pData[ ] | txOptions | funcId

ZW->HOST: RES | 0xAB | RetVal

ZW->HOST: REQ | 0xAB | funcId | txStatus

4.4.3.8   ZW_SendDataAbort

void ZW_SendDataAbort( )

Macro: ZW_SEND_DATA_ABORT

Abort the ongoing transmit started with **ZW_SendData()** or **ZW_SendDataMulti()**. If an ongoing transmission is aborted, the callback function from the send call will return with the status TRANSMIT_COMPLETE_NO_ACK.

Defined in: ZW_transport_api.h

Serial API:

HOST->ZW: REQ | 0x16

4.4.3.9   ZW_SendConst

void ZW_SendConst(void)

Macro: ZW_SEND_CONST()

This function causes the transmitter to send out a constant signal on a fixed frequency until another RF function is called.

This API call can only be called in production test mode from **ApplicationTestPoll**.

Defined in: ZW_transport_api.h

Serial API (Not supported)

CONFIDENTIAL

4.4.4   Z-Wave TRIAC API

The built-in TRIAC Controller is targeted at TRIAC or FET controlled light / power dimming applications. For a detailed description of the TRIAC refer to [25] or [26] depending on the single chip used. The Application software can use the following TRIAC API calls to control the ZW0201/ZW0301 TRIAC Controller.

4.4.4.1   TRIAC_Init

void TRIAC_Init(BYTE bridgeType,
**   BYTE mainsfreq,**
**   BYTE voltageDrop,**
**   BYTE minimumPulse)**

Macros:

ZW_TRIAC_INIT

ZW_TRIAC_INIT_2_WIRE

TRIAC_Init initializes the ASIC's integrated TRIAC for mainsfreq AC mains frequency usage, bridge type, voltageDrop is the voltage drop across the ZEROX input and the minimumPulse is the minimum pulse time of the TRIAC output signal. Configures the ZEROX and TRIAC pins for triac control.

The purpose of the voltageDrop parameter is to adjust when to fire the TRIAC pulse in the negative half period. This is due to the threshold level of the circuit connected to the the ZEROX pin generating a difference between the negative and positive half periods with respect to time. The values for voltageDrop are between 0mV - 2000mV in 100mV intervals.

In half bridge mode can the fire angle be different for the positive and the negative half period. Sometimes it is not possible to see the fire pulse in the positive half period. This can be because the duty cycle of the zero-cross is not 50%/50%. Adjust the voltageDrop parameter to obtain the same positive and the negative half period.

The minimumPulse defines the minimum length of the TRIAC pulse. Refer to the datasheet of the specific Triac to see how long time the Triac gate pulse must be to guarantee that the Triac starts to conduct. The minimumPulse can have the value from 64μs to 512μs with 64μs intervals.

CONFIDENTIAL

Defined in: ZW_triac_api.h

Parameters:

bridgeType IN   Bridge types:

	TRIAC_FULLBRIDGE	The TRIAC signal is triggered ONLY on the rising edge of the ZEROX signal which is fed through a FULL diode bridge.
	TRIAC_HALFBRIDGE	The TRIAC signal is triggered on the rising AND the falling edge of the ZEROX signal which is fed through a NON-FULL diode bridge.

mainsfreq IN   AC Mains frequencies:

	FREQUENCY_50HZ	Controlling a 50Hz AC mains supply
	FREQUENCY_60HZ	Controlling a 60Hz AC mains supply

voltageDrop IN	Valid values for voltageDrop are from 0 to 2000mv with 100mv step.	The voltage drop values are defined as constants and are listed in the ZW_triac_api.h header file.
minimumPulse IN	Valid values for the triac minimum pulse are from 64 us to 512 us with 64 us step.	The minimum pulse values are defined as constants and are listed in the ZW_triac_api.h.

Serial API (Not supported)

CONFIDENTIAL

4.4.4.2   TRIAC_SetDimLevel

void TRIAC_SetDimLevel(BYTE dimLevel)

Macros:

ZW_TRIAC_DIM_SET_LEVEL(dimLevel)

ZW_TRIAC_LIGHT_SET_LEVEL(lightLevel)

TRIAC_SetDimLevel turns the triac controller ON and sets it to dim at dimLevel (0-100) or sets the light level to lightLevel (0-100) if the ZW_TRIAC_LIGHT_SET_LEVEL macro is used. This is done for the mains frequency selected in the TRIAC_Init function call (50/60Hz).

   Defined in: ZW_triac_api.h

   Parameters:

   dimLevel IN   Level (0…100)

   Serial API (Not supported)

4.4.4.3   TRIAC_Off

void TRIAC_Off(void)

Macro:

ZW_TRIAC_OFF

TRIAC_Off turns the triac controller OFF.

   Defined in: ZW_triac_api.h

   Serial API (Not supported)

CONFIDENTIAL

4.4.5   Z-Wave Timer API

The timer is based on a "tick-function" that is activated from the RF timer interrupt function every 10 msec. The "tick-function" will handle a global tick counter and a number of active timers. The global tick counter is incremented and the active timers are decremented on each "tick". When an active timer value changes from 1 to 0, the registered timeout function is called. The timeout function is called from the Z-Wave main loop (non-interrupt environment).

The timer implementation is targeted for shorter (second) timeout functionality. The global tick counter and active timers are inaccurate because they stops while changing RF transmission direction and during sleep mode. Therefore the global tick counter and active timers will pick up where they left off when leaving sleep mode.

Global tick counter:

> **WORD tickTime**

4.4.5.1   TimerStart

BYTE TimerStart(VOID_CALLBACKFUNC(func)(), BYTE timerTicks, BYTE repeats)

Macro: ZW_TIMER_START(func,ticks,repeats)

Register a function that is called when the specified time has elapsed. Remember to check if the timer is allocated by testing the return value. The call back function is called "repeats" times before the timer is stopped. It's possible to have up to 5 timers running simultaneously on a slave and 4 timers on a controller. Additional software timers can be implemented by for example using the PWM API as "tick-function".

> Defined in: ZW_timer_api.h

Return value:

BYTE	Timer handle (timer table index). 0xFF is returned if the timer start operation failed.

Parameters:

func IN	Timeout function address (not NULL).
timerTicks IN	Timeout value (value * 10 msec.). Predefined values:
	TIMER_ONE_SECOND
repeats IN	Number of function calls. Max value is 253. Predefined values:
	TIMER_ONE_TIME
	TIMER_FOREVER

CONFIDENTIAL

Serial API (Not supported)

4.4.5.2   TimerRestart

BYTE TimerRestart(BYTE timerHandle)

Macro: ZW_TIMER_RESTART(handle)

Set the specified timer's tick count to the initial value (extend timeout value).

NOTE: There is no protection in the API against calling this function with a wrong handler, so care should be taken not to use a handler of a timer that has already expired or has been canceled.

Defined in: ZW_timer_api.h

Return value:

BYTE   TRUE   If timer restarted

Parameters:

timerHandle IN   Timer to restart

Serial API (Not supported)

4.4.5.3   TimerCancel

BYTE TimerCancel(BYTE timerHandle)

Macro: ZW_TIMER_CANCEL(handle)

Stop the specified timer.

NOTE: There is no protection in the API against calling this function with a wrong handler, so care should be taken not to use a handler of a timer that has already expired.

Defined in: ZW_timer_api.h

Return value:

BYTE   TRUE   If timer cancelled

Parameters:

timerHandle IN   Timer number to stop

Serial API (Not supported)

CONFIDENTIAL

4.4.6   Z-Wave PWM API

The Z-Wave PWM/GP Timer API is an API that grants the application programmer protected access to a ZW0201/ZW0301 PWM or a timer interrupt. The timer, called General Purpose Timer or GP Timer, is a 16 bit reloadable timer. Both the PWM and the GP Timer runs on a clock from a prescaler that can be set to either divide by 4 or divide by 512.

4.4.6.1   ZW_PWMSetup

BYTE ZW_PWMSetup (BYTE bValue)

Macro: ZW_PWM_SETUP(value)

Configures and enables/disables the PWM or the GP Timer. The PWM and the GP Timer functions are mutual exclusive. That is, when the PWM is enabled, the GP Timer cannot be used at the same time and vice versa. When enabled the PWM uses P1_6 as output pin.

Parameters:

bValue IN	TIMER_RUN_BIT (bit 0)	
	0	PWM/GP Timer is inactive and counters are cleared.
	1	PWM/GP Timer is active and counters are enabled.
	PWM_MODE_BIT (bit 1)	
	0	GP Timer mode
	1	PWM mode
	PRESCALER_BIT (bit 2)	
	0	PWM/GP Timer runs with CPU_FREQ/4 speed.
	1	PWM/GP Timer runs with CPU_FREQ/512 speed.
	RELOAD_BIT (bit 3)	
	0	The GP timer stops upon overflow. Not applicable for the PWM
	1	The GP timer reloads its counter registers upon overflow. Not applicable for the PWM
	(bit 4–7) don't care.	

Serial API (Not supported)

CONFIDENTIAL

4.4.6.2   ZW_PWMPrescale

BYTE ZW_PWMPrescale(BYTE bValueMSB, BYTE bValueLSB)

Macro ZW_PWM_PRESCALE(msb,lsb)

This function either sets up the PWM period (PWN mode) or reloads value of the GP timer (Timer mode). Constant CPU_FREQ is defined in Z-Wave header file.

PWM mode:

High time of PWM signal: $t_{hPWM} = (msb * prescaler) / CPU\_FREQ$

Low time of PWM signal $t_{lPWM} = (lsb * prescaler) / CPU\_FREQ$

Total period of PWM signal: $T_{PWM} = t_{hPWM} + t_{lPWM}$
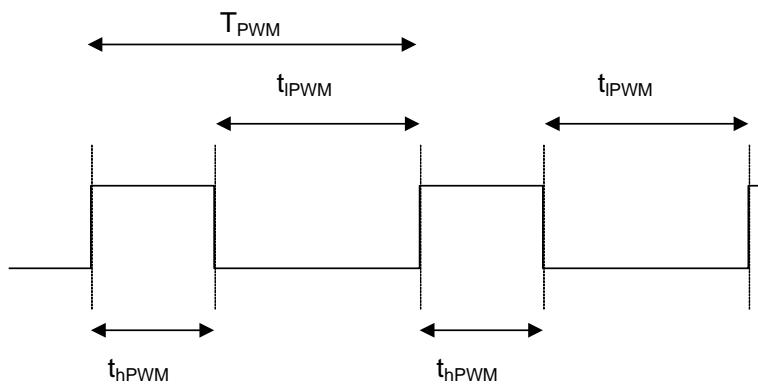
Figure 10. PWM waveform

Timer mode (Interrupt period):

The GP timer is loaded with the reload value when it is enabled, then it decrements the timer, until it underruns, issues an interrupt, reloads the reload value, etc. The resulting interrupt interval is set by:

$T_{int} = (msb * 256 + lsb + 1) * prescaler / CPU\_FREQ$

Defined in: ZW_appltimer_api.h

Parameters:

bValueMSB IN   Used to calculate PWM period and PWM "High" time or interrupt timeout (See above formular).

bValueLSB IN   Used to calculate PWM period and PWM "Low" time or interrupt timeout (See above formular).

Serial API (Not supported)

CONFIDENTIAL

4.4.6.3   ZW_PWMClearInterrupt

BYTE ZW_PWMClearInterrupt(void)

Macro ZW _PWM_CLEAR_INTERRUPT()

Clears the GP Timer interrupt. Must be done by software when servicing interrupt.

  Defined in: ZW_appltimer_api.h

  Serial API (Not supported)

4.4.6.4   ZW_PWMEnable

BYTE ZW_PWMEnable(BOOL bValue)

Macro ZW_PWM_INT_ENABLE (value)

Enables or disables the GP Timer interrupt (ISR number: ZW_PWM).

  Defined in: ZW_appltimer_api.h

  Parameters:

  bValue IN   TRUE   Interrupt enabled.

                     FALSE   Interrupt disabled.

  Serial API (Not supported)

CONFIDENTIAL

4.4.7   Z-Wave Memory API

The memory application interface handles accesses to the application data area in NVM.

Routing slave nodes use flash on 200/300 Series chip for storing application data. The flash area allocated for application data result in typically 320K write cycles for the 200/300 Series chip.

Enhanced slave and all controller nodes use an external EEPROM for storing application data. The Z-Wave protocol uses the first part of the external EEPROM for home ID, node ID, routing table etc. The SPI interface on the ZW0x0x access the external EEPROM. The SPI interface related pins CLK, MOSI, and MISO are only free as GPIO for the application on a routing slave. The external EEPROM typically supports 1000K write cycles.

The memory functions are internally offset by EEPROM_APPL_OFFSET because the addresses between 0x0000 and EEPROM_APPL_OFFSET are used by the protocol. The offset parameter equal to 0x0000 is therefore the first byte of the reserved area for application data.

NOTE: The CPU halts while the API is writing to flash memory, so care should be taken not to write to flash to often.

4.4.7.1   MemoryGetID

void MemoryGetID(BYTE *pHomeID, BYTE *pNodeID)

Macro: ZW_MEMORY_GET_ID(homeID, nodeID)

The **MemoryGetID** function copy the Home-ID and Node-ID from the NVM to the specified RAM addresses.

NOTE: A NULL pointer can be given as the pHomeID parameter if the application is only intereste in reading the Node ID.

  Defined in: ZW_mem_api.h

 Parameters:

  pHomeID OUT   Home-ID pointer

  pNodeID OUT   Node-ID pointer

 Serial API:

  HOST->ZW: REQ | 0x20

  ZW->HOST: RES | 0x20 | HomeId(4 bytes) | NodeId

CONFIDENTIAL

4.4.7.2   MemoryGetByte

BYTE MemoryGetByte(WORD offset)

Macro: ZW_MEM_GET_BYTE(offset)

Read one byte from the NVM

If a write is in progress, the write queue will be checked for the actual data.

  Defined in: ZW_mem_api.h

  Return value:

  BYTE   Data from the application area of the
                      EEPROM

  Parameters:

  offset IN   Application area offset from 0x0000.

  Serial API:

  HOST->ZW: REQ | 0x21 | offset (2 bytes)

  ZW->HOST: RES | 0x21 | RetVal

CONFIDENTIAL

4.4.7.3   MemoryPutByte

BYTE MemoryPutByte(WORD offset, BYTE data)

Macro: ZW_MEM_PUT_BYTE(offset,data)

Write one byte to the application area of the NVM

On controllers and enhanced slaves this function works on EEPROM and it should be considered that the write operation have a somewhat long write time (2-5 msec.).

The data to be written to FLASH are not written immediately to the FLASH. Instead it is saved in a RAM buffer and then written when the RF is not active and when it is more than 200ms the buffer was accessed.

On routing slaves this function works on flash RAM so writing one byte will cause a write to a whole flash page of 128 and 256 bytes for the ZW0102 and ZW0201 consequently. While the write takes place the CPU will be halted in 15-25 msec. and will therefore not be able to execute code or receive frames, so care should be taken not to disrupt radio communication or other time critical functions when using this function.

Defined in: ZW_mem_api.h

Return value:

BYTE   FALSE   If write buffer full.

Parameters:

offset IN   Application area offset from 0x0000.

data IN   Data to store

Serial API:

HOST->ZW: REQ | 0x22 | offset(2bytes) | data

ZW->HOST: RES | 0x22 | RetVal

CONFIDENTIAL

4.4.7.4   MemoryGetBuffer

void MemoryGetBuffer(WORD offset, BYTE *buffer, BYTE length)

Macro: ZW_MEM_GET_BUFFER(offset,buffer,length)

Read a number of bytes from the application area of the EEPROM to a RAM buffer.

If a write operation is in progress the write queue will be checked for the actual data.

Defined in: ZW_mem_api.h

Parameters:

offset IN   Application area offset from 0x0000.

buffer IN   Buffer pointer

length IN   Number of bytes to read

Serial API:

HOST->ZW: REQ | 0x23 | offset(2 bytes) | length

ZW->HOST: RES | 0x23 | buffer[ ]

CONFIDENTIAL

4.4.7.5   MemoryPutBuffer

BYTE MemoryPutBuffer(WORD offset, BYTE *buffer, WORD length,
VOID_CALLBACKFUNC(func)(void))

Macro: ZW_MEM_PUT_BUFFER(offset,buffer,length, func)

Copy number of bytes from a RAM buffer to the application area in the NVM.

The write operation requires some time to complete (2-5msec per byte); therefore the data buffer must be in "static" memory. The data buffer can be reused when the completion callback function is called.

The data to be written to FLASH are not written immediately to the FLASH. Instead it is saved in a RAM buffer and then written when the RF is not active and when it is more than 200ms the buffer was accessed.

If an area is to be set to zero there is no need to specify a buffer, just specify a NULL pointer.

On routing slaves this function works on FLASH so writing will cause a write to a whole FLASH page of 128 and 256 bytes for the ZW0102 and ZW0201 consequently. While the write takes place the CPU will be halted in 15-25 msec. and will therefore not be able to execute code or receive frames, so care should be taken not to disrupt radio communication or other time critical functions when using this function.

Defined in: ZW_mem_api.h

Return value:

BYTE   FALSE   If the buffer put queue is full.

Parameters:

offset IN   Application area offset from 0x0000.

buffer IN   Buffer pointer   If NULL all of the area will be set to 0x00

length IN   Number of bytes to read

func IN   Buffer write completed function pointer

Timeout: 200ms

Exception recovery: Resume normal operation.

CONFIDENTIAL

Serial API:

HOST->ZW: REQ | 0x24 | offset(2bytes) | length(2bytes) | buffer[ ] | funcID

ZW->HOST: RES | 0x24 | RetVal

ZW->HOST: REQ | 0x24 | funcID

SerialAPI Note:

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 approx. 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.

CONFIDENTIAL

4.4.7.6   ZW_EepromInit

BOOL ZW_EepromInit(BYTE *homeID)

Macro: ZW_EEPROM_INIT(HOMEID)

NOTE: This function is only implemented in Z-Wave Controller and Enhanced Slave APIs.

Initialize the external EEPROM by writing zeros to the entire EEPROM. The API then writes the content of homeID if not zero to the home ID address in the external EEPROM.

This API call can only be called in production test mode from **ApplicationTestPoll**.

NOTE: This API call is only meant for small-scale production where pre-programmed EEPROMs or a production EEPROM programmer is not available.

Defined in: ZW_mem_api.h

Return value:

BOOL	TRUE	If the EEPROM initialized successfully
	FALSE	Initialization failed

Parameters:

homeID IN   The home ID to be written to the external
                   EEPROM.

Serial API (Not supported)

4.4.7.7   ZW_MemoryFlush

void ZW_MemoryFlush(void)

Macro: ZW_MEM_FLUSH()

This call writes data immediately to the FLASH from the temporary SRAM buffer.

The data to be written to FLASH are not written immediately to the FLASH. Instead it is saved in a SRAM buffer and then written when the RF is not active and when it is more than 200ms the buffer was accessed. This function can be used to write data immediately to FLASH without waiting for the RF to be idle.

NOTE: This function is only implemented in Routing Slave API libraries because they are the only libaries that use a temporary SRAM buffer. The other libraries use an external EEPROM as NVM . Data is written directly to the EEPROM.

Defined in: ZW_mem_api.h

Serial API (Not supported)

CONFIDENTIAL

4.4.8   Z-Wave ADC API

The ADC API is both a slave and a controller application's interface to the ADC unit in the ZW0201/ZW0301. For a detailed description of the ZW0201/ZW0301 ADC refer to [27].

4.4.8.1   ADC_Off

void ADC_Off( )

Macro: ZW_ADC_OFF

Call turns the power off to the ADC unit.
Units not depending on an operational ADC during sleep (e.g. for keyboard decoding) may save battery lifetime by turning off the ADC before entering sleep mode.

   Defined in: ZW_adcdriv_api.h

   Serial API (Not supported)

4.4.8.2   ADC_Start

void ADC_Start( )

Macro: ZW_ADC_START

Start the conversion process.

   Defined in: ZW_adcdriv_api.h

   Serial API (Not supported)

4.4.8.3   ADC_Stop

void ADC_Stop( )

Macro: ZW_ADC_STOP

Stop the conversion process.

   Defined in: ZW_adcdriv_api.h

   Serial API (Not supported)

CONFIDENTIAL

4.4.8.4   ADC_Init

Void ADC_Init(BYTE mode, BYTE upper_ref, BYTE lower_ref, BYTE pin_en)

Macro: ZW_ADC_INIT (MODE, UPPER_REF, LOWER_REF, INPUT)

Initialize the ADC unit to work in the wanted conversion mode etc. The ADC unit can work in one of two different modes.

The ADC unit has 5 multiplexed inputs. The Upper reference voltage can be set to be VCC, internal bandgab or external voltage on ADC_PIN_1. Lower reference voltage can be set to be either GND or external voltage on pin ADC_PIN_2.

ADC_Init only enable one or more of the I/O pins (ADC_PIN_1, ADC_PIN_2, ADC_PIN_3, ADC_PIN_4 and ADC_BAT (internal "pin")) as ADC inputs pins. No I/O pin that was enabled in the ADC_Init call will be selected as the active ADC input. To select the active ADC input, **ADC_SelectPin** must be called.

Defined in: ZW_adcdriv_api.h

Parameters:

mode IN	The ADC mode to be used the mode value can be on of the following:	
	ADC_SINGLE_MODE	Single conversion mode: The ADC will always stop after one conversion. The ADC should be started each time a conversion is wanted.
	ADC_MULTI_CON_MODE	Multi conversion continues mode: The ADC will always sample the input until the ADC is stopped.
upper_ref IN	The source of the upper reference voltage used for the ADC:	
	ADC_REF_U_EXT	External voltage reference applied on pin P0.0
	ADC_REF_U_BGAB	Internal voltage reference is 1.21V generated internally by a bandgap reference.
	ADC_REF_U_VDD	Internal voltage reference is VDD (supply voltage).
lower_ref IN	The source of the upper reference voltage used for the ADC:	
	ADC_REF_L_EXT	External voltage reference applied on pin P0.1
	ADC_REF_L_VSS	Internal ground.

CONFIDENTIAL

pin_en IN   Enabling of the pins to be used by the
ADC. To enabled pin 1 and pin 3 the
value should be set to:
ADC_PIN_1 | ADC_PIN_3.

ADC_PIN_1 (I/O P0.0)	Equal to ADC0 in hardware documentation.
ADC_PIN_2 (I/O P0.1)	Equal to ADC1 in hardware documentation.
ADC_PIN_3 (I/O P1.0)	Equal to ADC2 in hardware documentation.
ADC_PIN_4 (I/O P1.1)	Equal to ADC3 in hardware documentation.
ADC_PIN_BATT	Using the ADC as battery monitor for battery operated devices. Regarding details refer to [21]

Serial API (Not supported)

CONFIDENTIAL

4.4.8.5   ADC_SelectPin

ADC_SelectPin(BYTE adcPin);

Macro:

ZW_ADC_SELECT_AD1 -select pin 1 as the ADC input

ZW_ADC_SELECT_AD2 - select pin 2 as the ADC input

ZW_ADC_SELECT_AD3 - select pin 3 as the ADC input

ZW_ADC_SELECT_AD4 - select pin 4 as the ADC input

Select a pin to use as the active ADC input.

   Defined in: ZW_adcdriv_api.h

   Parameters:

   adcPin IN   The pin to use as ADC input:

                      ADC_PIN_1

                      ADC_PIN_2

                      ADC_PIN_3

                      ADC_PIN_4

   Serial API (Not supported)

CONFIDENTIAL

4.4.8.6   ADC_Buf

ADC_Buf (BYTE enable);

Macro:

ZW_ADC_BUFFER_ENABLE - Enable the input buffer.

ZW_ADC_BUFFER_DISABLE - Disable the input buffer.

Enable / disable an input buffer between the analog input and the ADC converter. Default is the input buffer disabled. If a high impedance driver is used on the input, this can lower the sample rate. The input buffer can be enabled to achieve high sample rate when using high impedance driver.

   Defined in: ZW_adcdriv_api.h

   Parameters:

   enable IN   Switch the input buffer on/off:

                     TRUE   Switch the input buffer on.

                     FALSE   Switch the input buffer off.

   Serial API (Not supported)

CONFIDENTIAL

4.4.8.7   ADC_SetAZPL

ADC_SetAZPL (BYTE azpl);

Macro: ZW_ADC_SET_AZPL(PERIOD)

Set the length of the ADC sample period. The length of the period depends on the source impedance. Default value is ADC_AZPL_128.

Defined in: ZW_adcdriv_api.h

Parameters:

apzl IN   Length of the ADC auto zero period:

	ADC_AZPL_1024	Set the sample period to 1024 clocks, valid for high impedance sources. Only valid for 8 bit resolution.
	ADC_AZPL_512	Set the sample period to 512 clocks, valid for medium to high impedance sources. Only valid for 8 bit resolution.
	ADC_AZPL_256	Set the sample period to 256 clocks, valid for medium to low impedance sources. Only valid for 8 bit resolution.
	ADC_AZPL_128	Set the sample period to 128 clocks, valid for low impedance sources. Valid for both 8 bit and 12 bit resolution.

Serial API (Not supported)

4.4.8.8   ADC_SetResolution

ADC_SetResolution (BYTE reso);

Macro:

ZW_ADC_RESOLUTION_8 - Set the ADC resolution to 8 bit.

ZW_ADC_RESOLUTION_12 - Set the ADC resolution to 12 bit.

Set the resolution of the ADC. Note: ADC_12_BIT only work in single step mode.

CONFIDENTIAL

Defined in: ZW_adcdriv_api.h

Parameters:

reso IN   The resolution of the ADC

       ADC_8_BIT   Set the ADC resolution to 8 bit.

       ADC_12_BIT   Set the ADC resolution to 12 bit.

Serial API (Not supported)

4.4.8.9   ADC_SetThresMode

ADC_SetThresMode (BYTE thresMode);

Macro:

ZW_ADC_THRESHOLD_UP
ADC fire when input above/equal to the threshold value.

ZW_ADC_THRESHOLD_LO
ADC fire when input below/equal to the threshold value.

Set the ADC threshold type.

Defined in: ZW_adcdriv_api.h

Parameters:

thresMode IN   The ADC threshold mode.

       ADC_THRES_UPPER   The ADC fire when input is above/equal
                                                  to the threshold value.

       ADC_THRES_LOWER   The ADC fire when input is below/equal
                                                  to the threshold value.

Serial API (Not supported)

CONFIDENTIAL

4.4.8.10   ADC_SetThres

void ADC_SetThres(WORD thresHold)

Macro: ZW_ADC_SET_THRESHOLD(THRES)

Set the ADC threshold value. Depending on the threshold mode, the threshold value is used to trigger an event when the sampled value is above/equal or below/equal the threshold value. The event triggered depend on the ADC mode:

Single conversion mode: The ADC interrupt will fire and the ADC will stop converting.

Multi conversion continues mode: The ADC interrupt will fire and the ADC will continue converting.

  Defined in: ZW_adcdriv_api.h

  Parameters:

  thresHold IN   The ADC threshold value, it ranges from
                        0 to 4095.

  Serial API (Not supported)

4.4.8.11   ADC_Int

void ADC_Int(BYTE enable)

Macro:

ZW_ADC_INT_ENABLE

ZW_ADC_INT_DISABLE

Call will enable or disable the ADC interrupt. If enabled an interrupt routine must be defined. Default is the ADC interrupt disabled.

NOTE: If the ADC interrupt is used, then the ADC interrupt flag should be reset before exit of interrupt routine by calling ZW_ADC_CLR_FLAG.

  Defined in: ZW_adcdriv_api.h

  Parameters:

  enable IN   The start of the ADC interrupt routine.

                     TRUE   Enables ADC interrupt.

                     FALSE   Disables ADC interrupt.

  Serial API (Not supported)

CONFIDENTIAL

4.4.8.12　　　　ADC_IntFlagClr

void ADC_IntFlagClr()

Macro: ZW_ADC_CLR_FLAG

Clear the ADC interrupt flag.

Defined in:　　　ZW_adcdriv_api.h

Serial API (Not supported)

4.4.8.13　　　　ADC_GetRes

WORD ADC_GetRes()

Macro: ZW_ADC_GET_READING

The call returns the result of the ADC conversion. The return value is an 8-bit or 12-bit value depending on if the ADC is in 8-bit or 12-bit resolution mode. The call will return the value ADC_NOT_FINISHED in case conversion isn't finished yet.

Defined in:　　　ZW_adcdriv_api.h

Return value:

WORD　　　　　　Returns the unsigned 16-bit value
　　　　　　　　　representing the result of the ADC
　　　　　　　　　conversion.

Serial API (Not supported)

CONFIDENTIAL

4.4.9   Z-Wave Power API

The purpose of the Power API is to define functions that make it easy for the Z-Wave application developer to use the power management capabilities of the ZW0201/ZW0301 ASIC. The API can be used both for slave and controller applications.

4.4.9.1   ZW_SetWutTimeout

void ZW_SetWutTimeout (BYTE wutTimeout)

Macro: ZW_SET_WUT_TIMEOUT(TIME)

ZW_SetWutTimeout initializes the time out value of the wakeup timer used in WUT mode.

   Defined in: ZW_power_api.h

   Parameters:

   wutTimeout IN   The Wakeup Timer timeout value in
                     seconds. 0 = 1 sec.:

   Serial API

   HOST->ZW: REQ | 0xB4 | wutTimeout

CONFIDENTIAL

4.4.10   UART interface API

The serial interface API handles transfer of data via the serial interface (UART – RS232). This serial API support transmissions of either a single byte, or a data buffer. The received characters are read by the application one-by-one.

4.4.10.1   UART_Init

void UART_Init(WORD baudRate)

Macro: ZW_UART_INIT(baud)

Initializes the MCU's integrated UART.

Enables UART transmit and receive, selects 8 data bits and sets the specified baud rate.

   Defined in: ZW_uart_api.h

   Parameters:

   baudRate IN   Baud Rate / 100   ZW0201 support only 9600, 38400 and
                                                             115200 baud.

   Serial API (Not supported)

4.4.10.2   UART_RecStatus

BYTE UART_RecStatus(void)

Macro: ZW_UART_REC_STATUS

Read the UART receive data status

   Defined in: ZW_uart_api.h

   Return value:

   BYTE   TRUE   If data received.

   Serial API (Not supported)

CONFIDENTIAL

4.4.10.3   UART_RecByte

BYTE UART_RecByte(void)

Macro: ZW_UART_REC_BYTE

Function receives a byte over the UART.

This function waits until data received. See also: UART_Read

  Defined in: ZW_uart_api.h

  Return value:

  BYTE   Received data.

  Serial API (Not supported)

4.4.10.4   UART_SendStatus

BYTE UART_SendStatus(void)

Macro: ZW_UART_SEND_STATUS

Read the UART send data status.

  Defined in: ZW_uart_api.h

  Return value:

  BYTE   TRUE   If transmitter busy

  Serial API (Not supported)

CONFIDENTIAL

4.4.10.5   UART_SendByte

void UART_SendByte(BYTE data)

Macro: ZW_UART_SEND_BYTE(data)

Function transmits a byte over the UART.

This function waits to transmit data until data register is free.

  Defined in: ZW_uart_api.h

  Parameters:

  data IN   Data to send.

  Serial API (Not supported)

4.4.10.6   UART_SendNum

void UART_SendNum(BYTE data)

Macro: ZW_UART_SEND_NUM(data)

Converts a byte to a two-byte hexadecimal ASCII representation, and transmits it over the UART.

  Defined in: ZW_uart_api.h

  Parameters:

  data IN   Data to convert and send.

  Serial API (Not supported)

CONFIDENTIAL

4.4.10.7   UART_SendStr

void UART_SendStr(BYTE *str)

Macro: ZW_UART_SEND_STRING(str)

Transmit a null terminated string over the UART. The null data is not transmitted.

  Defined in: ZW_uart_api.h

  Parameters:

  str IN   String pointer.

  Serial API (Not supported)

4.4.10.8   UART_SendNL

void UART_SendNL(void)

Macro: ZW_UART_SEND_NL

Transmit "new line" sequence (CR + LF) over the UART.

  Defined in: ZW_uart_api.h

  Serial API (Not supported)

4.4.10.9   UART_Enable

void UART_Enable(void)

Macro: ZW_UART_ENABLE

Enable the UART and take control of the I/Os that are shared with the ADC.

  Defined in: ZW_uart_api.h

  Serial API (Not supported)

CONFIDENTIAL

4.4.10.10   UART_Disable

void UART_Disable(void)

Macro: ZW_UART_DISABLE

Disable the UART and release the I/Os that are shared with the ADC.

   Defined in: ZW_uart_api.h

  Serial API (Not supported)

4.4.10.11   UART_ClearTx

void UART_ClearTx(void)

Macro: ZW_UART_CLEAR_TX

Clear the UART transmit done flag.

   Defined in: ZW_uart_api.h

  Serial API (Not supported)

4.4.10.12   UART_ClearRx

void UART_ClearRx(void)

Macro: ZW_UART_CLEAR_RX

Clear the UART receiver ready flag.

   Defined in: ZW_uart_api.h

  Serial API (Not supported)

CONFIDENTIAL

4.4.10.13   UART_Write

void UART_Write(BYTE txByte)

Macro: ZW_UART_WRITE(TXBYTE)

Function writes a byte to the UART transmit register. UART_Write makes an immediate write to the UART without checking the SEND_STATUS register. Function returns immediately.
See also: UART_SendByte

   Defined in: ZW_uart_api.h

   Parameters:

   txByte IN   Data to send.

   Serial API (Not supported)

4.4.10.14   UART_Read

BYTE UART_Read(void)

Macro: ZW_UART_READ

Function reads a byte from the UART receive register. UART_Read makes an immediate read and returns without first checking the receive data status. Function returns immediately.
See also: UART_RecByte

   Defined in: ZW_uart_api.h

   Return value:

   BYTE   The contents of the UART receive
                      register.

   Serial API (Not supported)

CONFIDENTIAL

4.4.10.15   Serial debug output.

The serial application interface includes a few macros that can be used for debugging the application software. Defining the "ZW_DEBUG" compile flag enables the following macros. If the "ZW_DEBUG" flag is not defined, the serial interface will not be initialized, and no debug information will be showed on the debug terminal.

4.4.10.15.1   ZW_DEBUG_INIT(baud)

This macro initializes the serial interface. The macro can be placed within the application initialization function (see function **ApplicationInitSW**).

Example:

ZW_DEBUG_INIT(96); /* setup debug output speed to 9600 bps. */

Defined in: ZW_uart_api.h

Serial API (Not supported)

4.4.10.15.2   ZW_DEBUG_SEND_BYTE(data)

This macro sends one byte via the serial interface. The macro can be placed anywhere in the application programs (non interrupt functions).

Example:

ZW_DEBUG_SEND_BYTE('Z'); /* show "Z" on the debug terminal */

Defined in: ZW_uart_api.h

Serial API (Not supported)

4.4.10.15.3   ZW_DEBUG_SEND_NUM(data)

Example:

ZW_DEBUG_SEND_NUM(count); /* show the current value (hexadecimal) of */
                                              /* the local variable "count" on the debug terminal */

Defined in: ZW_uart_api.h

Serial API (Not supported)

CONFIDENTIAL

4.4.11   Z-Wave Node Mask API

The Node Mask API contains a set of functions to manipulate bit masks. This API is not necessary when writing a Z-Wave application, but is provided as an easy way to work with node ID lists as bit masks.

4.4.11.1   ZW_NodeMaskSetBit

void ZW_NodeMaskSetBit(BYTE_P pMask, BYTE bNodeID)

Macro: ZW_NODE_MASK_SET_BIT(pMask, bNodeID)

Set the node bit in a node bit mask.

   Defined in: ZW_nodemask_api.h

   Parameters:

   pMask IN   Pointer to node mask

   bnodeID IN   Node id (1..232) to set in node mask

   Serial API (Not supported)

4.4.11.2   ZW_NodeMaskClearBit

void ZW_NodeMaskClearBit(BYTE_P pMask, BYTE bNodeID)

Macro: ZW_NODE_MASK_CLEAR_BIT(pMask, bNodeID)

Clear the node bit in a node bit mask.

   Defined in: ZW_nodemask_api.h

   Parameters:

   PMask   IN Pointer to node mask

   bNodeID   IN Node ID (1..232) to clear in node
                          mask

   Serial API (Not supported)

CONFIDENTIAL

4.4.11.3   ZW_NodeMaskClear

void ZW_NodeMaskClear(BYTE_P pMask, BYTE bLength)

Macro: ZW_NODE_MASK_CLEAR(pMask, bLength)

Clear all bits in a node mask.

  Defined in: ZW_nodemask_api.h

  Parameters:

  pMask   IN Pointer to node mask

  bLength   IN Length of node mask

  Serial API (Not supported)

4.4.11.4   ZW_NodeMaskBitsIn

BYTE ZW_NodeMaskBitsIn(BYTE_P pMask, BYTE bLength)

Macro: ZW_NODE_MASK_BITS_IN (pMask, bLength)

Number of bits set in node mask.

  Defined in: ZW_nodemask_api.h

  Return value:

  BYTE   Number of bits set in node mask

  Parameters:

  pMask   IN Pointer to node mask

  bLength   IN Length of node mask

  Serial API (Not supported)

CONFIDENTIAL

4.4.11.5   ZW_NodeMaskNodeIn

BYTE ZW_NodeMaskNodeIn (BYTE_P pMask, BYTE bNode)

Macro: ZW_NODE_MASK_NODE_IN (pMask, bNode)

Check if a node is in a node mask.

   Defined in: ZW_nodemask_api.h

   Return value:

   BYTE   ZERO   If not in node mask

                     NONEZERO   If in node mask

   Parameters:

   pMask   IN Pointer to node mask

   bNode   IN Node to clear in node mask

   Serial API (Not supported)

4.5   Z-Wave Controller API

The Z-Wave Controller API makes it possible for different controllers to control the Z-Wave nodes and
get information about each node's capabilities and current state. The node control commands can be
sent to a single node, all nodes or to a list of nodes (group, scene…).

4.5.1 ZW_AddNodeToNetwork

void ZW_AddNodeToNetwork(BYTE bMode,
**   VOID_CALLBACKFUNC(completedFunc)(LEARN_INFO *learnNodeInfo))**

Macro: ZW_ADD_NODE_TO_NETWORK(bMode, func)
Defined in: ZW_controller_api.h

Serial API: Func_ID = 0x4A
HOST->ZW: REQ | 0x4A | mode | funcID
ZW->HOST: REQ | 0x4A | funcID | bStatus | bSource | bLen | basic | generic | specific | cmdclasses[ ]

ZW_AddNodeToNetwork is used to add a node to a Z-Wave network.

The AddNodeToNetwork function MAY be called by a primary controller application to invoke the
inclusion of new nodes in a Z-Wave network. Slave and secondary controller applications MUST NOT
call this function.
A controller application MUST implement support for the AddNodeToNetwork function. The controller
application MUST provide a user interface for activation of the AddNodeToNetwork function.

The bMode and completedFunc parameters MUST be specified for the AddNodeToNetwork function.

CONFIDENTIAL

Refer to Figure 11 for a state diagram outlining the processing of status callbacks and timeouts.

4.5.1.1   bMode parameter

The bMode parameter MUST be composed of commands and flags found in Table 14. The bMode parameter MUST NOT be assigned more than one command. The bMode parameter MAY be assigned one or more option flags. One command and multiple options are combined by logically OR'ing the bMode flags of Table 14.

Table 14. AddNode :: bMode

bMode flag	Description	Usage
ADD_NODE_ANY	Command to initiate inclusion of new node of any type.	MUST be included when initiating inclusion.
ADD_NODE_SLAVE	-	<u>DEPRECATED</u>. Use ADD_NODE_ANY
ADD_NODE_CONTROLLER	-	<u>DEPRECATED</u>. Use ADD_NODE_ANY
ADD_NODE_EXISTING	-	<u>DEPRECATED</u>. Use ADD_NODE_ANY
ADD_NODE_STOP	Command to abort the inclusion process. May only be used in certain states.	MAY be used to abort an active inclusion process. MUST be used to terminate the inclusion process when completed.
ADD_NODE_STOP_FAILED	Command to notify the remote end when a controller replication is aborted.	SHOULD be used if aborting a controller replication.
ADD_NODE_OPTION_ HIGH_POWER	Option flag to enable normal inclusion range.	SHOULD be included with ADD_NODE_ ANY to achieve normal inclusion range. MAY be ommitted for ADD_NODE_ANY to achieve reduced inclusion range.
ADD_NODE_OPTION_ NETWORK_WIDE	Option flag to enable Network-Wide Inclusion (NWI).	MUST be used.

CONFIDENTIAL

4.5.1.1.1   ADD_NODE_ANY command

To invoke inclusion of a new node, a primary controller MUST call the AddNodeToNetwork function with a bMode value including the ADD_NODE_ANY command. Slave and secondary controller nodes MUST NOT call the AddNodeToNetwork function.

The RECOMMENDED call of the AddNodeToNetwork function () when adding nodes is as follows:

```
ZW_ADD_NODE_TO_NETWORK((ADD_NODE_ANY |

                        ADD_NODE_OPTION_HIGH_POWER |

                        ADD_NODE_OPTION_NETWORK_WIDE),

                        completedFunc);
```

While defined in Z-Wave protocol libraries, it is NOT RECOMMENDED to use the ADD_NODE_SLAVE, ADD_NODE_CONTROLLER or ADD_NODE_EXISTING command codes.

4.5.1.1.2   ADD_NODE_STOP command

A controller MAY use the ADD_NODE_STOP command to abort an ongoing inclusion process.

After receiving an ADD_NODE_STATUS_DONE status callback, the application MUST terminate the inclusion process by calling the AddNodeToNetwork function one more time. This time, the completedFunc parameter MUST be the NULL pointer.

Due to the inherent risk of creating ghost nodes with duplicate NodeIDs, a controller SHOULD NOT call the AddNodeToNetwork function in the time window starting with the reception of an ADD_NODE_STATUS_NODE_FOUND status callback and ending with the reception of an ADD_NODE_STATUS_PROTOCOL_DONE status callback. An application may time out waiting for the ADD_NODE_STATUS_PROTOCOL_DONE status callback or the application may receive an ADD_NODE_STATUS_ FAILED status callback. In all three cases, the application MUST terminate the inclusion process by calling AddNodeToNetwork(ADD_NODE_STOP) with a valid completedFunc callback pointer. The API MUST return an ADD_NODE_STATUS_DONE status callback in response.

After receiving an ADD_NODE_STATUS_DONE status callback, the application MUST terminate the inclusion process by calling the AddNodeToNetwork function one more time. This time, the completedFunc parameter MUST be the NULL pointer.

4.5.1.1.3   ADD_NODE_STOP_FAILED command

When a new controller node is included in a Z-Wave network, the primary controller replicates protocol-specific databases to the new controller. An optional application-specific phase may follow after protocol-specific replication.
A primary controller SHOULD use the ADD_NODE_STOP_FAILED command during the application-specific phase to notify the receiving end of the application replication that the process is being aborted.

4.5.1.1.4   ADD_NODE_OPTION_HIGH_POWER option

The default power level for Z-Wave communication is the high power level. Therefore, the high power level is frequently referred to as the normal power level.
When including a new node, the ADD_NODE_OPTION_HIGH_POWER option SHOULD be added to the bMode parameter.

If special application requirements dictate the need for low power transmission during inclusion of a new node, a primary controller MAY omit the ADD_NODE_OPTION_HIGH_POWER option from the bMode parameter. This is however NOT RECOMMENDED.

CONFIDENTIAL

4.5.1.1.5   ADD_NODE_OPTION_NETWORK_WIDE option

Network-Wide Inclusion (NWI) allows a new node to be included across an existing Z-Wave network without direct range connectivity between the primary controller and the new node. The ADD_NODE_OPTION_NETWORK_WIDE option enables NWI. NWI inclusion is backwards compatible with old nodes that do not implement NWI support.

When including a new node, the ADD_NODE_OPTION_NETWORK_WIDE option MUST be added to the bMode parameter.

4.5.1.2   completedFunc parameter

Being the exception to the rule, an application calling AddNodeToNetwork(ADD_NODE_STOP) to confirm the reception of a ADD_NODE_STATUS_DONE return code MUST specify the NULL pointer for the completedFunc parameter.

In all other cases, an application calling the AddNodeToNetwork function with any command and option combination MUST specify a valid pointer to a callback function provided by the application. The callback function MUST accept a pointer parameter to a LEARN_INFO struct. The parameter provides access to actual status as well as companion data presenting a new node. The LEARN_INFO struct only contains a valid pointer to the Node Information Frame of the new node when the status of the callback is ADD_NODE_STATUS_ADDING_SLAVE or ADD_NODE_STATUS_ADDING_CONTROLLER.

Table 15. AddNode :: completedFunc :: learnNodeInfo

LEARN_NODE struct member	Description
*learnNodeInfo.bStatus	Callback status code
*learnNodeInfo.bSource	NodeID of the new node
*learnNodeInfo.bLen	Length of pCmd element following the bLen element. If bLen is zero, there is no valid pCmd element.
*learnNodeInfo.pCmd	Pointer to Application Node Information (see ApplicationNodeInformation - nodeParm). NULL if no information present.

Individual status codes are presented in the following sections.

CONFIDENTIAL

LEARN_NODE.bStatus	Description
ADD_NODE_STATUS_LEARN_READY	Z-Wave protocol is ready to include new node.
ADD_NODE_STATUS_NODE_FOUND	Z-Wave protocol detected node.
ADD_NODE_STATUS_ADDING_SLAVE	Z-Wave protocol included a slave type node
ADD_NODE_STATUS_ADDING_CONTROLLER	Z-Wave protocol included a controller type node
ADD_NODE_STATUS_PROTOCOL_DONE	Z-Wave protocol completed operations related to inclusion. If new node type is controller, the controller application MAY invoke application replication.
ADD_NODE_STATUS_DONE	All operations completed. Protocol is ready to return to idle state.
ADD_NODE_STATUS_FAILED	Z-Wave protocol reports that inclusion was not successful. New node is not ready for operation
ADD_NODE_STATUS_NOT_PRIMARY	Z-Wave protocol reports that the requested operation cannot be performed since it requires that the node is in primary controller state.

Refer to Figure 11 for a state diagram outlining the processing of status callbacks and timeouts.

4.5.1.2.1   ADD_NODE_STATUS_LEARN_READY status

Z-Wave protocol is ready to include new node. An application MAY time out waiting for the ADD_NODE_STATUS_LEARN_READY status if it does not receive the indication within 10 sec after calling AddNodeToNetwork(ADD_NODE_ANY)

If the application times out waiting for the ADD_NODE_STATUS_LEARN_READY status, the application MUST call AddNodeToNetwork(ADD_NODE_STOP, NULL).

4.5.1.2.2   ADD_NODE_STATUS_NODE_FOUND status

Z-Wave protocol detected node. An application MUST time out waiting for the ADD_NODE_STATUS_NODE_FOUND status if it does not receive the indication after calling AddNodeToNetwork(ADD_NODE_ANY). The RECOMMENDED interval is 60 sec.

If the application times out waiting for the ADD_NODE_STATUS_NODE_FOUND status, the application MUST call AddNodeToNetwork(ADD_NODE_STOP, NULL).

The application MUST NOT call AddNodeToNetwork() before the timeout occurs. This may cause the protocol to malfunction.

4.5.1.2.3   ADD_NODE_STATUS_ADDING_SLAVE status

Z-Wave protocol included a slave type node.

An application MUST time out waiting for the ADD_NODE_STATUS_ADDING_SLAVE status if it does not receive the indication within a time period after receiving the ADD_NODE_STATUS_NODE_FOUND status. The time period depends on the network size and the node types in the network. Refer to 4.5.1.3.3.

If the application times out waiting for the ADD_NODE_STATUS_ADDING_SLAVE status, the application MUST call AddNodeToNetwork(ADD_NODE_STOP). The application MUST specify a valid callback function. This allows the application to receive a ADD_NODE_STATUS_DONE once the protocol has completed cleaning up its datastructures.

The application MUST NOT call AddNodeToNetwork() before the timeout occurs. This may cause the protocol to malfunction.

4.5.1.2.4   ADD_NODE_STATUS_ADDING_CONTROLLER status

Z-Wave protocol included a controller type node.

An application MUST time out waiting for the ADD_NODE_STATUS_ADDING_CONTROLLER status if it does not receive the indication within a time period after receiving the ADD_NODE_STATUS_NODE_FOUND status. The time period depends on the network size and the node types in the network. Refer to 4.5.1.3.3.

If the application times out waiting for the ADD_NODE_STATUS_ADDING_CONTROLLER status, the application MUST call AddNodeToNetwork(ADD_NODE_STOP). The application MUST specify a valid callback function. This allows the application to receive an ADD_NODE_STATUS_DONE once the protocol has completed cleaning up its datastructures.

The application MUST NOT call AddNodeToNetwork() before the timeout occurs. This may cause the protocol to malfunction.

4.5.1.2.5   ADD_NODE_STATUS_PROTOCOL_DONE status

Z Wave protocol completed operations related to inclusion. If new node type is controller, the controller application MAY invoke application replication.

In response to the ADD_NODE_STATUS_PROTOCOL_DONE , the application MUST call AddNodeToNetwork(ADD_NODE_STOP). The application MUST specify a valid callback function. This allows the application to receive an ADD_NODE_STATUS_DONE once the protocol has completed cleaning up its datastructures.

An application MUST time out waiting for the ADD_NODE_STATUS_PROTOCOL_DONE status if it does not receive the indication within a time period after receiving the ADD_NODE_STATUS_NODE_FOUND status. The time period depends on the network size and the node types in the network. Refer to 4.5.1.3.3.

If the application times out waiting for the ADD_NODE_STATUS_PROTOCOL_DONE status, the application MUST call AddNodeToNetwork(ADD_NODE_STOP). The application MUST specify a valid callback function. This allows the application to receive a ADD_NODE_STATUS_DONE once the protocol has completed cleaning up its datastructures.

The application MUST NOT call AddNodeToNetwork() before the timeout occurs. This may cause the protocol to malfunction.

4.5.1.2.6   ADD_NODE_STATUS_DONE status

All operations completed. Protocol is ready to return to idle state.

In response to the ADD_NODE_STATUS_DONE status callback, the application MUST call AddNodeToNetwork(ADD_NODE_STOP, NULL). The application MUST specify the NULL pointer for the callback function.

4.5.1.2.7   ADD_NODE_STATUS_FAILED status

An application may time out waiting for the ADD_NODE_STATUS_PROTOCOL_DONE status callback or the application may receive an ADD_NODE_STATUS_PROTOCOL_FAILED status callback. In either case, the application MUST terminate the inclusion process by calling AddNodeToNetwork(ADD_NODE_STOP). Refer to 4.5.1.1.2.

4.5.1.2.8   ADD_NODE_STATUS_NOT_PRIMARY status

An application MUST NOT call the AddNodeToNetwork function if the application is not running in a primary controller. If the function is called by an application running in slave or a secondary controller, the API MUST return the ADD_NODE_STATUS_NOT_PRIMARY status callback.

4.5.1.3   completedFunc callback timeouts

4.5.1.3.1   ProtocolReadyTimeout

The API MUST return an ADD_NODE_STATUS_LEARN_READY status callback within less than 10 sec after receiving a call to AddNodeToNetwork(ADD_NODE_ANY).

If an application has not received an ADD_NODE_STATUS_LEARN_READY status callback 200 msec after calling AddNodeToNetwork(ADD_NODE_ANY), the application MAY time out and return to its idle state.

4.5.1.3.2   NodeTimeout

An application MUST implement a timeout for waiting for an ADD_NODE_STATUS_NODE_FOUND status callback.

The application SHOULD NOT wait for an ADD_NODE_STATUS_NODE_FOUND status callback for more than 60 sec after calling AddNodeToNetwork(ADD_NODE_ANY). If timing out, the application SHOULD abort inclusion.

4.5.1.3.3   AddNodeTimeout

An application MUST implement a timeout for waiting for the protocol library to complete inclusion. The timeout MUST be calculated according to the formulas presented in sections 4.5.1.3.3.1 and 4.5.1.3.3.2.

4.5.1.3.3.1 New slave

AddNodeTimeout.NewSlave =   76000ms +
LISTENINGNODES*217ms +
FLIRSNODES*3517ms

where LISTENINGNODES is the number of listening nodes in the network,
and FLIRSNODES is the number of nodes in the network that are reached via beaming.

4.5.1.3.3.2 New controller

$AddNodeTimeout.NewController = 76000ms +$
$LISTENINGNODES*217ms +$
$FLIRSNODES*3517ms +$
$NETWORKNODES*732ms,$

where LISTENINGNODES is the number of listening nodes in the network,
and FLIRSNODES is the number of nodes in the network that are reached via beaming.

NETWORKNODES is the total number of nodes in the network, i.e.
NONLISTENINGNODES + LISTENINGNODES + FLIRSNODES.

CONFIDENTIAL
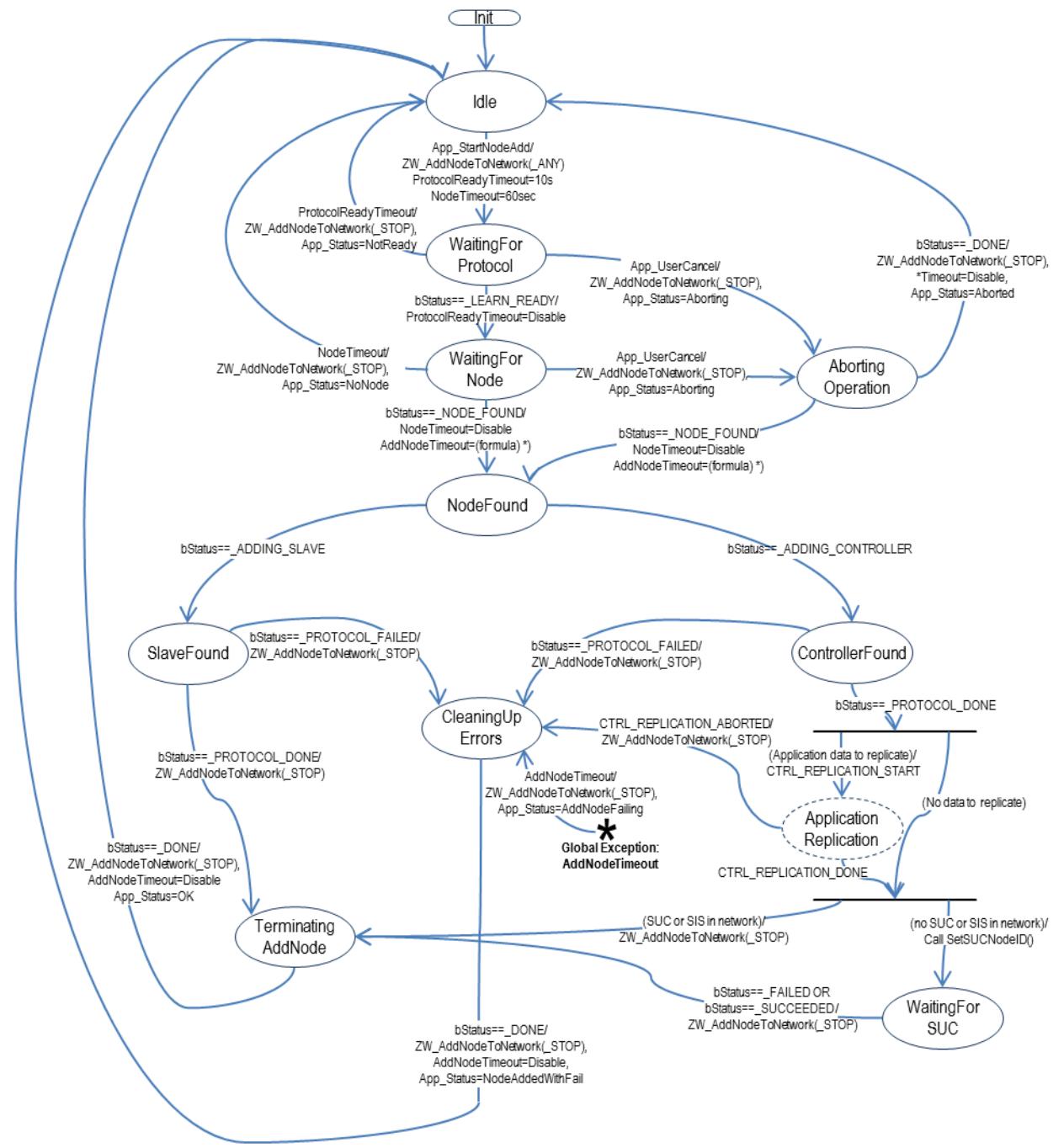
Figure 11. Adding a node to the network

CONFIDENTIAL

Table 17. AddNode : State/Event processing – 1

(Any State)	Event: AddNodeTimeout => // GLOBAL Timer event
	    New state: <CleaningUpErrors>
	    Actions: Call ZW_AddNodeToNetwork(ADD_NODE_STOP)

Idle	Event: (Init) => // Initialize timers, etc.
	
	Event: App_StartNodeAdd => // Higher layer application event calls for node to be added
	    New state: <WaitingForProtocol>
	    Actions: Call ZW_AddNodeToNetwork(ADD_NODE_ANY)
	      ProtocolReadyTimeout=10s
	      AddNodeTimeout=60sec

WaitingForProtocol	Event: App_UserCancel => // Higher layer application event calls for process to be stopped
	    New state: <AbortingOperation>
	    Actions: Call ZW_AddNodeToNetwork(ADD_NODE_STOP)
	      Generate App_Status=Aborting event for application
	
	Event: bStatus==ADD_NODE_STATUS_LEARN_READY => //Callback
	    New state: <WaitingForNode>
	    Actions: Disable ProtocolReadyTimeout timer
	
	Event: ProtocolReadyTimeout => // Timer event
	    New state: <Idle>
	    Actions: Call ZW_AddNodeToNetwork(ADD_NODE_STOP, func=NULL)
	      // Stop operation; do not specify a callbackfunction
	      Generate App_Status=NotReady event for application

WaitingForNode	Event: App_UserCancel => // Higher layer application event calls for process to be stopped
	    New state: < AbortingOperation>
	    Actions: Call ZW_AddNodeToNetwork(ADD_NODE_STOP)
	      Generate App_Status=Aborting event for application
	
	Event: bStatus==ADD_NODE_STATUS_NODE_FOUND => //Callback
	    New state: <NodeFound>
	    Actions: AddNodeTimeout= (calculated as follows:)
	  Slave: AddNodeTimeout = 76000ms + LISTENINGNODES*217ms + FLIRSNODES*3517ms
	  Controller: AddNodeTimeout = 76000ms + LISTENINGNODES*217ms + FLIRSNODES*3517ms +
	        NETWORKNODES*732ms,
	  where NETWORKNODES is the total number of nodes of any type
	
	Event: NodeTimeout => // Timer event
	    New state: <Idle>
	    Actions: Call ZW_AddNodeToNetwork(ADD_NODE_STOP, func=NULL)
	      // Stop operation; do not specify a callbackfunction
	      Generate App_Status=NoNode event for application

CONFIDENTIAL

Table 18. AddNode : State/Event processing – 2

NodeFound	Event: App_UserCancel => // Higher layer application event calls for process to be stopped 　　　　　　New state: < AbortingOperation> 　　　　　　Actions: Call ZW_AddNodeToNetwork(ADD_NODE_STOP) 　　　　　　　　　　Generate App_Status=Aborting event for application Event: bStatus==ADD_NODE_STATUS_ADDING_SLAVE => //Callback 　　　　　　New state: <SlaveFound> Event: bStatus==ADD_NODE_STATUS_ADDING_CONTROLLER => //Callback 　　　　　　New state: <ControllerFound> Event: NodeFoundTimeout => // Timer event 　　　　　　New state: <CleaningUpErrors> 　　　　　　Actions: Call ZW_AddNodeToNetwork(ADD_NODE_STOP)
SlaveFound	Event: bStatus==ADD_NODE_STATUS_PROTOCOL_DONE => //Callback 　　　　　　New state: <TerminatingAddNode> 　　　　　　Actions: Call ZW_AddNodeToNetwork(ADD_NODE_STOP) Event: bStatus==ADD_NODE_STATUS_FAILED => //Callback 　　　　　　New state: <CleaningUpErrors> 　　　　　　Actions: Call ZW_AddNodeToNetwork(ADD_NODE_STOP) Event: AddNodeTimeout => // Timer event 　　　　　　New state: <CleaningUpErrors> 　　　　　　Actions: Call ZW_AddNodeToNetwork(ADD_NODE_STOP)

ControllerFound	Event: bStatus==ADD_NODE_STATUS_PROTOCOL_DONE => //Callback
	　　　　　IF (Application data to replicate) THEN
	　　　　　　New state: <ApplicationReplication>
	　　　　　　Actions: Generate CTRL_REPLICATION_START for
	　　　　　　　<ApplicationReplication> sub-state machine
	　　　　　ELSE // No data to replicate
	　　　　　　IF (No SUC or SIS in the network) THEN
	　　　　　　　New state: <WaitingForSUC>
	　　　　　　　Actions: Call ZW_SetSUCNodeID(NodeID)
	　　　　　　ELSE
	　　　　　　　New state: <TerminatingAddNode>
	　　　　　　　Actions: Call ZW_AddNodeToNetwork(ADD_NODE_STOP)
	　　　　　　ENDIF
	　　　　　ENDIF
	
	Event: bStatus==ADD_NODE_STATUS_FAILED => //Callback
	　　　　　　New state: <CleaningUpErrors>
	　　　　　　Actions: Call ZW_AddNodeToNetwork(ADD_NODE_STOP)

CONFIDENTIAL

Table 19. AddNode : State/Event processing – 3

<ApplicationReplication>	<Application Replication> is a self-contained state diagram. Stay here until finished.
	
	Event: CTRL_REPLICATION_ABORTED
	     New state: <CleaningUpErrors>
	     Actions: Call ZW_AddNodeToNetwork(ADD_NODE_STOP)
	
	Event: CTRL_REPLICATION_DONE
	     IF (No SUC or SIS in the network) THEN
	       New state: <WaitingForSUC>
	       Actions: Call ZW_SetSUCNodeID(NodeID)
	     ELSE
	       New state: <TerminatingAddNode>
	       Actions: Call ZW_AddNodeToNetwork(ADD_NODE_STOP)
	     ENDIF

WaitingForSUC	Event: bStatus==ZW_SUC_SET_SUCCEEDED => //Callback
	     New state: <TerminatingAddNode>
	     Actions: Call ZW_AddNodeToNetwork(ADD_NODE_STOP)
	
	Event: bStatus==ZW_SUC_SET_FAILED => //Callback
	     New state: <CleaningUpErrors>
	     Actions: Call ZW_AddNodeToNetwork(ADD_NODE_STOP)
	
	Event: AddNodeTimeout => // Timer event
	     New state: <CleaningUpErrors>
	     Actions: Call ZW_AddNodeToNetwork(ADD_NODE_STOP)

TerminatingAddNode	Event: bStatus==ADD_NODE_STATUS_DONE => //Callback
	     New state: <Idle>
	     Actions: Call ZW_AddNodeToNetwork(ADD_NODE_STOP)
	     Actions: AddNodeTimeout=Disable
	     Actions: Generate App_Status=OK event for application

CleaningUpErrors	Event: bStatus==ADD_NODE_STATUS_DONE => //Callback
	     New state: <Idle>
	     Actions: Disable all timeouts,
	       Call ZW_AddNodeToNetwork(ADD_NODE_STOP, NULL)
	       Generate App_Status=NodeAddedWithFail event for application

AbortingOperation	Event: bStatus==ADD_NODE_STATUS_DONE => //Callback
	     New state: <Idle>
	     Actions: Disable all timeouts,
	       Call ZW_AddNodeToNetwork(ADD_NODE_STOP, NULL)
	       Generate App_Status=Aborted event for application
	
	Event: bStatus==ADD_NODE_STATUS_NODE_FOUND => //Callback
	     New state: <NodeFound>
	     Actions: AddNodeTimeout= (calculated as follows:)
	     Slave: AddNodeTimeout = 76000ms + LISTENINGNODES*217ms + FLIRSNODES*3517ms
	     Controller: AddNodeTimeout = 76000ms + LISTENINGNODES*217ms + FLIRSNODES*3517ms +
	            NETWORKNODES*732ms,
	     where NETWORKNODES is the total number of nodes of any type

4.5.2   ZW_AreNodesNeighbours

BYTE ZW_AreNodesNeighbours (BYTE bNodeA, BYTE bNodeB)

Macro: ZW_ARE_NODES_NEIGHBOURS (nodeA, nodeB)

Used check if two nodes are marked as being within direct range of each other

Defined in: ZW_controller_api.h

Return value:

BYTE	FALSE	Nodes are not neighbours.
	TRUE	Nodes are neighbours.

Parameters:

bNodeA IN   Node ID A (1...232)

bNodeB IN   Node ID B (1...232)

Serial API

HOST->ZW: REQ | 0xBC | nodeID | nodeID

ZW->HOST: RES | 0xBC | retVal

CONFIDENTIAL

4.5.3   ZW_AssignReturnRoute

BOOL ZW_AssignReturnRoute(BYTE bSrcNodeID,
                              BYTE bDstNodeID,
                              VOID_CALLBACKFUNC(completedFunc)(BYTE txStatus))

Macro: ZW_ASSIGN_RETURN_ROUTE(routingNodeID,destNodeID,func)

Use to assign static return routes (up to 4) to a Routing Slave, Enhanced Slave or Enhanced 232 Slave node. This allows the Routing Slave node to communicate directly with either controllers or other slave nodes. The API call calculates the shortest routes from the Routing Slave node (bSrcNodeID) to the destination node (bDestNodeID) and transmits the return routes to the Routing Slave node (bSrcNodeID). The destination node is part of the return routes assigned to the slave. Up to 5 different destinations can be allocated return routes in a Routing Slave and Enhanced Slave. Attempts to assign new return routes when all 5 destinations already are allocated will be ignored. It is possible to allocate up to 232 different destinations in an Enhanced 232 Slave. Call **ZW_AssignReturnRoute** repeatedly to allocate more than 5 destinations in an Enhanced 232 Slave. Use the API call **ZW_DeleteReturnRoute** to clear assigned return routes.

   Defined in: ZW_controller_api.h

Return value:

BOOL	TRUE	If Assign return route operation started
	FALSE	If an "assign/delete return route" operation already is active.

Parameters:

bSrcNodeID IN	Node ID (1...232) of the routing slave that should get the return routes.
bDstNodeID IN	Destination node ID (1...232)
completedFunc IN	Transmit completed call back function

Callback function Parameters:

txStatus IN	Status of return route assignment	
	(all status codes from ZW_SendData)	See **ZW_SendData**, section 4.4.3.1
	TRANSMIT_COMPLETE_NOROUTE	No routes assigned because a route between source and destination node could not be found.

Timeout: 74s

Exception Recovery: Resume nomal operation.

CONFIDENTIAL

Serial API:

HOST->ZW: REQ | 0x46 | bSrcNodeID | bDstNodeID | funcID

ZW->HOST: RES | 0x46 | retVal

ZW->HOST: REQ | 0x46 | funcID | bStatus

4.5.4   ZW_AssignSUCReturnRoute

BOOL ZW_AssignSUCReturnRoute (BYTE bSrcNodeID,
**   VOID_CALLBACKFUNC (completedFunc)(BYTE bStatus))**

Macro: ZW_ASSIGN_SUC_RETURN_ROUTE(srcnode,func)

Notify presence of a SUC/SIS to a Routing Slave or Enhanced Slave. Furthermore is static return routes (up to 4) assigned to the Routing Slave or Enhanced Slave to enable communication with the SUC/SIS node. The return routes can be used to get updated return routes from the SUC/SIS node by calling ZW_RequestNetWorkUpdated in the Routing Slave or Enhanced Slave. The Routing Slave or Enhanced Slave can call ZW_RediscoveryNeeded in case it detects that none of the return routes are usefull anymore.

Defined in: ZW_controller_api.h

Return value:

| BOOL | TRUE | If the assign SUC return route operation is started. |
| | FALSE | If an "assign/delete return route" operation already is active. |

Parameters:

bSrcNodeID IN   The node ID (1...232) of the routing slave that
                 should get the return route to the SUC node.

completedFunc   Transmit complete call back.
IN

Callback function Parameters:

bStatus IN   (see **ZW_SendData**)

Timeout: 74s

Exception Recovery: Resume nomal operation.

CONFIDENTIAL

Serial API:

HOST->ZW: REQ | 0x51 | bSrcNodeID | funcID | funcID

The extra funcID is added to ensures backward compatible. This parameter has been removed starting from dev. kit 4.1x. and onwards and has therefore no meaning anymore.

ZW->HOST: RES | 0x51 | retVal

ZW->HOST: REQ | 0x51 | funcID | bStatus

4.5.5   ZW_ControllerChange

void ZW_ControllerChange (BYTE mode,
        VOID_CALLBACKFUNC(completedFunc)(LEARN_INFO *learnNodeInfo))

Macro: ZW_CONTROLLER_CHANGE(mode, func)

ZW_ControllerChange is used to add a controller to the Z-Wave network and transfer the role as primary controller to it.

This function has the same functionality as ZW_AddNodeToNetwork(ADD_NODE_ANY,…) except that the new controller will be a primary controller and the controller invoking the function will become secondary.

Defined in: ZW_controller_api.h

Parameters:

mode IN	The learn node states are:	
	CONTROLLER_CHANGE_START	Start the process of adding a controller to the network.
	CONTROLLER_CHANGE_STOP	Stop the controller change
	CONTROLLER_CHANGE_STOP_FAILED	Stop the controller change and report a failure
completedFunc IN	Callback function pointer (Should only be NULL if state is turned off).	

CONFIDENTIAL

Callback function Parameters (completedFunc):

*learnNodeInfo.bStatus IN	Status of learn mode:	
	ADD_NODE_STATUS_LEARN_READY	The controller is now ready to include a node into the network.
	ADD_NODE_STATUS_NODE_FOUND	A node that wants to be included into the network has been found
	ADD_NODE_STATUS_ADDING_CONTROLLER	A new controller has been added to the network
	ADD_NODE_STATUS_PROTOCOL_DONE	The protocol part of adding a controller is complete, the application can now send data to the new controller using **ZW_ReplicationSend()**
	ADD_NODE_STATUS_DONE	The new node has now been included and the controller is ready to continue normal operation again.
	ADD_NODE_STATUS_FAILED	The learn process failed
*learnNodeInfo.bSource IN	Node id of the new node	
*learnNodeInfo.pCmd IN	Pointer to Application Node information data (see **ApplicationNodeInformation** - nodeParm**)**. NULL if no information present.	
	The pCmd only contain information when bLen is not zero, so the information should be stored when that is the case. Regardless of the bStatus.	
*learnNodeInfo.bLen IN	Node info length.	

Timeout: see ZW_AddNodeToNetwork

Exception Recovery: see ZW_AddNodeToNetwork

CONFIDENTIAL

Serial API:

HOST->ZW: REQ | 0x4D | mode | funcID

ZW->HOST: REQ | 0x4D | funcID | bStatus | bSource | bLen | basic | generic | specific | cmdclasses[ ]

4.5.6 ZW_DeleteReturnRoute

BOOL ZW_DeleteReturnRoute(BYTE nodeID,
                           VOID_CALLBACKFUNC(completedFunc)(BYTE txStatus))

Macro: ZW_DELETE_RETURN_ROUTE(nodeID, func)

Delete all static return routes from a Routing Slave, Enhanced Slave or Enhanced 232 Slave node.

Defined in: ZW_controller_api.h

Return value:

BOOL	TRUE	If Delete return route operation started
	FALSE	If an "assign/delete return route" operation already is active.

Parameters:

nodeID IN	Node ID (1...232) of the routing slave node.
completedFunc IN	Transmit completed call back function

Callback function Parameters:

txStatus IN	(see **ZW_SendData**)

Serial API:

HOST->ZW: REQ | 0x47 | nodeID | funcID

ZW->HOST: RES | 0x47 | retVal

ZW->HOST: REQ | 0x47 | funcID | bStatus

Timeout: 40s

Exception Recovery: Resume normal operation

CONFIDENTIAL

4.5.7 ZW_DeleteSUCReturnRoute

BOOL ZW_DeleteSUCReturnRoute (BYTE bNodeID,
VOID_CALLBACKFUNC (completedFunc)(BYTE txStatus))

Macro: ZW_DELETE_SUC_RETURN_ROUTE (nodeID, func)

Delete the return routes of the SUC node from a Routing Slave node or Enhanced Slave node.

Defined in: ZW_controller_api.h

Return value:

BOOL	TRUE	If the delete SUC return route operation is started.
	FALSE	If an "assign/delete return route" operation already is active.

Parameters:

bNodeID IN   Node ID (1..232) of the routing slave node.

completedFunc   Transmit complete call back.
IN

Callback function Parameters:

txStatus IN   (see **ZW_SendData**)

Serial API:

HOST->ZW: REQ | 0x55 | nodeID | funcID

ZW->HOST: RES | 0x55 | retVal

ZW->HOST: REQ | 0x55 | funcID | bStatus

The Serial API implementation do not return the callback function (no parameter in the Serial API frame refers to the callback), this is done via the **ApplicationControllerUpdate** callback function:

- If request nodeinfo transmission was unsuccessful (no ACK received) then the **ApplicationControllerUpdate** is called with UPDATE_STATE_NODE_INFO_REQ_FAILED (status only available in the Serial API implementation).

- If request nodeinfo transmission was successful there is no indication that it went well apart from the returned Nodeinfo frame which should be received via the **ApplicationControllerUpdate** with status UPDATE_STATE_NODE_INFO_RECEIVED.

Timeout: 40s

Exception Recovery: Resume normal operation

4.5.8   ZW_GetControllerCapabilities

BYTE ZW_GetControllerCapabilities (void)

Macro: ZW_GET_CONTROLLER_CAPABILITIES()

ZW_GetControllerCapabilities returns a bitmask containing the capabilities of the controller. It's an old type of primary controller (node ID = 0xEF) in case zero is returned.

NOTE: Not all status bits are available on all controllers types

Defined in: ZW_controller_api.h

Return value:

BYTE	CONTROLLER_IS_SECONDARY	If bit is set then the controller is a secondary controller
	CONTROLLER_ON_OTHER_NETWORK	If this bit is set then this controller is not using its build in home ID
	CONTROLLER_IS_SUC	If this bit is set then this controller is a SUC
	CONTROLLER_NODEID_SERVER_PRESENT	If this bit is set then there is a SUC ID server (SIS) in the network and this controller can therefore include/exclude nodes in the network. This is called an inclusion controller.
	CONTROLLER_IS_REAL_PRIMARY	If this bit is set then this controller was the original primary controller in the network before the SIS was added to the network

Serial API:

HOST->ZW: REQ | 0x05

ZW->HOST: RES | 0x05 | RetVal

CONFIDENTIAL

4.5.9   ZW_GetNeighborCount

BYTE ZW_GetNeighborCount(BYTE nodeID)

Macro: ZW_GET_NEIGHBOR_COUNT (nodeID)

Used to get the number of neighbors the specified node has registered.

Defined in: ZW_controller_api.h

Return value:

BYTE	0x00-0xE7	Number of neighbors registered.
	NEIGHBORS_ID_INVALID	Specified node ID is invalid.
	NEIGHBORS_COUNT_FAILED	Could not access routing information - try again later.

Parameters:

nodeID IN   Node ID (1...232) on the node to count neighbors on.

Serial API

HOST->ZW: REQ | 0xBB | nodeID

ZW->HOST: RES | 0xBB | retVal

CONFIDENTIAL

4.5.10   ZW_GetLastWorkingRoute

BOOL ZW_GetLastWorkingRoute(BYTE nodeID, XBYTE *pLastWorkingRoute)

Macro: ZW_GET_LAST_WORKING_ROUTE(bNodeID, pLastWorkingRoute)

Use this API call to get the Last Working Route (LWR) for a destination node if any exist. The LWR is the last successful route used between sender and destination node. The LWR is stored in NVM.

Defined in: ZW_controller_api.h

Return value:

BOOL	TRUE	A LWR exists for bNodeID and the found route placed in the 4-byte array pointed out by pLastWorkingRoute.
	FALSE	No LWR found for bNodeID.

Parameters:

nodeID IN	The Node ID (1...232) specifies the destination node whom the LWR is wanted from.
pLastWorkingRoute OUT	The parameter returns the pointer to a 4-byte array containing the LWR. If retVal = TRUE and first repeater = 0, then the LWR for bNodeID is a direct route.

Serial API

HOST->ZW: REQ | 0x92 | nodeID

ZW->HOST: RES | 0x92 | nodeID | retVal | repeater0 | repeater1 | repeater2 | repeater3

CONFIDENTIAL

4.5.11   ZW_GetNodeProtocolInfo

void ZW_GetNodeProtocolInfo(BYTE bNodeID,
                                        NODEINFO, *nodeInfo)

Macro: ZW_GET_NODE_STATE(nodeID, nodeInfo)

Return the Node Information Frame without command classes from the NVM for a given node ID:

Byte descriptor \ Bit number	7	6	5	4	3	2	1	0
Capability	Liste-ning	Z-Wave Protocol Specific Part						
Security	Opt. Func.	Sensor 1000ms	Sensor 250ms	Z-Wave Protocol Specific Part				
Reserved	Z-Wave Protocol Specific Part							
Basic	Basic Device Class (Z-Wave Protocol Specific Part)							
Generic	Generic Device Class (Z-Wave Appl. Specific Part)							
Specific	Specific Device Class (Z-Wave Appl. Specific Part)							

Figure 12. Node Information frame structure without command classes

All the Z-Wave protocol specific fields are initialised by the protocol. The Listening flag, Generic and Specific Device Class fields are initialized by the application. Regarding initialisation, refer to the function **ApplicationNodeInformation**.

Defined in: ZW_controller_api.h

Parameters:

bNodeID IN   Node ID   1..232

nodeInfo OUT   Node info buffer (see Figure 12)   If (*nodeInfo).nodeType.generic is 0 then the node doesn't exist.

Serial API:

HOST->ZW: REQ | 0x41 | bNodeID

ZW->HOST: RES | 0x41 | nodeInfo (see Figure 12)

4.5.12   ZW_GetRoutingInfo

void ZW_GetRoutingInfo(BYTE bNodeID,
**   BYTE_P pMask,**
**   BYTE bRemove)**
Macro: ZW_GET_ROUTING_INFO(bNodeID, pMask, bRemove)

ZW_GetRoutingInfo is a function that can be used to read out neighbor information from the protocol.

This information can be used to ensure that all nodes have a sufficient number of neighbors and to ensure that the network is in fact one network.

The format of the data returned in the buffer pointed to by pMask is as follows:

pMask[i] ($0 \le i <$ (ZW_MAX_NODES/8)							
Bit	0	1	2	3	4	5	6	7
NodeID	i*8+1	I*8+2	i*8+3	i*8+4	i*8+5	i*8+6	i*8+7	i*8+8

If a bit n in pMask[i] is 1 it indicates that the node bNodeID has node (i*8)+n+1 as a neighbour. If n in pMask[i] is 0, bNodeID cannot reach node (i*8)+n+1 directly.

Defined in: ZW_controller_api.h

Parameters:

bNodeID IN   Node ID (1…232) specifies the node whom routing info is needed from.

pMask OUT   Pointer to buffer where routing info should be put. The buffer should be at least ZW_MAX_NODES/8 bytes

bRemove IN   GET_ROUTING_INFO_REMOVE_BAD   Remove bad routes from the routing info.

               GET_ROUTING_INFO_REMOVE_NON_REPS   Remove non-repeaters from the routing info.

               GET_ROUTING_INFO_REMOVE_9600   Remove 9.6K nodes from the routing info.

Serial API:

HOST->ZW: REQ | 0x80 | bNodeID | bRemoveBad | bRemoveNonReps | funcID

ZW->HOST: RES | 0x80 | NodeMask[29]

CONFIDENTIAL

4.5.13   ZW_GetSUCNodeID

BYTE ZW_GetSUCNodeID(void)

Macro: ZW_GET_SUC_NODE_ID()

API call used to get the currently registered SUC node ID.

Defined in: ZW_controller_api.h

Return value:

BYTE   The node ID (1..232) on the currently
                  registered SUC, if ZERO then no SUC
                  available.

Serial API:

HOST->ZW: REQ | 0x56

ZW->HOST: RES | 0x56 | SUCNodeID

4.5.14   ZW_isFailedNode

BYTE ZW_isFailedNode(BYTE nodeID)

Macro: ZW_IS_FAILED_NODE_ID(nodeID)

Used to test if a node ID is stored in the failed node ID list.

Defined in: ZW_controller_api.h

Return value:

BYTE   TRUE   If node ID (1..232) is in the list of failing
                                                          nodes.

Parameters:

nodeID IN   The node ID (1...232) to check.

Serial API:

HOST->ZW: REQ | 0x62 | nodeID

ZW->HOST: RES | 0x62 | retVal

CONFIDENTIAL

4.5.15   ZW_IsPrimaryCtrl

BOOL ZW_IsPrimaryCtrl (void)

Macro: ZW_PRIMARYCTRL()

This function is used to request whether the controller is a primary controller or a secondary controller in the network.

Defined in: ZW_controller_api.h

Return value:

BOOL	TRUE	Returns TRUE when the controller is a primary controller in the network.
	FALSE	Return FALSE when the controller is a secondary controller in the network.

Serial API

Not supported but used instead Serial API Node List Command having FUNC_ID_SERIAL_API_GET_INIT_DATA.

CONFIDENTIAL

4.5.16   ZW_RemoveFailedNodeID

BYTE ZW_RemoveFailedNodeID(BYTE NodeID,
**   BOOL bNormalPower,**
**   VOID_CALLBACKFUNC(completedFunc)(BYTE txStatus))**
Macro: ZW_REMOVE_FAILED_NODE_ID(node,func)

Used to remove a non-responding node from the routing table in the requesting controller. A non-responding node is put onto the failed node ID list in the requesting controller. In case the node responds again at a later stage then it is removed from the failed node ID list. A node must be on the failed node ID list and as an extra precaution also fail to respond before it is removed. Responding nodes cannot be removed. The call works on a primary controller and an inclusion controller.

A call back function should be provided otherwise the function would return without removing the node.

Defined in: ZW_controller_api.h

Return value (If the replacing process started successfully then the function will return):

BYTE   ZW_FAILED_NODE_REMOVE_STARTED   The removing process started

Return values (If the replacing process cannot be started then the API function will return one or more of the following flags):

BYTE	ZW_NOT_PRIMARY_CONTROLLER	The removing process was aborted because the controller is not the primary one.
	ZW_NO_CALLBACK_FUNCTION	The removing process was aborted because no call back function is used.
	ZW_FAILED_NODE_NOT_FOUND	The requested process failed. The nodeID was not found in the controller list of failing nodes.
	ZW_FAILED_NODE_REMOVE_PROCESS_BUSY	The removing process is busy.
	ZW_FAILED_NODE_REMOVE_FAIL	The requested process failed. Reasons include: • Controller is busy • The node responded to a NOP; thus the node is no longer failing.

Parameters:

nodeID IN   The node ID (1..232) of the failed node
                 to be deleted.

bNormalPower   If TRUE then using Normal RF Power.
IN

completedFunc   Remove process completed call back
IN   function

CONFIDENTIAL

Callback function Parameters:

txStatus IN   Status of removal of failed node:

        ZW_NODE_OK   The node is working properly (removed from the failed nodes list).

        ZW_FAILED_NODE_REMOVED   The failed node was removed from the failed nodes list.

        ZW_FAILED_NODE_NOT_REMOVED   The failed node was not removed because the removing process cannot be completed.

Serial API:

HOST->ZW: REQ | 0x61 | nodeID | funcID

ZW->HOST: RES | 0x61 | retVal

ZW->HOST: REQ | 0x61 | funcID | txStatus

Timeout: 65s

Exception Recovery: Resume normal operation

4.5.17   ZW_ReplaceFailedNode

BYTE ZW_ReplaceFailedNode(BYTE NodeID,
                              BOOL bNormalPower,
                              VOID_CALLBACKFUNC(completedFunc)(BYTE txStatus))

Macro: ZW_REPLACE_FAILED_NODE(node,func)

This function replaces a non-responding node with a new one in the requesting controller. A non-responding node is put onto the failed node ID list in the requesting controller. In case the node responds again at a later stage then it is removed from the failed node ID list. A node must be on the failed node ID list and as an extra precaution also fail to respond before it is removed. Responding nodes can't be replace. The call works on a primary controller and an inclusion controller.

A call back function should be provided otherwise the function will return without replacing the node.

Defined in: ZW_controller_api.h

Return value (If the replacing process started successfully then the function will return):

BYTE   ZW_FAILED_NODE_REMOVE_STARTED   The replacing process has started.

Return values (If the replacing process cannot be started then the API function will return one or more of the following flags:):

BYTE	ZW_NOT_PRIMARY_CONTROLLER	The replacing process was aborted because the controller is not a primary/inclusion/SIS controller.
	ZW_NO_CALLBACK_FUNCTION	The replacing process was aborted because no call back function is used.
	ZW_FAILED_NODE_NOT_FOUND	The requested process failed. The nodeID was not found in the controller list of failing nodes.
	ZW_FAILED_NODE_REMOVE_PROCESS_BUSY	The removing process is busy.
	ZW_FAILED_NODE_REMOVE_FAIL	The requested process failed. Reasons include: • Controller is busy • The node responded to a NOP; thus the node is no longer failing.

CONFIDENTIAL

Parameters:

nodeID IN   The node ID (1…232) of the failed node
                   to be deleted.

bNormalPower   If TRUE then using Normal RF Power.
IN

completedFunc   Replace process completed call back
IN   function

Callback function Parameters:

txStatus IN   Status of replace of failed node:

ZW_NODE_OK	The node is working properly (removed from the failed nodes list). Replace process is stopped.
ZW_FAILED_NODE_REPLACE	The failed node is ready to be replaced and controller is ready to add new node with the nodeID of the failed node. Meaning that the new node must now emit a nodeinformation frame to be included.
ZW_FAILED_NODE_REPLACE_DONE	The failed node has been replaced.
ZW_FAILED_NODE_REPLACE_FAILED	The failed node has not been replaced.

Serial API:

HOST->ZW: REQ | 0x63 | nodeID | funcID

ZW->HOST: RES | 0x63 | retVal

ZW->HOST: REQ | 0x63 | funcID | txStatus

Timeout: The same ass in AddNodeToNetwork, but for this function there is no "NODE_FOUND" callback. This means that the time from ZW_FAILED_NODE_REPLACE to ZW_FAILED_NODE_REPLACE_DONE includes the time it takes for the user to push the Z-Wave button on the new device.

Exception Recovery: call ZW_AddNodeToNetwok(ADD_NODE_STOP)

CONFIDENTIAL
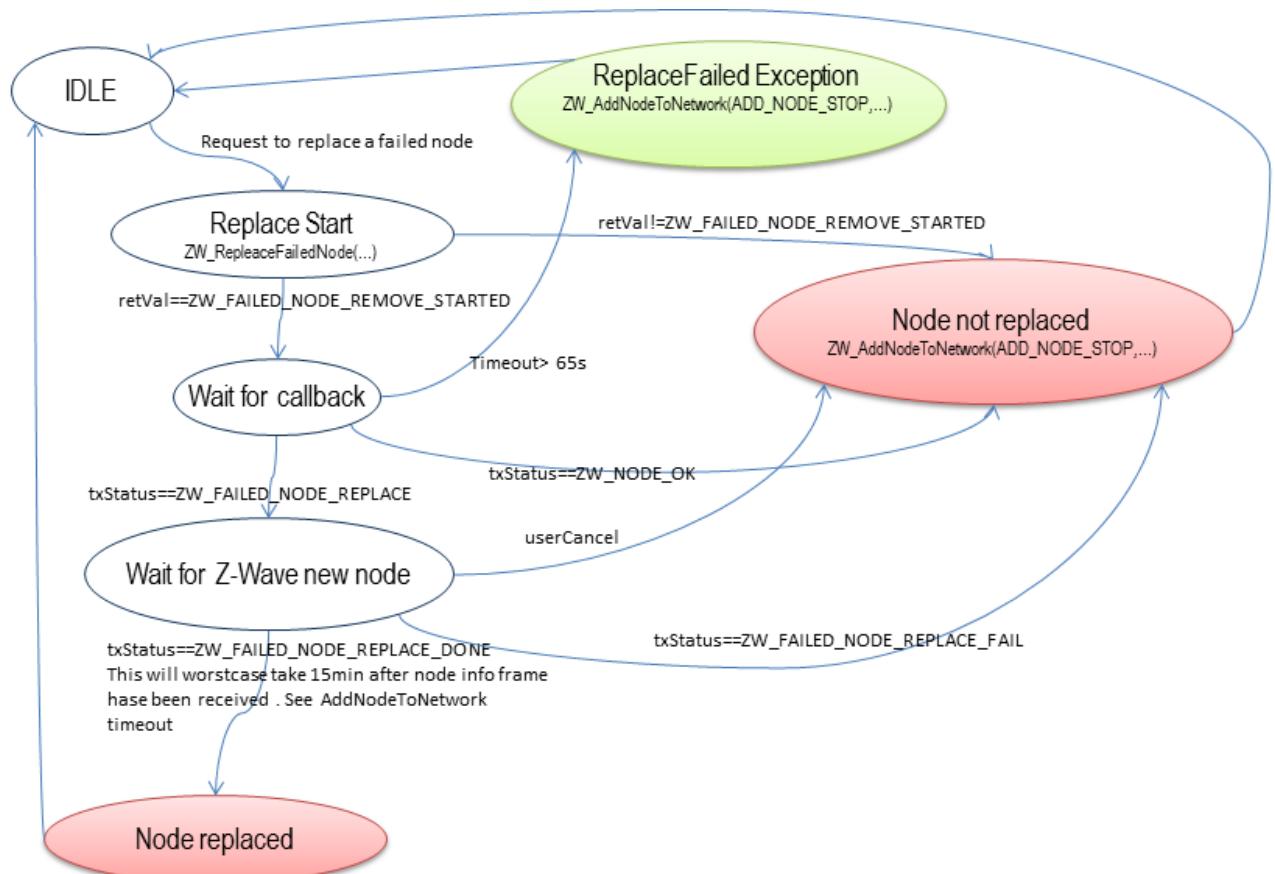
Figure 13. Replacing a failed node

4.5.18   ZW_RemoveNodeFromNetwork

void ZW_RemoveNodeFromNetwork(BYTE mode,
        VOID_CALLBACKFUNC(completedFunc)(LEARN_INFO *learnNodeInfo))

Macro: ZW_REMOVE_NODE_FROM_NETWORK(mode, func)
Defined in: ZW_controller_api.h

Serial API: Func_ID = 0x4B
HOST->ZW: REQ | 0x4B | mode | funcID
ZW->HOST: REQ | 0x4B | funcID | bStatus | bSource | bLen | basic | generic | specific | cmdclasses[ ]

ZW_RemoveNodeFromNetwork is used to remove a node from a Z-Wave network.

The RemoveNodeFromNetwork function MAY be called by a primary controller application to invoke the
removal of nodes from a Z-Wave network. Slave and secondary controller applications MUST NOT call
this function.
A controller application MUST implement support for the RemoveNodeFromNetwork function. The
controller application MUST provide a user interface for activation of the RemoveNodeFromNetwork
function.

CONFIDENTIAL

The bMode and completedFunc parameters MUST be specified for the RemoveNodeFromNetwork function.

Refer to Figure 14 for a state diagram outlining the processing of status callbacks and timeouts.

4.5.18.1   bMode parameter

The bMode parameter MUST be carry one of the commands found in Table 20. The bMode parameter MUST NOT be assigned more than one command. The bMode parameter MAY be assigned one or more option flags. One command and multiple options are combined by logically OR'ing the bMode flags of Table 14.

Table 20. RemoveNode :: bMode

bMode flag	Description	Usage
REMOVE_NODE_ANY	Command to initiate removal of node of any type.	MUST be included when initiating removal.
REMOVE_NODE_SLAVE	-	<u>DEPRECATED</u>. Use REMOVE_NODE_ANY
REMOVE_NODE_CONTROLLER	-	<u>DEPRECATED</u>. Use REMOVE_NODE_ANY
REMOVE_NODE_STOP	Command to abort the removal process. May only be used in certain states.	MAY be used to abort an active removal process. MUST be used to terminate the removal process when completed.

4.5.18.1.1   REMOVE_NODE_ANY command

To invoke removal of a node, a primary controller MUST call the RemoveNodeFromNetwork function with a bMode value including the REMOVE_NODE_ANY command. Slave and secondary controller nodes MUST NOT call the RemoveNodeFromNetwork function.

While defined in Z-Wave protocol libraries, it is NOT RECOMMENDED to use the REMOVE_NODE_SLAVE or REMOVE_NODE_CONTROLLER command codes.

4.5.18.1.2   REMOVE_NODE_STOP command

A controller MAY use the REMOVE_NODE_STOP command to abort an ongoing removal process.

After receiving a REMOVE_NODE_STATUS_DONE status callback, the application MUST terminate the removal process by calling the RemoveNodeFromNetwork function one more time. This time, the completedFunc parameter MUST be the NULL pointer.

4.5.18.2   completedFunc parameter

Being the exception to the rule, an application calling RemoveNodeFromNetwork(REMOVE_NODE_STOP) to confirm the reception of a REMOVE_NODE_STATUS_DONE return code MUST specify the NULL pointer for the completedFunc parameter.

CONFIDENTIAL

In all other cases, an application calling the RemoveNodeFromNetwork function MUST specify a valid pointer to a callback function provided by the application. The callback function MUST accept a pointer parameter to a LEARN_INFO struct. The parameter provides access to actual status as well as companion data presenting the node being removed. The LEARN_INFO struct only contains a valid pointer to the Node Information Frame of a node when the status of the callback is REMOVE_NODE_STATUS_REMOVING_SLAVE or REMOVE_NODE_STATUS_REMOVING_CONTROLLER.

Table 21. RemoveNode :: completedFunc :: learnNodeInfo

LEARN_NODE struct member	Description
*learnNodeInfo.bStatus	Callback status code
*learnNodeInfo.bSource	NodeID of the node that was removed
*learnNodeInfo.bLen	Length of pCmd element following the bLen element. If bLen is zero, there is no valid pCmd element.
*learnNodeInfo.pCmd	Pointer to Application Node Information (see ApplicationNodeInformation - nodeParm). NULL if no information present.

Individual status codes are presented in the following sections.

Table 22. RemoveNode :: completedFunc :: learnNodeInfo.bStatus

LEARN_NODE.bStatus	Description
REMOVE_NODE_STATUS_LEARN_READY	Z-Wave protocol is ready to remove a node.
REMOVE_NODE_STATUS_NODE_FOUND	Z-Wave protocol detected node.
REMOVE_NODE_STATUS_REMOVING_ SLAVE	Z-Wave protocol removed a slave type node
REMOVE_NODE_STATUS_REMOVING_ CONTROLLER	Z-Wave protocol removed a controller type node
REMOVE_NODE_STATUS_DONE	All operations completed. Protocol is ready to return to idle state.
REMOVE_NODE_STATUS_FAILED	Z-Wave protocol reports that removal was not successful. Node may not have been removed.
ADD_NODE_STATUS_NOT_PRIMARY	Z Wave protocol reports that the requested operation cannot be performed since it requires that the node is in primary controller state.

Refer to Figure 14 for a state diagram outlining the processing of status callbacks and timeouts.

CONFIDENTIAL

4.5.18.2.1   REMOVE_NODE_STATUS_LEARN_READY status

Z-Wave protocol is ready to remove a node. An application MAY time out waiting for the REMOVE_NODE_STATUS_LEARN_READY status if it does not receive the indication within 200 msec after calling RemoveNodeFromNetwork(REMOVE_NODE_ANY).

If the application times out waiting for the REMOVE_NODE_STATUS_LEARN_READY status, the application MUST call RemoveNodeFromNetwork(REMOVE_NODE_STOP, NULL).

4.5.18.2.2   REMOVE_NODE_STATUS_NODE_FOUND status

Z-Wave protocol detected node. An application MUST time out waiting for the REMOVE_NODE_STATUS_NODE_FOUND status if it does not receive the indication after calling RemoveNodeFromNetwork(REMOVE_NODE_ANY). The RECOMMENDED interval is 60 sec.

If the application times out waiting for the REMOVE_NODE_STATUS_NODE_FOUND status, the application MUST call RemoveNodeFromNetwork(REMOVE_NODE_STOP, NULL).

The application MUST NOT call RemoveNodeFromNetwork() before the timeout occurs. This may cause the protocol to malfunction.

4.5.18.2.3   REMOVE_NODE_STATUS_REMOVING_SLAVE status

Z-Wave protocol is removing a slave type node. The NodeID of the node is included in the callback.

An application MUST time out waiting for the REMOVE_NODE_STATUS_REMOVING_SLAVE status if it does not receive the indication within a 14 sec after receiving the REMOVE_NODE_STATUS_NODE_FOUND status.

If the application times out waiting for the REMOVE_NODE_STATUS_REMOVNG_SLAVE status, the application MUST call RemoveNodeFromNetwork(REMOVE_NODE_STOP). The application MUST specify a valid callback function. This allows the application to receive a REMOVE_NODE_STATUS_DONE once the protocol has completed cleaning up its datastructures.

The application MUST NOT call RemoveNodeFromNetwork() before the timeout occurs. This may cause the protocol to malfunction.

4.5.18.2.4   REMOVE_NODE_STATUS_REMOVING_CONTROLLER status

Z-Wave protocol is removing a controller type node. The NodeID of the node is included in the callback.

An application MUST time out waiting for the REMOVE_NODE_STATUS_REMOVING_CONTROLLER status if it does not receive the indication within a 14 sec after receiving the REMOVE_NODE_STATUS_NODE_FOUND status.

If the application times out waiting for the REMOVE_NODE_STATUS_REMOVING_CONTROLLER status, the application MUST call RemoveNodeFromNetwork(REMOVE_NODE_STOP). The application MUST specify a valid callback function. This allows the application to receive an REMOVE_NODE_STATUS_DONE once the protocol has completed cleaning up its datastructures.

The application MUST NOT call RemoveNodeFromNetwork() before the timeout occurs. This may cause the protocol to malfunction.

CONFIDENTIAL

4.5.18.2.5   REMOVE_NODE_STATUS_DONE status

All operations completed. Protocol is ready to return to idle state.

In response to the REMOVE_NODE_STATUS_DONE status callback, the application MUST call
RemoveNodeFromNetwork(REMOVE_NODE_STOP, NULL). The application MUST specify the NULL
pointer for the callback function.

4.5.18.2.6   REMOVE_NODE_STATUS_FAILED status

If an application receives a REMOVE_NODE_STATUS_PROTOCOL_FAILED status callback, the
application MUST terminate the removal process by calling
RemoveNodeFromNetwork(REMOVE_NODE_STOP). Refer to 4.5.18.1.2.

4.5.18.2.7   ADD_NODE_STATUS_NOT_PRIMARY status

An application MUST NOT call the RemoveNodeFromNetwork function if the application is not running in
a primary controller. If the function is called by an application running in slave or a secondary controller,
the API MUST return the ADD_NODE_STATUS_NOT_PRIMARY status callback.

4.5.18.3   completedFunc callback timeouts

4.5.18.3.1   ProtocolReadyTimeout

The API MUST return a REMOVE_NODE_STATUS_LEARN_READY status callback within less than
200 msec after receiving a call to RemoveNodeFromNetwork(REMOVE_NODE_ANY).

If an application has not received a REMOVE_NODE_STATUS_LEARN_READY status callback 200
msec after calling RemoveNodeFromNetwork(REMOVE_NODE_ANY), the application MAY time out and
return to its idle state.

4.5.18.3.2   NodeTimeout

An application MUST implement a timeout for waiting for an REMOVE_NODE_STATUS_NODE_FOUND
status callback.

The application SHOULD NOT wait for a REMOVE_NODE_STATUS_NODE_FOUND status callback for
more than 60 sec after calling RemoveNodeFromNetwork(REMOVE_NODE_ANY). If timing out, the
application SHOULD abort removal.

4.5.18.3.3   RemoveNodeTimeout

An application MUST time out if removal has not been completed within 14 sec after the reception of the
REMOVE_NODE_STATUS_NODE_FOUND status callback.

If timing out, the application MUST evaluate the controller node list to verify that the NodeID was
removed. The removal process SHOULD be repeated if the NodeID is still found in the node list.

CONFIDENTIAL
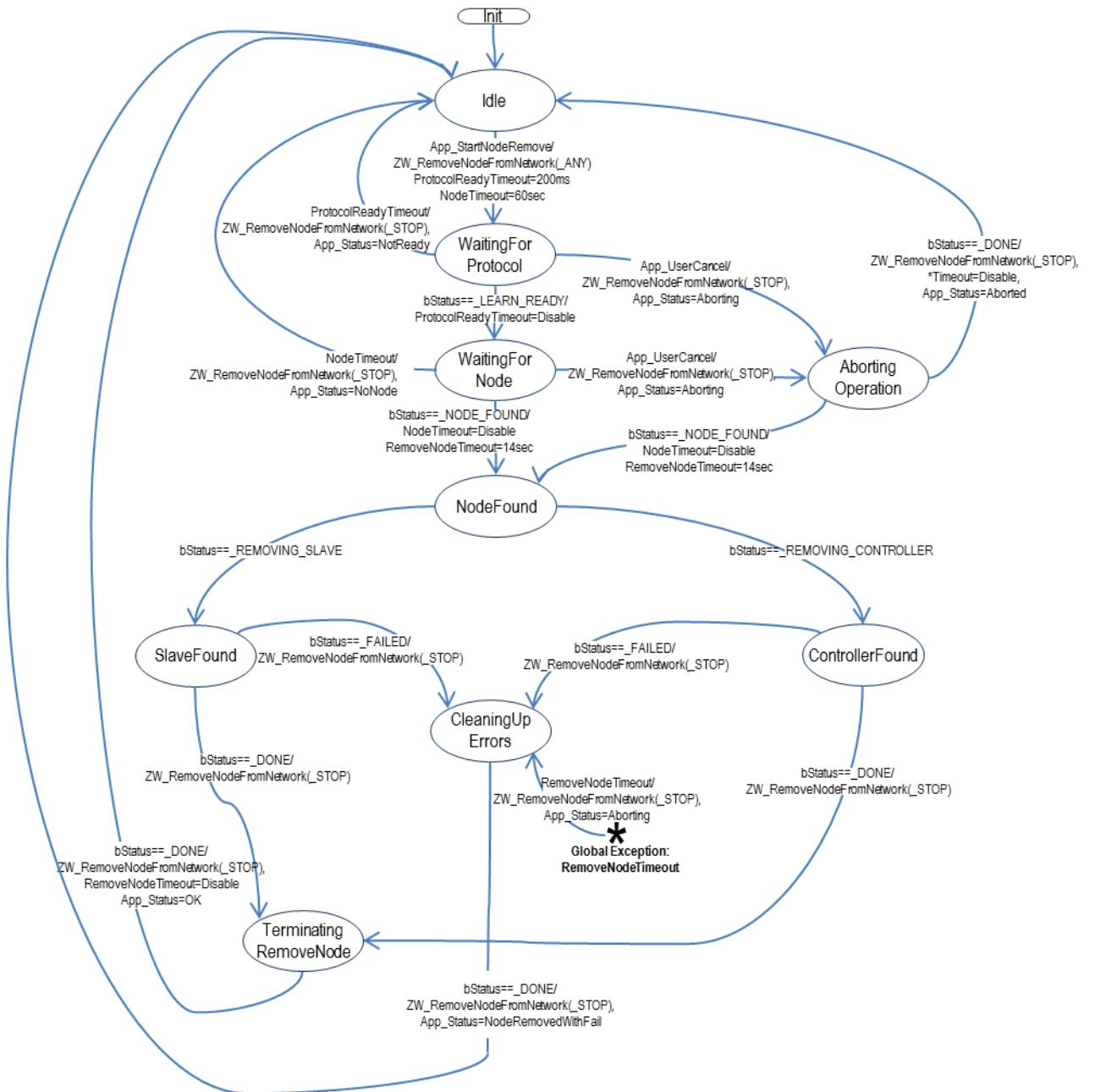
Figure 14. Removing a node from the network

Table 23. RemoveNode : State/Event processing - 1

(Any State)	Event: RemoveNodeTimeout => // GLOBAL Timer event
	        New state: <CleaningUpErrors>
	        Actions: Call ZW_RemoveNodeFromNetwork(REMOVE_NODE_STOP)

Idle	Event: (Init) => // Initialize timers, etc.
	
	Event: App_StartNodeRemove => // Higher layer application event calls for node to be removed
	        New state: <WaitingForProtocol>
	        Actions: Call ZW_RemoveNodeFromNetwork(REMOVE_NODE_ANY)
	            ProtocolReadyTimeout=200ms
	            NodeTimeout=60sec

WaitingForProtocol	Event: App_UserCancel => // Higher layer application event calls for process to be stopped
	        New state: <AbortingOperation>
	        Actions: Call ZW_AddNodeToNetwork(REMOVE_NODE_STOP)
	            Generate App_Status=Aborting event for application
	
	Event: bStatus==REMOVE_NODE_STATUS_LEARN_READY => //Callback
	        New state: <WaitingForNode>
	        Actions: Disable ProtocolReadyTimeout timer
	
	Event: ProtocolReadyTimeout => // Timer event
	        New state: <Idle>
	        Actions: Call ZW_RemoveNodeFromNetwork(REMOVE_NODE_STOP, NULL)
	            // Stop operation; do not specify a callbackfunction
	            Generate App_Status=NotReady event for application

WaitingForNode	Event: App_UserCancel => // Higher layer application event calls for process to be stopped
	        New state: <AbortingOperation>
	        Actions: Call ZW_RemoveNodeFromNetwork(REMOVE_NODE_STOP)
	            Generate App_Status=Aborting event for application
	
	Event: bStatus==REMOVE_NODE_STATUS_NODE_FOUND => //Callback
	        New state: <NodeFound>
	        Actions: NodeTimeout=Disable
	        Actions: RemoveNodeTimeout=14 sec
	
	Event: NewNodeTimeout => // Timer event
	        New state: <Idle>
	        Actions: Call ZW_RemoveNodeFromNetwork(REMOVE_NODE_STOP, NULL)
	            // Stop operation; do not specify a callbackfunction
	            Generate App_Status=NoNode event for application

NodeFound	Event: bStatus==REMOVE_NODE_STATUS_REMOVING_SLAVE => //Callback
	        New state: <SlaveFound>
	
	Event: bStatus==REMOVE_NODE_STATUS_REMOVING_CONTROLLER => //Callback
	        New state: <ControllerFound>
	
	Event: NodeFoundTimeout => // Timer event
	        New state: <CleaningUpErrors>
	        Actions: Call ZW_RemoveNodeFromNetwork(REMOVE_NODE_STOP)

CONFIDENTIAL

Table 24. RemoveNode : State/Event processing - 2

SlaveFound	Event: bStatus==REMOVE_NODE_STATUS_DONE => //Callback 　　　New state: <TerminatingRemoveNode> 　　　Actions: Call ZW_RemoveNodeFromNetwork(REMOVE_NODE_STOP) Event: bStatus==REMOVE_NODE_STATUS_FAILED => //Callback 　　　New state: <CleaningUpErrors> 　　　Actions: Call ZW_RemoveNodeFromNetwork(REMOVE_NODE_STOP) Event: RemoveNodeTimeout => // Timer event 　　　New state: <CleaningUpErrors> 　　　Actions: Call ZW_RemoveNodeFromNetwork(REMOVE_NODE_STOP)
ControllerFound	Event: bStatus==REMOVE_NODE_STATUS_DONE => //Callback 　　　New state: <TerminatingRemoveNode> 　　　Actions: Call ZW_RemoveNodeFromNetwork(REMOVE_NODE_STOP) Event: bStatus==REMOVE_NODE_STATUS_FAILED => //Callback 　　　New state: <CleaningUpErrors> 　　　Actions: Call ZW_RemoveNodeFromNetwork(REMOVE_NODE_STOP) Event: RemoveNodeTimeout => // Timer event 　　　New state: <CleaningUpErrors> 　　　Actions: Call ZW_RemoveNodeFromNetwork(REMOVE_NODE_STOP)
TerminatingRemoveNode	Event: bStatus==REMOVE_NODE_STATUS_DONE => //Callback 　　　New state: <Idle> 　　　Actions: Call ZW_RemoveNodeFromNetwork(REMOVE_NODE_STOP) 　　　Actions: RemoveNodeTimeout=Disable 　　　Actions: Generate App_Status=OK event for application
CleaningUpErrors	Event: bStatus==REMOVE_NODE_STATUS_DONE => //Callback 　　　New state: <Idle> 　　　Actions: Disable all timeouts, 　　　　　Call ZW_RemoveNodeFromNetwork(REMOVE_NODE_STOP, NULL) 　　　　　Generate App_Status=NodeRemoveWithFail event for application
AbortingOperation	Event: bStatus==REMOVE_NODE_STATUS_DONE => //Callback 　　　New state: <Idle> 　　　Actions: Disable all timeouts, 　　　　　Call ZW_RemoveNodeFromNetwork(REMOVE_NODE_STOP, NULL) 　　　　　Generate App_Status=Aborted event for application Event: bStatus==REMOVE_NODE_STATUS_NODE_FOUND => //Callback 　　　New state: <NodeFound> 　　　Actions: NodeTimeout=Disable 　　　Actions: RemoveNodeTimeout=14 sec

CONFIDENTIAL

4.5.19   ZW_ReplicationReceiveComplete

void ZW_ReplicationReceiveComplete(void)

Macro: ZW_REPLICATION_COMMAND_COMPLETE

Sends command completed to sending controller. Called in conjunction with Controller Replication Command Class when payload in **ZW_ReplicationSend** from the sender has been processed indicating that the controller is ready for next packet.

WARNING: Use API call only in conjunction with Controller Replication Command Class [33].

  Defined in: ZW_controller_api.h

  Serial API:

  HOST->ZW: REQ | 0x44

4.5.20   ZW_ReplicationSend

**BYTE ZW_ReplicationSend(BYTE destNodeID, BYTE *pData, BYTE dataLength,
                        BYTE txOptions,
                        VOID_CALLBACKFUNC(completedFunc)(BYTE txStatus))**

Macro: ZW_REPLICATION_SEND_DATA(node,data,length,options,func)

Used when the controller wants to make a controller replication to another controller using Controller Replication Command Class. Controller sends the payload in **ZW_ReplicationSend** and expects the receiver to respond with a command processed message **ZW_ReplicationReceiveComplete**.

WARNING: Use API call only in conjunction with Controller Replication Command Class [33].

  Defined in: ZW_controller_api.h

  Return value:

  BYTE   FALSE   If transmit queue overflow.

  Parameters:

  destNode IN   Destination Node ID
                    (not equal NODE_BROADCAST).

  pData IN   Data buffer pointer

  dataLength IN   Data buffer length

  txOptions IN   Transmit option flags. (see
                    ZW_SendData, but avoid using
                    routing!)

  completedFunc   Transmit completed call back function

CONFIDENTIAL

IN

Callback function Parameters:

txStatus IN   (see **ZW_SendData**)

Serial API:

HOST->ZW: REQ | 0x45 | destNodeID | dataLength | pData[ ] | txOptions | funcID

ZW->HOST: RES | 0x45 | RetVal

ZW->HOST: REQ | 0x45 | funcID | txStatus

Timeout: 4s

Exception recovery: Abort the ZW_AddNodeToNetwork Retry the inclustio of the controller

CONFIDENTIAL

4.5.21   ZW_RequestNodeInfo

BOOL ZW_RequestNodeInfo (BYTE nodeID,
                                VOID (*completedFunc)(BYTE txStatus))

Macro: ZW_REQUEST_NODE_INFO(NODEID)

This function is used to request the Node Information Frame from a controller based node in the network. The Node info is retrieved using the **ApplicationControllerUpdate** callback function with the status UPDATE_STATE_NODE_INFO_RECEIVED. This call is also available for routing slaves.

Defined in: ZW_controller_api.h

Return value:

BOOL	TRUE	If the request could be put in the transmit queue successfully.
	FALSE	If the request could not be put in the transmit queue. Request failed.

Parameters:

nodeID IN   The node ID (1...232) of the node to request the Node Information Frame from.

completedFunc   Transmit complete call back.
IN

Callback function Parameters:

txStatus IN   (see **ZW_SendData**)

Serial API:

HOST->ZW: REQ | 0x60 | NodeID

ZW->HOST: RES | 0x60 | retVal

SerialAPI Note:

The serial API implements no callback for this function. As a special serialAPI feature a transmit failure will result in a call to the ApplicationControllerUpdate with the status code UPDATE_STATE_NODE_INFO_REQ_FAILED. No call back I given on success, however the application should receive a ApplicationControllerUpdate(UPDATE_STATE_NODE_INFO_RECEIVED,…) when the node info arrives.

Timeout: 65s

Exception Recovery: Resume normal operation, retry the command.

CONFIDENTIAL

4.5.22   ZW_RequestNodeNeighborUpdate

BYTE ZW_RequestNodeNeighborUpdate(NODEID,
                                     VOID_CALLBACKFUNC (completedFunc)(BYTE
                                     bStatus))

Macro: ZW_REQUEST_NODE_NEIGHBOR_UPDATE(nodeid, func)

Get the neighbors from the specified node. This call can only be called by a primary/inclusion controller. An inclusion controller should call **ZW_RequestNetWorkUpdate** in advance because the inclusion controller may not have the latest network topology.

Defined in: ZW_controller_api.h

Return value:

BYTE	TRUE	The discovery process is started and the function will be completed by the callback
	FALSE	The discovery was not started and the callback will not be called. The reason for the failure can be one of the following: • This is not a primary/inclusion controller • There is only one node in the network, nothing to update. • The controller is busy doing another update.

Parameters:

nodeID IN   Node ID (1...232) of the node that the
                 controller wants to get new neighbors from.

completedFunc   Transmit complete call back.
IN

Callback function Parameters:

bStatus IN   Status of command:

REQUEST_NEIGHBOR_UPDATE_STARTED	Requesting neighbor list from the node is in progress.
REQUEST_NEIGHBOR_UPDATE_DONE	New neighbor list received
REQUEST_NEIGHBOR_UPDATE_FAIL	Getting new neighbor list failed

Serial API:

HOST->ZW: REQ | 0x48 | nodeID | funcID

CONFIDENTIAL

ZW->HOST: REQ | 0x48 | funcID | bStatus

Timeout: 3*1100ms*(num_of_nodes_in_network), worst case 12 minutes. This process is not cancable.

Exception Recovery: Resume normal operation
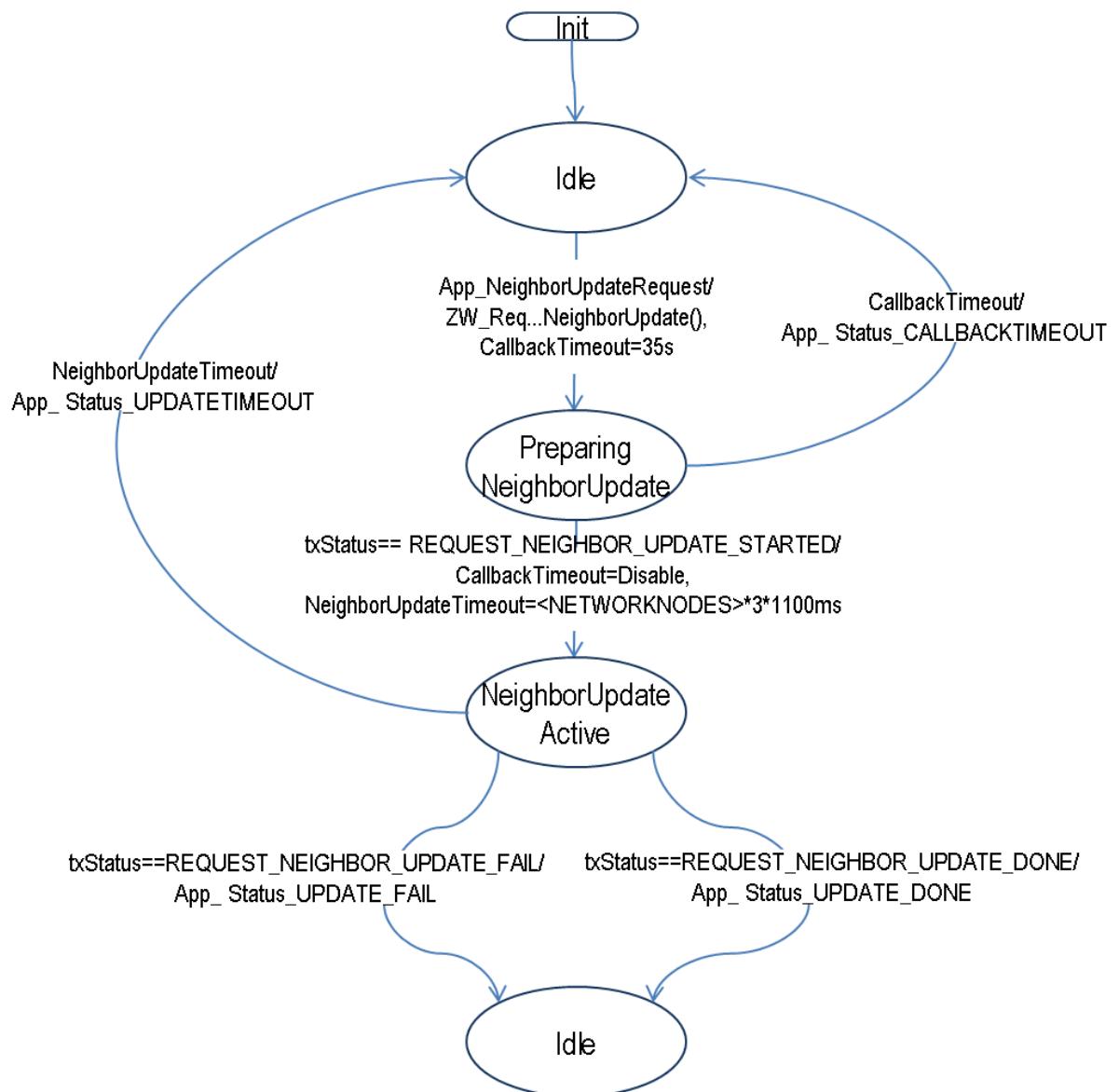
Figure 15. Application state machine for ZW_RequestNodeNeighborUpdate

4.5.23   ZW_SendSUCID

BYTE ZW_SendSUCID (BYTE node,
                    BYTE txOption,
                    VOID_CALLBACKFUNC (completedFunc)(BYTE txStatus))

Macro: ZW_SEND_SUC_ID(nodeID, txOption, func)

Transmit SUC node ID from a primary controller or static controller to the controller node ID specified.
Routing slaves ignore this command, use instead ZW_AssignSUCReturnRoute.

Defined in: ZW_controller_api.h

Return value:

TRUE		In progress.
FALSE		Not a primary controller or static controller.

Parameters:

node IN   The node ID (1...232) of the node to
                     receive the current SUC node ID.

txOption IN   Transmit option flags. (see
                     ZW_SendData)

completedFunc   Transmit complete call back.
IN

Callback function parameters:

txStatus IN   (see **ZW_SendData**)

Serial API:

HOST->ZW: REQ | 0x57 | node | txOption | funcID

ZW->HOST: RES | 0x57 | RetVal

ZW->HOST: REQ | 0x57 | funcID | txStatus

Timeout: 65s

Exception Recovery: Resume normal operation

CONFIDENTIAL

4.5.24 ZW_SetDefault

void ZW_SetDefault(VOID_CALLBACKFUNC(completedFunc)(void))

Macro: ZW_SET_DEFAULT(func)

This function set the Controller back to the factory default state. Erase all Nodes, routing information and assigned homeID/nodeID from the NVM. In case the previous home ID was randomly generated then a new random home ID written to the NVM (random range: 0xC0000000-0xFFFFFFFE). A home ID outside random range reuses the initially configured home ID (configured during production).

Warning: Use this function with care as it could render a Z-Wave network unusable if the primary controller in an existing network is set back to default.

Defined in: ZW_controller_api.h

Parameters:

completedFunc IN   Command completed call back function

Serial API:

HOST->ZW: REQ | 0x42 | funcID

ZW->HOST: REQ | 0x42 | funcID

Timeout: 1000ms

Exception Recovery: Resume normal operation, check nodelist to see if the controller has been reset. A controller MUST have nodeID ==1 after a set default.

4.5.25   ZW_SetLearnMode

void ZW_SetLearnMode (BYTE mode,
        VOID_CALLBACKFUNC(completedFunc)(LEARN_INFO *learnNodeInfo))

Macro: ZW_SET_LEARN_MODE(mode, func)

ZW_SetLearnMode is used to add or remove the controller to a Z-Wave network.

This function is used to instruct the controller to allow it to be added or removed from the network.

When a controller is added to the network the following things will happen:
1. If the current stored ID's are zero and the assigned ID's are nonzero, the received ID's will be stored (node was added to the network).
2. If the received ID's are zero the stored ID's will be set to zero (node was removed from the network).
3. The controller receives updates to the node list and the routing table but the ID's remain unchanged.

This function will probably change the capabilities of the controller so it is recommended that the application calls ZW_GetControllerCapabilities() after completion to check the controller status.

The **learnFunc** is called as the "Assign" process progresses. The returned nodeID is the nodes new Node ID. If no "Assign" is received from the including controller the callback function will not be called. It is then up to the application code to switch of Learn mode. Once the assignment process has been started the Callback function may be called more than once. The learn process is not complete before the callback function is called with LEARN_MODE_DONE.

Network wide inclusion should always be used as the default mode in inclusion to ensure compability with all implementations of Z-Wave controllers.

Correct way to call ZW_SetLearn() in a slave node:

```
ZW_SetLearnMode(ZW_SET_LEARN_MODE_NWI,

                learnFunc);
```

NOTE: Learn mode should only be enabled when necessary and disabled again as quickly as possible. It is recommended that learn mode is not enabling for more than 2 second in ZW_SET_LEARN_MODE_CLASSIC mode and 5 seconds in ZW_SET_LEARN_MODE_NWI mode.

NOTE: When the controller is already included into a network (secondary or inclusion controller) the callback status LEARN_MODE_STARTED will not be made but the LEARN_MODE_DONE/FAILED callback will be made as normal.

WARNING: The learn process should not be stopped with ZW_SetLearnMode(FALSE,..) between the LEARN_MODE_STARTED and the LEARN_MODE_DONE status callback.

Defined in: ZW_controller_api.h

Parameters:

mode IN   The learn node states are:

                          ZW_SET_LEARN_MODE_CLASSIC   Start the learn mode on the controller and only accept being included and excluded in direct range.

                          ZW_SET_LEARN_MODE_NWI   Start the learn mode on the controller and accept routed inclusion. NWI mode must not be used for exclusion.

                          ZW_SET_LEARN_MODE_DISABLE   Stop learn mode on the controller

completedFunc IN   Callback function pointer (Should only be NULL if state is turned off).

CONFIDENTIAL

Callback function Parameters (completedFunc):

*learnNodeInfo.bStatus IN	Status of learn mode:	
	LEARN_MODE_STARTED	The learn process has been started
	LEARN_MODE_DONE	The learn process is complete and the controller is now included into the network
	LEARN_MODE_FAILED	The learn process failed.
*learnNodeInfo.bSource IN	Node id of the new node	
*learnNodeInfo.pCmd IN	Pointer to Application Node information data (see **ApplicationNodeInformation** - nodeParm**)**. NULL if no information present.	
	The pCmd only contain information when bLen is not zero, so the information should be stored when that is the case. Regardless of the bStatus.	
*learnNodeInfo.bLen IN	Node info length.	

Serial API:

HOST->ZW: REQ | 0x50 | mode | funcID

ZW->HOST: REQ | 0x50 | funcID | bStatus | bSource | bLen | pCmd[ ]

Timeout: 15mins LEARN_MODE_STATED-> LEARN_MODE_DONE (See ZW_AddNodeToNetwork for more details on timing.)

Exception Recovery: ZW_SetLearnMode(FALSE,0), retry operation

CONFIDENTIAL
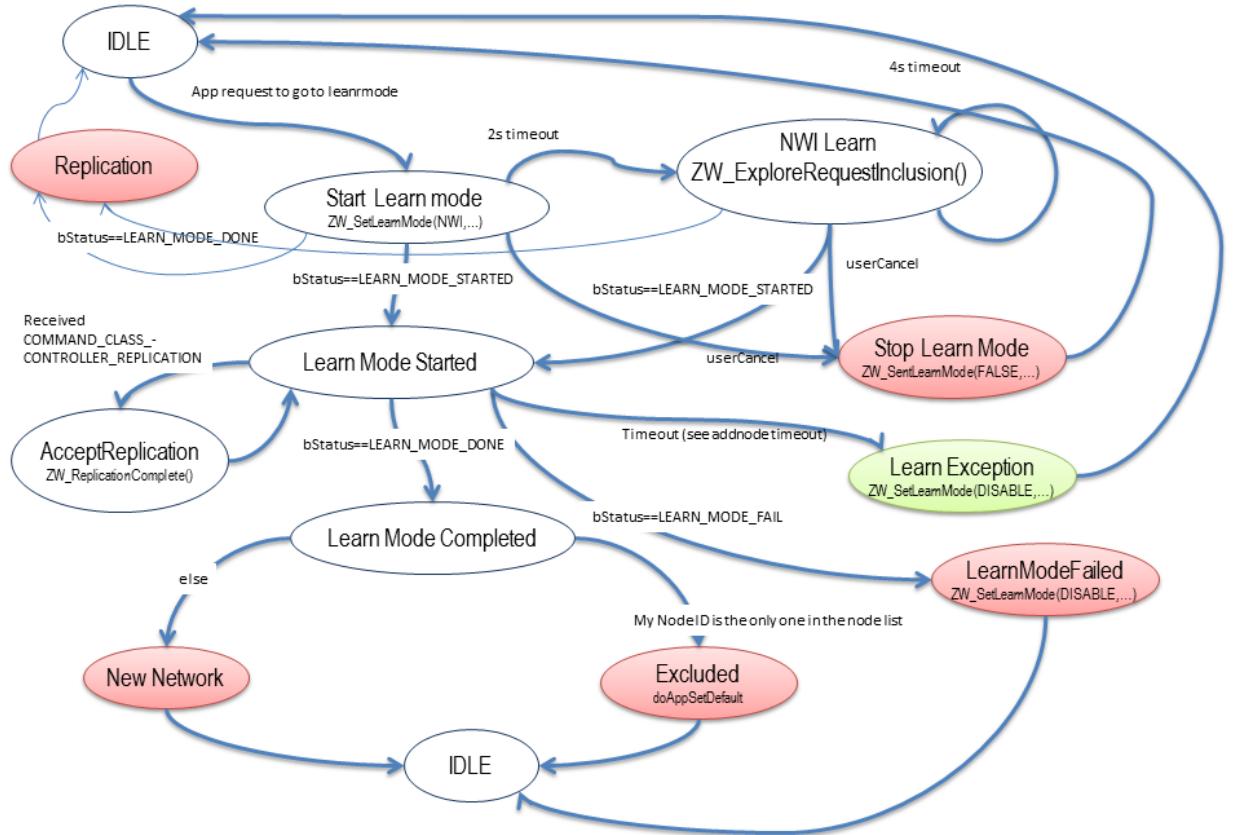
Figure 16. Statechart of Controller learnmode.

Note that is the controller isExcluded, it re-init its internal parameters like after a ZW_SetDefault

CONFIDENTIAL

4.5.26   ZW_SetRoutingInfo

void ZW_SetRoutingInfo(BYTE bNodeID,
                                    BYTE bLength,
                                    BYTE_P pMask)

Macro: ZW_SET_ROUTING_INFO(bNodeID, bLength, pMask)

NOTE: This function is not available in the Bridge Controller library and Static Controller library without repeater and manual routing functionality.

ZW_SetRoutingInfo is a function that can be used to overwrite the current neighbor information for a given node ID in the protocol locally.

The format of the routing info must be organised as follows:

pMask[i] (0 $\le$ i < (ZW_MAX_NODES/8)							
Bit	0	1	2	3	4	5	6	7
NodeID	i*8+1	i*8+2	i*8+3	i*8+4	i*8+5	i*8+6	i*8+7	i*8+8

If a bit n in pMask[i] is 1 it indicates that the node bNodeID has node (i*8)+n+1 as a neighbour. If n in pMask[i] is 0, bNodeID cannot reach node (i*8)+n+1 directly.

Defined in: ZW_controller_api.h

Return value:

BOOL	TRUE	Neighbor information updated successfully.
	FALSE	Failed to update neighbor information.

Parameters:

bNodeID IN   Node ID (1…232) to be updated with respect to neighbor information.

bLength IN   Routing info buffer length in bytes.

pMask IN   Pointer to buffer where routing info should be taken from. The buffer should be at least ZW_MAX_NODES/8 bytes

Serial API (Only Developer's Kit v4.5x):

HOST->ZW: REQ | 0x1B | bNodeID | NodeMask[29]

ZW->HOST: RES | 0x1B | retVal

CONFIDENTIAL

4.5.27   ZW_SetRoutingMAX

BOOL ZW_SetRoutingMAX(BYTE maxRouteTries)

Use this function to set the maximum number of source routing attempts before the next mechanism kicks-in. Default value with respect to maximum number of source routing attempts is five. The source routing mechanism requires the transmit option flags TRANSMIT_OPTION_ACK and TRANSMIT_OPTION_AUTO_ROUTE to be specified in send data API calls.

   Defined in: ZW_controller_api.h

   Parameters:

   maxRouteTries IN   1...20   Maximum number of source routing attempts

   Serial API:

   Not implemented

4.5.28   ZW_SetSUCNodeID

BYTE ZW_SetSUCNodeID (BYTE nodeID,
                        BYTE SUCState,
                        BYTE bTxOption,
                        BYTE capabilities,
                        VOID_CALLBACKFUNC (completedFunc)(BYTE txStatus))

Macro: ZW_SET_SUC_NODE_ID(nodeID, SUCState, bTxOption, capabilities, func)

Used to configure a static/bridge controller to be a SUC/SIS node or not. The primary controller should use this function to set a static/bridge controller to be the SUC/SIS node, or it could be used to stop previously chosen static/bridge controller being a SUC/SIS node.

A controller can set itself to a SUC/SIS by calling **ZW_EnableSUC** and **ZW_SetSUCNodeID** with its own node ID. It's recommended to do this when the Z-Wave network only comprise of the primary controller to get the SUC/SIS role distributed when new nodes are included. It's possible to include a virgin primary controller with SUC/SIS capabilities configured into another Z-Wave network.

   Defined in: ZW_controller_api.h

   Return value:

                TRUE   If the process of configuring the
                                                        static/bridge controller is started.

                FALSE   The process not started because the
                                                        calling controller is not the master or the
                                                        destination node is not a static/bridge
                                                        controller.

CONFIDENTIAL

Parameters:

nodeID IN   The node ID (1...232) of the static
                   controller to configure.

SUCState IN   TRUE   Want the static controller to be a SUC
                                                           node.

                   FALSE   If the static/bridge controller should not
                                                           be a SUC node.

bTxOption IN   TRUE   Want to send the frame with low
                                                           transmission power

                   FALSE   Want to send the frame at normal
                                                           transmission power

capabilities IN   SUC capabilities that is enabled:

                   ZW_SUC_FUNC_BASIC_SUC   Only enables the basic SUC functionality.

                   ZW_SUC_FUNC_NODEID_SERVER   Enable the node ID server functionality to
                                                           become a SIS.

completedFunc   Transmit complete call back.
IN

Callback function Parameters:

txStatus IN   Status of command:

                   ZW_SUC_SET_SUCCEEDED   The process ended successfully.

                   ZW_SUC_SET_FAILED   The process failed.

Serial API:

HOST->ZW: REQ | 0x54 | nodeID | SUCState | bTxOption | capabilities | funcID

ZW->HOST: RES | 0x54 | RetVal

ZW->HOST: REQ | 0x54 | funcID | txStatus

In case **ZW_SetSUCNodeID** is called locally with the controllers own node ID then only the response is
returned. In case true is returned in the response then it can be interpreted as the command is now
executed successfully.

Timeout: 65s

Faliure Recovery: retry operation twice, resume nomal operation.

CONFIDENTIAL

4.6   Z-Wave Static Controller API

The Static Controller application interface is an extended Controller application interface with added functionality specific for the Static Controller.

4.6.1   ZW_CreateNewPrimaryCtrl

Void ZW_CreateNewPrimaryCtrl(BYTE mode,
      VOID_CALLBACKFUNC(completedFunc)(LEARN_INFO *learnNodeInfo))

Macro: ZW_CREATE_NEW_PRIMARY_CTRL

ZW_CreateNewPrimaryCtrl is used to add a controller to the Z-Wave network as a replacement for the old primary controller.

This function has the same functionality as ZW_AddNodeToNetwork(ADD_NODE_CONTROLLER,…) except that the new controller will be a primary controller and it can only be called by a SUC. The function is not available if the SUC is a node ID server (SIS).

WARNING: This function should only be used when it is 100% certain that the original primary controller is lost or broken and will not return to the network.

    Defined in: ZW_controller_static_api.h

    Parameters:

mode IN	The learn node states are:	
	CREATE_PRIMARY_START	Start the process of adding a a new primary controller to the network.
	CREATE_PRIMARY_STOP	Stop the process.
	CREATE_PRIMARY_STOP_FAILED	Stop the inclusion and report a failure to the other controller.
completedFunc IN	Callback function pointer (Should only be NULL if state is turned off).	

CONFIDENTIAL

Callback function Parameters:

*learnNodeInfo.bStatus   IN Status of learn mode:

	ADD_NODE_STATUS_LEARN_READY	The controller is now ready to include a controller into the network.
	ADD_NODE_STATUS_NODE_FOUND	A controller that wants to be included into the network has been found
	ADD_NODE_STATUS_ADDING_CONTROLLER	A new controller has been added to the network
	ADD_NODE_STATUS_PROTOCOL_DONE	The protocol part of adding a controller is complete, the application can now send data to the new controller using **ZW_ReplicationSend()**
	ADD_NODE_STATUS_DONE	The new controller has now been included and the controller is ready to continue normal operation again.
	ADD_NODE_STATUS_FAILED	The learn process failed

*learnNodeInfo.bSource   IN Node id of the new node

*learnNodeInfo.pCmd   IN Pointer to Application Node information data (see **ApplicationNodeInformation** - nodeParm**)**. NULL if no information present.

The pCmd only contain information when bLen is not zero, so the information should be stored when that is the case. Regardless of the bStatus.

*learnNodeInfo.bLen   IN Node info length.

Timouts: see ZW_AddNodeToNetwork

Error recovery: see ZW_AddNodeToNetwork

CONFIDENTIAL

Serial API:

HOST->ZW: REQ | 0x4C | mode | funcID

ZW->HOST: REQ | 0x4C | funcID | bStatus | bSource | bLen | basic | generic | specific | cmdclasses[ ]

4.6.2   ZW_EnableSUC

BYTE ZW_EnableSUC (BYTE state, BYTE capabilities)

Macro: ZW_ENABLE_SUC (state, capabilities)

Used to enable/disable assignment of the SUC/SIS functionality in the controller. Assignment is default enabled. Assignment is done by the API call **ZW_SetSUCNodeID**.

If SUC is enabled then the static controller can store network changes sent from the primary, send network topology updates requested by controllers.

If SUC is disabled, then the static controller will ignore the frames sent from the primary controller after calling **ZW_SetSUCNodeID**. If the primary controller called **ZW_RequestNetWorkUpdate**, then the call back function will return with ZW_SUC_UPDATE_DISABLED.

Defined in: ZW_controller_static_api.h

Return value:

BYTE	TRUE	The SUC functionality was enabled/disabled.
	FALSE	Attempting to disable a running SUC, not allowed.

Parameters:

State IN	TRUE	SUC functionality is enabled.
	FALSE	SUC functionality is disabled.
capabilities IN	SUC capabilities that is enabled:	
	ZW_SUC_FUNC_BASIC_SUC	Only enables the basic SUC functionality.
	ZW_SUC_FUNC_NODEID_SERVER	Enable the SUC node ID server functionality to become a SIS.

Serial API:

HOST->ZW: REQ | 0x52 | state | capabilities

ZW->HOST: RES | 0x52 | retVal

CONFIDENTIAL

4.7　Z-Wave Bridge Controller API

The Bridge Controller application interface is an extended Controller application interface with added functionality specific for the Bridge Controller.

4.7.1　ZW_GetVirtualNodes

VOID ZW_GetVirtualNodes(BYTE *pnodeMask)

Macro: ZW_GET_VIRTUAL_NODES (pnodemask)

Request a buffer containing available Virtual Slave nodes in the Z-Wave network.

The format of the data returned in the buffer pointed to by pnodeMask is as follows:

pnodeMask[i] ($0 \leq i$ < (ZW_MAX_NODES/8)							
Bit	0	1	2	3	4	5	6	7
NodeID	i*8+1	i*8+2	i*8+3	i*8+4	i*8+5	i*8+6	i*8+7	i*8+8

If bit n in pnodeMask[i] is 1, it indicates that node (i*8)+n+1 is a Virtual Slave node. If bit n in pnodeMask[i] is 0, it indicates that node (i*8)+n+1 is not a Virtual Slave node.

Defined in:　　　ZW_controller_bridge_api.h

Parameters:

pNodeMask IN　　　Pointer to nodemask (29 byte size) buffer where the Virtual Slave nodeMask should be copied.

Serial API:

HOST->ZW: REQ | 0xA5

ZW->HOST: RES | 0xA5 | pnodeMask[29]

CONFIDENTIAL

4.7.2   ZW_IsVirtualNode

BYTE ZW_IsVirtualNode(BYTE nodeID)

Macro: ZW_IS_VIRTUAL_NODE (nodeid)

Checks if "nodeID" is a Virtual Slave node.

Defined in: ZW_controller_bridge_api.h

Return value:

BYTE	TRUE	If "nodeID" is a Virtual Slave node.
	FALSE	If "nodeID" is not a Virtual Slave node.

Parameters:

nodeID IN   Node ID (1...232) on node to check if it is
                 a Virtual Slave node.

Serial API:

HOST->ZW: REQ | 0xA6 | nodeID

ZW->HOST: RES | 0xA6 | retVal

CONFIDENTIAL

4.7.3   ZW_SendSlaveNodeInformation

BYTE ZW_SendSlaveNodeInformation(BYTE srcNode,
BYTE destNode,
BYTE txOptions,
VOID_CALLBACKFUNC(completedFunc)(BYTE txStatus))

Macro: ZW_SEND_SLAVE_NODE_INFO(srcnode, destnode, option, func)

Create and transmit a Virtual Slave node "Node Information" frame from Virtual Slave node srcNode. The Z-Wave transport layer builds a frame, request the application slave node information (see **ApplicationSlaveNodeInformation**) and queue the "Node Information" frame for transmission. The completed call back function (**completedFunc**) is called when the transmission is complete.

NOTE: ZW_SendSlaveNodeInformation uses the transmit queue in the API, so using other transmit functions before the complete callback has been called by the API might fail.

Defined in: ZW_controller_bridge_api.h

Return value:

BYTE	TRUE	If frame was put in the transmit queue.
	FALSE	If transmitter queue overflow or if bridge controller is primary or srcNode is invalid then completedFunc will NOT be called.

Parameters:

srcNode IN   Source Virtual Slave Node ID

destNode IN   Destination Node ID
(NODE_BROADCAST == all nodes)

txOptions   IN Transmit option flags:

	TRANSMIT_OPTION_LOW_POWER	Transmit at low output power level (1/3 of normal RF range). **NOTE:** The TRANSMIT_OPTION_LOW_POWER option should only be used when the two nodes that are communicating are close to each other (<2 meter). In all other cases this option should **not** be used.
	TRANSMIT_OPTION_ACK	Request acknowledge from destination node.

completedFunc   Transmit completed call back function
IN

Callback function Parameters:

txStatus   (see **ZW_SendData**)

Serial API:

CONFIDENTIAL

HOST->ZW: REQ | 0xA2 | srcNode | destNode | txOptions | funcID

ZW->HOST: RES | 0xA2 | retVal

ZW->HOST; REQ | 0xA2 | funcID | txStatus

4.7.4   ZW_SetSlaveLearnMode

BYTE ZW_SetSlaveLearnMode(BYTE node,
**   BYTE mode,**
**   VOID_CALLBACKFUNC(learnSlaveFunc)(BYTE state, BYTE orgID,**
**   BYTE newID))**

Macro: ZW_SET_SLAVE_LEARN_MODE (node, mode, func)

ZW_SetSlaveLearnMode enables the possibility for enabling or disabling "Slave Learn Mode", which when enabled makes it possible for other controllers (primary or inclusion controllers) to add or remove a Virtual Slave Node to the Z-Wave network. Also is it possible for the bridge controller (only when primary or inclusion controller) to add or remove a Virtual Slave Node without involving other controllers. Available Slave Learn Modes are:

VIRTUAL_SLAVE_LEARN_MODE_DISABLE – Disables the Slave Learn Mode so that no Virtual Slave Node can be added or removed.

VIRTUAL_SLAVE_LEARN_MODE_ENABLE – Enables the possibility for other Primary/Inclusion controllers to add or remove a Virtual Slave Node. To add a new Virtual Slave node to the Z-Wave Network the provided "node" ID must be ZERO and to make it possible to remove a specific Virtual Slave Node the provided "node" ID must be the nodeID for this specific (locally present) Virtual Slave Node. When the Slave Learn Mode has been enabled the Virtual Slave node must identify itself to the external Primary/Inclusion Controller node by sending a "Node Information" frame (see **ZW_SendSlaveNodeInformation**) to make the add/remove operation commence.

VIRTUAL_SLAVE_LEARN_MODE_ADD   - Add Virtual Slave Node to the Z-Wave network without involving any external controller. This Slave Learn Mode is only possible when bridge controller is either a Primary controller or an Inclusion controller.

VIRTUAL_SLAVE_LEARN_MODE_REMOVE - Remove a locally present Virtual Slave Node from the Z-Wave network without involving any external controller. This Slave Learn Mode is only possible when bridge controller is either a Primary controller or an Inclusion controller.

The **learnSlaveFunc** is called as the "Assign" process progresses. The returned "orgID" is the Virtual Slave node put into Slave Learn Mode, the "newID" is the new Node ID. If the Slave Learn Mode is VIRTUAL_SLAVE_LEARN_MODE_ENABLE and nothing is received from the assigning controller the callback function will not be called. It is then up to the main application code to switch of Slave Learn mode by setting the VIRTUAL_SLAVE_LEARN_MODE_DISABLE Slave Learn Mode. Once the assignment process has been started the Callback function may be called more than once.

NOTE: Slave Learn Mode should only be set to VIRTUAL_SLAVE_LEARN_MODE_ENABLE when necessary, and it should always be set to VIRTUAL_SLAVE_LEARN_MODE_DISABLE again as quickly as possible. It is recommended that Slave Learn Mode is never set to VIRTUAL_SLAVE_LEARN_MODE_ENABLE for more than 1 second.

CONFIDENTIAL

Defined in: ZW_controller_bridge_api.h

Return value:

BYTE	TRUE	If learnSlaveMode change was succesful.
	FALSE	If learnSlaveMode change could not be done.

Parameters:

node IN   Node ID (1...232) on node to set in Slave Learn Mode, ZERO if new node is to be learned.

mode IN   Valid modes:

VIRTUAL_SLAVE_LEARN_MODE_DISABLE	Disable Slave Learn Mode
VIRTUAL_SLAVE_LEARN_MODE_ENABLE	Enable Slave Learn Mode
VIRTUAL_SLAVE_LEARN_MODE_ADD	ADD: Create locally a Virtual Slave Node and add it to the Z-Wave network (only possible if Primary/Inclusion Controller).
VIRTUAL_SLAVE_LEARN_MODE_REMOVE	Remove locally present Virtual Slave Node from the Z-Wave network (only possible if Primary/Inclusion Controller).

learnFunc IN   Slave Learn mode complete call back function

CONFIDENTIAL

Callback function Parameters:

bStatus   Status of the assign process.

	ASSIGN_COMPLETE	Is returned by the callback function when in the VIRTUAL_SLAVE_LEARN_MODE_ENABLE Slave Learn Mode and assignment is done. Now the Application can continue normal operation.
	ASSIGN_NODEID_DONE	Node ID have been assigned. The "orgID" contains the node ID on the Virtual Slave Node who was put into Slave Learn Mode. The "newID" contains the new node ID for "orgID". If "newID" is ZERO then the "orgID" Virtual Slave node has been deleted and the assign operation is completed. When this status is received the Slave Learn Mode is complete for all Slave Learn Modes except the VIRTUAL_SLAVE_LEARN_MODE_ENABLE mode.
	ASSIGN_RANGE_INFO_UPDATE	Node is doing Neighbour discovery Application should not attempt to send any frames during this time, this is only applicable when in VIRTUAL_SLAVE_LEARN_MODE_ENABLE.

orgID   The original node ID that was put into Slave Learn Mode.

newID   The new Node ID. Zero if "OrgID" was deleted from the Z-Wave network.

Serial API:

HOST->ZW: REQ | 0xA4 | node | mode | funcID

ZW->HOST: RES | 0xA4 | retVal

ZW->HOST: REQ | 0xA4 | funcID | bStatus | OrgID | newID

4.8   Z-Wave Installer Controller API

The Installer application interface is basically an extended Controller interface that gives the application access to functions that can be used to create more advanced installation tools, which provide better diagnostics and error locating capabilities.

4.8.1   zwTransmitCount

BYTE zwTransmitCount

Macro: ZW_TX_COUNTER

ZW_TX_COUNTER is a variable that returns the number of transmits that the protocol has done since last reset of the variable. If the number returned is 255 then the number of transmits ≥ 255. The variable should be reset by the application, when it is to be restarted.

Defined in: ZW_controller_installer_api.h

Serial API:

To read the transmit counter:

HOST->ZW: REQ | 0x81| (FUNC_ID_GET_TX_COUNTER)

ZW->HOST: RES | 0x81 | ZW_TX_COUNTER (1 byte)

To reset the transmit counter:

HOST->ZW: REQ | 0x82| (FUNC_ID_RESET_TX_COUNTER)

CONFIDENTIAL

4.8.2   ZW_StoreHomeID

void ZW_StoreHomeID(BYTE_P pHomeID,
                                   BYTE bNodeID)
Macro: ZW_STORE_HOME_ID(pHomeID, NodeID)

ZW_StoreHomeID is a function that can be used to restore HomeID and NodeID information from a backup.

   Defined in: ZW_controller_installer_api.h

   Parameters:

   pHomeID IN   Pointer to HomeID structure to store

   bNodeID IN   NodeID to store.

   Serial API:

   HOST->ZW: REQ | 0x84 | pHomeID[0] | pHomeID[1] | pHomeID[2] | pHomeID[3] | bNodeID

CONFIDENTIAL

4.8.3   ZW_StoreNodeInfo

BOOL ZW_StoreNodeInfo(BYTE bNodeID,
                        BYTE_P pNodeInfo,
                        VOID_CALLBACKFUNC(func)())

Macro: ZW_STORE_NODE_INFO(NodeID,NodeInfo,function)

ZW_StoreNodeInfo is a function that can be used to restore protocol node information from a backup or the like. The format of the node info frame should be identical with the format used by ZW_GET_NODE_STATE.

   Defined in: ZW_controller_installer_api.h

   Return value:

   BOOL   TRUE   If NodeInfo was Stored.

                           FALSE   If NodeInfo was not Stored. (Illegal NodeId or
                                                   MemoryWrite failed)

   Parameters:

   bNodeID IN   Node ID (1...232) to store information at.

   pNodeInfo IN   Pointer to Node Information Frame.

   func IN   Callback function. Called when data has
                   been stored.

   Serial API:

   HOST->ZW: REQ | 0x83 | bNodeID | nodeInfo (nodeInfo is a NODEINFO field) | funcID

   ZW->HOST: RES | 0x83 | retVal

   ZW->HOST: REQ| 0x83 | funcId

4.9   Z-Wave Slave API

The Slave application interface is an extension to the Basis application interface enabling inclusion/exclusion of Routing Slave, and Enhanced Slave nodes.

4.9.1   ZW_SetDefault

void ZW_SetDefault(void)

Macros: ZW_SET_DEFAULT

This function set the slave back to the factory default state. Erase routing information and assigned homeID/nodeID from the EEPROM memory. Support of 9.6kbps communication only is also cleared and **ZW_Support9600Only** must be called to set it again. Finally write a new random home ID to the EEPROM memory.

   Defined in: ZW_slave_api.h

   Serial API:

   HOST->ZW: REQ | 0x42 | funcID

   ZW->HOST: REQ | 0x42 | funcID

CONFIDENTIAL

4.9.2   ZW_SetLearnMode

void ZW_SetLearnMode(BYTE mode,
                            VOID_CALLBACKFUNC(learnFunc)(BYTE bStatus, BYTE nodeID))

Macro: ZW_SET_LEARN_MODE(mode, func)

ZW_SetLearnMode enable or disable home and node ID's learn mode. Use this function to add a new Slave node to a Z-Wave network or to remove an already added node from the network again.

The Slave node must identify itself to the including controller node by sending a Node Information Frame (see **ZW_SendNodeInformation**).

When learn mode is enabled, the follwong two actions can be performed by the protocol:
1. If the current stored ID's are zero and the assigned ID's are nonzero, the received ID's will be stored (node was added to the network).
2. If the received ID's are zero the stored ID's will be set to zero (node was removed from the network).

The **learnFunc** is called as the "Assign" process progresses. The returned nodeID is the nodes new Node ID. If no "Assign" is received from the including controller the callback function will not be called. It is then up to the application code to switch of Learn mode. Once the assignment process has been started the Callback function may be called more than once. The learn process is not complete before the callback function is called with ASSIGN_COMPLETE.

Network wide inclusion should always be used as the default mode in inclusion to ensure compability with all implementations of Z-Wave controllers.

Correct way to call ZW_SetLearn() in a slave node:

```
ZW_SetLearnMode(ZW_SET_LEARN_MODE_NWI,

                learnFunc);
```

NOTE: Learn mode should only be enabled when necessary and disabled again as quickly as possible. It is recommended that learn mode is not enabled for more than 2 seconds in ZW_SET_LEARN_MODE_CLASSIC mode and 5 seconds in ZW_SET_LEARN_MODE_NWI mode.

  Defined in: ZW_slave_api.h

  Parameters:

mode IN	ZW_SET_LEARN_MODE_CLASSIC	Start the learn mode on the slave and only accept being included and excluded in direct range.
	ZW_SET_LEARN_MODE_NWI	Start the learn mode on the slave and accept routed inclusion. NWI mode must not be used for exclusion.
	ZW_SET_LEARN_MODE_DISABLE	Stop learn mode on the slave
learnFunc IN	Node ID learn mode completed call back function	

CONFIDENTIAL

Callback function Parameters:

bStatus   Status of the assign process

               ASSIGN_COMPLETE   Assignment is done and Application can
                                                                            continue normal operation.

               ASSIGN_NODEID_DONE   Node ID has been assigned. More
                                                                            information may follow.

               ASSIGN_RANGE_INFO_UPDATE   Node is doing Neighbor discovery
                                                                            Application should not attempt to send
                                                                            any frames during this time.

nodeID   The new (learned) Node ID (1...232)

Serial API:

HOST->ZW: REQ | 0x50 | mode | funcID

ZW->HOST: REQ | 0x50 | funcID | bstatus | nodeID

CONFIDENTIAL

4.9.3   ZW_Support9600Only

BOOL ZW_Support9600Only(BOOL bValue)

Macros: ZW_SUPPORT9600_ONLY(value)

The API call **ZW_Support9600Only** can select that non-listening ZW0201/ZW0301 slaves only want to support 9.6kbps communication. Support 9.6kbps only is cleared on RESET (this also includes WUT) and on calling **ZW_SetDefault**. Excluding the Node from the network will not clear support 9.6kbps only. The protocol will before sending still check for any ongoing 9.6 and 40kbps communication.

Important: Place this call **only** in ApplicationInitSW

Defined in: ZW_slave_api.h

Return value:

BOOL	TRUE	The baudrate change was succesfull.
	FALSE	Baudrate could not be changed because the node was listening.

Parameters:

bValue		Select if this node should only support 9.6kbit/s
	TRUE	This node will now act as a 9.6kbit/s
	FALSE	This node will respond on all supported baudrates

Serial API:

HOST->ZW: REQ | 0x5B | bValue

ZW->HOST: RES | 0x5B | retVal

CONFIDENTIAL

4.10   Z-Wave Routing and Enhanced Slave API

The Routing and Enhanced Slave application interface is an extension of the Basis and Slave application interface enabling control of other nodes in the Z-Wave network.

4.10.1   ZW_GetSUCNodeID

BYTE ZW_GetSUCNodeID(void)

Macro: ZW_GET_SUC_NODE_ID()

API call used to get the currently registered SUC node ID. A controller must have called **ZW_AssignSUCReturnRoute** before a SUC node ID is registered in the routing or enhanced slave.

  Defined in: ZW_slave_routing_api.h

  Return value:

  BYTE   The node ID (1..232) on the currently
                    registered SUC, if ZERO then no SUC
                    available.

  Serial API:

  HOST->ZW: REQ | 0x56

  ZW->HOST: RES | 0x56 | SUCNodeID

4.10.2   ZW_IsNodeWithinDirectRange

BYTE ZW_IsNodeWithinDirectRange(BYTE bNodeID)

Macro: ZW_IS_NODE_WITHIN_DIRECT_RANGE (bNodeID)

Check if the supplied nodeID is marked as being within direct range in any of the existing return routes.

  Defined in: ZW_slave_routing_api.h

  Return value:

TRUE	If node is within direct range
FALSE	If the node is beyond direct range or if status is unknown to the protocol

  Parameters:

  bNodeID IN   Node id to examine

CONFIDENTIAL

Serial API:

HOST->ZW: REQ | 0x5D | bNodeID

ZW->HOST: RES | 0x5D | retVal

4.10.3   ZW_RediscoveryNeeded

BYTE ZW_RediscoveryNeeded (BYTE bNodeID,
                           VOID_CALLBACKFUNC (completedFunc)(BYTE bStatus))

Macro: ZW_REDISCOVERY_NEEDED(nodeid, func)

This function can request a SUC/SIS controller to update the requesting nodes neighbors. The function will try to request a neighbor rediscovery from a SUC/SIS controller in the network. In order to reach a SUC/SIS controller it uses other nodes (bNodeID) in the network. The application must implement the algorithm for scanning the bNodeID's to find a node which can help.

If bNodeID supports this functionality (routing slave and enhanced slave libraries), bNodeID will try to contact a SUC/SIS controller on behalf of the node that requests the rediscovery. If the functionality is unsupported by bNodeID ZW_ROUTE_LOST_FAILED will be returned in the callback function and the next node can be tried.

The callback function is called when the request have been processed by the protocol.

   Defined in: ZW_slave_routing_api.h

Return value:

	FALSE	The node is busy doing another update.
	TRUE	The help process is started; status will come in the callback.

Parameters:

bNodeID IN   Node ID (1..232) to request help from

completedFunc   Transmit completed call back function
IN

CONFIDENTIAL

Callback function parameters:

	ZW_ROUTE_LOST_ACCEPT	The node bNodeID accepts to forward the help request. Wait for the next callback to determine the outcome of the rediscovery.

ZW_ROUTE_LOST_FAILED   The node bNodeID has responded it is unable to help and the application can try next node if it decides so.

ZW_ROUTE_UPDATE_ABORT   No reply was received before the protocol has timed out. The application can try the next node if it decides so.

ZW_ROUTE_UPDATE_DONE   The node bNodeID was able to contact a controller and the routing information has been updated.

Serial API:

HOST->ZW: REQ | 0x59 | bNodeID | funcID

ZW->HOST: RES | 0x59 | retVal

ZW->HOST: REQ | 0x59 | funcID | bStatus

CONFIDENTIAL

4.10.4   ZW_RequestNewRouteDestinations

BYTE ZW_RequestNewRouteDestinations(BYTE *pDestList,
                                              BYTE bDestListLen ,
                        VOID_CALLBACKFUNC (completedFunc)(BYTE bStatus))

Macro: ZW_REQUEST_NEW_ROUTE_DESTINATIONS (pdestList, destListLen, func)

Used to request new return route destinations from the SUC/SIS node.

NOTE: No more than the first ZW_MAX_RETURN_ROUTE_DESTINATIONS will be requested regardless of bDestListLen.

Defined in: ZW_slave_routing_api.h

Return value:

TRUE	If the updating process is started.
FALSE	If the requesting routing slave is busy or no SUC node known to the slave.

Parameters:

pDestList IN   Pointer to a list of new destinations for which
                    return routes is needed.

bDestListLen   Number of destinations contained in pDestList.

completedFunc   Transmit completed call back function
IN

Callback function parameters:

ZW_ROUTE_UPDATE_DONE	The update process is ended successfully
ZW_ROUTE_UPDATE_ABORT	The update process aborted because of error
ZW_ROUTE_UPDATE_WAIT	The SUC node is busy
ZW_ROUTE_UPDATE_DISABLED	The SUC functionality is disabled

Serial API:

HOST->ZW: REQ | 0x5C | destList[5] | funcID

ZW->HOST: RES | 0x5C | retVal

ZW->HOST: REQ | 0x5C | funcID | bStatus

CONFIDENTIAL

4.10.5   ZW_RequestNodeInfo

BOOL ZW_RequestNodeInfo (BYTE nodeID,
                                    VOID (*completedFunc)(BYTE txStatus))

Macro: ZW_REQUEST_NODE_INFO(NODEID)

This function is used to request the Node Information Frame from a slave based node in the network. The Node info is retrieved using the **ApplicationSlaveUpdate** callback function with the status UPDATE_STATE_NODE_INFO_RECEIVED. This call is also available for controllers.

Defined in: ZW_slave_routing_api.h

Return value:

BOOL	TRUE	If the request could be put in the transmit queue successfully.
	FALSE	If the request could not be put in the transmit queue. Request failed.

Parameters:

nodeID IN   The node ID (1…232) of the node to request the Node Information Frame from.

completedFunc   Transmit complete call back.
IN

Callback function Parameters:

txStatus IN   (see **ZW_SendData**)

Serial API:

HOST->ZW: REQ | 0x60 | NodeID

ZW->HOST: RES | 0x60 | retVal

The Serial API implementation do not return the callback function (no parameter in the Serial API frame refers to the callback), this is done via the **ApplicationSlaveUpdate** callback function:

- If request nodeinfo transmission was unsuccessful (no ACK received) then the **ApplicationSlaveUpdate** is called with UPDATE_STATE_NODE_INFO_REQ_FAILED (status only available in the Serial API implementation).

- If request nodeinfo transmission was successful there is no indication that it went well apart from the returned Nodeinfo frame which should be received via the **ApplicationSlaveUpdate** with status UPDATE_STATE_NODE_INFO_RECEIVED.

CONFIDENTIAL

4.11 Serial Command Line Debugger

The debug driver is a simple single line command interpreter, operated via the serial interface (UART – RS232). The command line debugger is used to dump and edit memory, including the memory mapped registers.

For a controller/slave_enhanced node the debugger startup by displaying the following help text on the debug terminal:

```
Z-Wave Commandline debugger Vx.nn
 Keyes(VT100): BS; ^,<,> arrows; F1.
H                             Help
D[X|E|F] <addr> [<length>]    Dump memory
E[X|E]   <addr>               Edit memory (Key: SP)
W[X|E|F] <addr>               Watch memory location
         is idata (80-FF is SFR)
   X     is xdata
     E   is External EEPROM
       F is flash
>
```

For a slave node the debugger startup by displaying the following help text on the debug terminal:

```
Z-Wave Commandline debugger Vx.nn
 Keyes(VT100): BS; ^,<,> arrows; F1.
H                             Help
D[X|I|F] <addr> [<length>]    Dump memory
E[X|I]   <addr>               Edit memory (Key: SP)
W[X|I|F] <addr>               Watch memory location
         is idata (80-FF is SFR)
   X     is xdata
     I   is "Internal EEPROM" flash
       F is flash
>
```

The command debugger is then ready to receive commands via the serial interface.

Special input keys:

F1   (function key 1)   same as the help command line.

BS   (backspace)   delete the character left to the curser.

< (left arrow)   move the cursor one character left.

> (right arrow)   move the cursor one character right.

^ (up arrow)   retrieve last command line.

CONFIDENTIAL

Commands:

H[elp]		Display the help text.
D[ump]	<addr> [<length>]	Dump idata (0-7F) or SFR memory (80-FF).
DX	<addr> [<length>]	Dump xdata (SRAM) memory.
DI	<addr> [<length>]	Dump "internal EEPROM" flash (slave only).
DE	<addr> [<length>]	Dump external EEPROM (controllers/slave_enhanced only).
DF	<addr> [<length>]	Dump FLASH memory.
E[dit]	<addr>	Edit idata (0-7F) or SFR memory (80-FF).
EX	<addr>	Edit xdata memory.
EI	<addr>	Edit "internal EEPROM" flash (slave only).
EE	<addr>	Edit external EEPROM (controllers/slave_enhanced only).
W[atch]	<addr>	Watch idata (0-7F) or SFR memory (80-FF).
WX	<addr>	Watch xdata memory.
WI	<addr>	Watch "internal EEPROM" flash (slave only).
WE	<addr>	Watch external EEPROM memory (controllers/slave_enhanced only).
WF	<addr>	Watch FLASH memory.

The Watch pointer gives the following log (when memory change):

idata SRAM memory   Rnn

xdata SRAM memory Xnn

Internal EEPROM   flash Inn   (slave only)

External EEPROM   Enn   (controllers/slave_enhanced only)

Examples:

```
>dx 0                   ; Edit offset 0x0000 and 0x0001 of xdata SRAM
0000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
>ex 0                   ; Edit offset 0x0000 and 0x0001 of xdata SRAM
0000 00-1 00-2
>dx 0                   ; Dump offset 0x0000 to 0x000f of xdata SRAM
0000  01 02 00 00 00 00 00 00  00 00 00 00 00 00 00 00
>wx 1X02                ; Watch offset 0x0001 of xdata SRAM
>ex 1
0001 02-1X01
>
```

CONFIDENTIAL

4.11.1   ZW_DebugInit

void ZW_DebugInit(WORD baudRate)

Macro: ZW_DEBUG_CMD_INIT(baud)

Command line debugger initialization. The macro can be placed within the application initialization function (see function **ApplicationInitSW**).

Example:

ZW_DEBUG_CMD_INIT(96); /* setup command line speed to 9600 bps. */

Defined in: ZW_debug_api.h

Parameters:

baudRate IN   Baud Rate / 100 (e.g. 96 = 9600 bps,
                   384 = 38400 bps, 1152 = 115200 bps)

Serial API (Not supported)

4.11.2   ZW_DebugPoll

void ZW_DebugPoll(void)

Macro: ZW_DEBUG_CMD_POLL

Command line debugger poll function. Collect characters from the debug terminal and execute the commands.

Should be called via the main poll loop (see function **ApplicationPoll**).

By using the debug macros (ZW_DEBUG_CMD_INIT, ZW_DEBUG_CMD_POLL) the command line debugger can be enabled by defining the compile flag "ZW_DEBUG_CMD" under CDEFINES in the makefile as follows:

CDEFINES+= EU,\
        ZW_DEBUG_CMD,\
        SUC_SUPPORT,\
        ASSOCIATION,\
        LOW_FOR_ON,\
        SIMPLELED

Both the debug output (ZW_DEBUG) and the command line debugger (ZW_DEBUG_CMD) can be enabled at the same time.

Defined in: ZW_debug_api.h

Serial API (Not supported)

CONFIDENTIAL

4.12   RF Settings in App_RFSetup.a51 file

The Z-Wave libraries are capable of transmitting/receiving on either 921.42MHz (ANZ) or 868.42MHz (EU) or 919.82MHz (HK) or 916.00MHz (IL) or 865.22MHz (IN) or 868.10MHz (MY) or 869.0MHz (RU) or 908.42MHz (US). The default frequency is set to US.

4.12.1   ZW0201/ZW0301 RF parameters

To allow for the selection of frequency, and transmit power levels (normal and low power) every application must have the App_RFSetup.a51 module linked in to define the const block placed in Flash memory. The ZW_RF020x.h / ZW_RF030x.h file contains various definitions such as default values:

Table 25. App_RFSetup.a51 module definitions for ZW0201/ZW0301

Offset to table start	Define name	Default value	Valid values	Description
0	FLASH_APPL_MAGIC_VALUE_OFFS	0xFF	0x42	If value is 0x42 then the table contents is valid.
1	FLASH_APPL_FREQ_OFFS	0x01	0x00-0x0A	0x00 = EU 0x01 = US 0x02 = ANZ 0x03 = HK 0x08 = MY 0x09 = IN 0x0A = RU 0x0B = IL 0xXX = use default.
2	FLASH_APPL_NORM_POWER_OFFS	0xFF		If 0xFF the default lib value is used: US = 0x2A EU = 0x2A ANZ = 0x2A HK = 0x2A IN = 0x2A MY = 0x2A RU = 0x2A IL = 0x2A
4	FLASH_APPL_LOW_POWER_OFFS	0xFF		If 0xFF the default lib value is used:0x14
5	FLASH_APPL_PLL_STEPUP_OFFS	0xFF	0x00	Only supported on ZW0301

An application programmer can select RF frequency (921.42MHz (ANZ) or 868.42MHz (EU) or 919.82MHz (HK) or 916.00MHz (IL) or 865.22MHz (IN) or 868.10MHz (MY) or 869.0MHz (RU) or 908.42MHz (US)), and the values for normal and low power transmissions by changing the const block defined in the App_RFSetup.a51 module. The RF frequency to use can be set by defining either ANZ or EU or HK or IL or IN or MY or RU or US in the application makefile. The TXnormal Power needs to be adjusted when making the FCC compliance test. According to the FCC part 15, the output radiated power shall not exceed 94dBuV/m. This radiated power is the result of the module output power and your product antenna gain. As the antenna gain is different from product to product, the module output power needs to be adjusted to comply with the FCC regulations.

When using an external PA, set the field at FLASH_APPL_PLL_STEPUP_OFFS to 0 (zero) for adjustment of the signal quality. This is necessary to be able to pass a FCC compliance test.

The match and power values can be adjusted directly on the module by the Z-Wave Programmer [14].

CONFIDENTIAL

5   HARDWARE SUPPORT DRIVERS

While the previous sections describe the generic Z-Wave modules that handle the wireless communication between the Z-Wave nodes, this section describe interfaces to common hardware components.

5.1   Hardware Pin Definitions

The hardware specific directories in the \product directory contain a ZW_pindefs.h file that defines macros for access to the I/O pins on the module.

Macros for accessing the I/O pins:

PIN_IN(pin, pullup)

Set I/O pin as input.

Parameters:

pin IN   Z-Wave pin name

pullup IN   If not zero activate the internal pull-up
                     resistor

Example:

PIN_IN(IO1,0) ; define pin IO1 as an input pin and disables the internal pull-up resistor.

NOTE: The pull-up feature is not available in the ZW010x ASIC

PIN_OUT(pin)

Set I/O pin as output.

Parameters:

pin IN   Z-Wave pin name

Example:

PIN_OUT(IO2)   ; define pin IO2 as an output pin.

CONFIDENTIAL

PIN_GET(pin)

Read pin value:

Parameters:

pin IN   Z-Wave pin name

Example:

if (PIN_GET(IO1))
  /* action when pin IO1 is 1 */

PIN_ON(pin)

Set output pin to 1 (on).

Parameters:

pin IN   Z-Wave pin name

Example:

PIN_ON(IO2); /* set pin IO2 to 1 */

PIN_OFF(pin)

Set output pin to 0 (off).

Parameters:

pin IN   Z-Wave pin name

Example:

PIN_OFF(IO2);/* set pin IO2 to 0 */

CONFIDENTIAL

PIN_TOGGLE(pin)

Toggle output pin.

> **Parameters:**

pin IN   Z-Wave pin name

> **Example:**

PIN_TOGGLE(IO2); /* toggle pin IO2 */

CONFIDENTIAL

6 APPLICATION NOTE: SUC/SIS IMPLEMENTATION

6.1   Implementing SUC support In All Nodes

Having Static Update Controller (SUC) support in Z-Wave products requires that several API calls must be used in the right order. This chapter provides details about how SUC support can be implemented in the different node types in the Z-Wave network.

6.2   Static Controllers

All static controllers has the functionality needed for acting as a SUC in the network, but it is up to the application to decide if it will allow the SUC functionality to be activated.

A Static Controller will not act as a SUC until the primary controller in the network has requested it to do so.

6.2.1   Request For Becoming SUC

The application in a static controller must enable for an assignment of the SUC capabilities by calling the **ZW_EnableSUC** The static controller will now accept to become SUC if/when the primary controller request it by calling **ZW_SetSUCNodeID**. In case assignment of the SUC capabilities is not enabled then the static controller will decline a SUC request from the primary controller.

NOTE: There can only be one SUC in a network, but there can be many static controllers that are enable for an assignment of the SUC capabilities in a network.

6.2.1.1   Request For Becoming a SUC Node ID Server (SIS)

Enabling assignment and requesting the SIS capabilities is done in a similar manner as for the SUC. The capability parameter in **ZW_EnableSUC** and **ZW_SetSUCNodeID** is used to indicate that a SIS is wanted and thereby accept becoming a SIS in the network.

NOTE: There can only be one SIS in a network, but there can be many static controllers that are enabled for an assignment of the SIS capabilities in a network. Even if the SIS functionality is enabled for an assignment in the static controller then the primary controller can still choose only to activate the basic SUC functionality.

CONFIDENTIAL

6.2.2   Updates From The Primary Controller
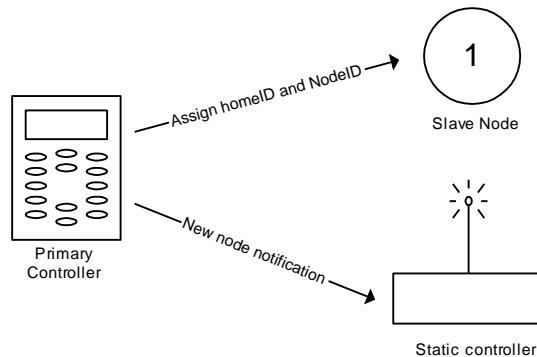
Figure 17. Inclusion of a node having a SUC in the network

When a new node is added to the network or an existing node is removed from the network the primary controller will send a network update to the SUC to notify the SUC about the changes in the network. The application in the SUC will be notified about such a change through the callback function **ApplicationControllerUpdate**). All update of node lists and routing tables is handled by the protocol so the call is just to notify the application in the static controller that a node has been added or removed.

6.2.3   Assigning SUC Routes To Routing Slaves

When the SUC is present in a Z-Wave network routing slaves can ask it for updates, but the routing slave must first be told that there is a SUC in the network and it must be told how to reach the SUC. That is done from the SUC by assigning a set of return routes to the routing slave so it knows how to reach the SUC. Assigning the routes to routing slaves is done by calling **ZW_AssignSUCReturnRoute** with the nodeID of the routing slave that should be configured.

NOTE: Routing slaves are not notified by the presence of a SUC as a part of the inclusion so it is always the Applications responsibility to tell a routing slave how it should reach the SUC.

6.2.4   Receiving Requests for Network Updates

When a SUC receives a request for sending network updates to a secondary controller or a routing slave, the protocol will handle all the communication needed for sending the update, so the application doesn't need to do anything and it will not get any notifications about the request.

6.2.5   Receiving Requests for new Node ID (SIS only)

When a SUC is configured to act as SIS in the system then it will receive requests for reserving node Ids for use when other controllers add nodes to the network. The protocol will handle all that communication without any involvement from the application.

6.3   The Primary Controller

The primary controller is responsible for choosing what static controller in the network that should act as a SUC and it will also send notifications to the SUC about all changes in the network topology. The application in a primary controller is responsible for choosing the static controller that should be the SUC. There is no fixed strategy for how to choose the static controller, so it is entirely up to the application to choose the controller that should become SUC. Once a static controller has been selected the

CONFIDENTIAL

application must use the **ZW_SetSUCNodeID** to request that the static controller becomes SUC. The capabilities parameter in the **ZW_SetSUCNodeID** call will determine if the primary controller enables the ID Server functionality in the SUC.

Once a SUC has been selected the protocol in the primary controller will automatically send notifications to the SUC about all changes in the network topology.

NOTE: A static controller can decline the role as SUC and in that case the callback function from **ZW_SetSUCNodeID** will return with a FAILED status. The static controller can also refuse to become SIS if that was what the primary controller requested, but accept to become a SUC.

6.4   Secondary Controllers

The secondary controllers in a network containing a SUC can ask the SUC for network topology changes and receive the updates from the SUC. It is entirely up to the application if and when an update is needed.
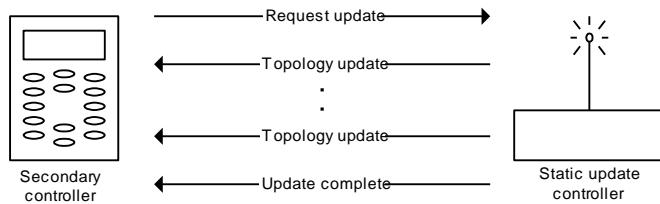
Figure 18. Requesting network updates from a SUC in the network

6.4.1   Knowing The SUC

The first thing the secondary controller should check is if it knows a SUC at all. Checking if a SUC is known by the controller is done with the **ZW_GetSUCNodeID** call and until this call returns a valid node ID the secondary controller can't use the SUC. The only time a secondary controller gets information about the presence of a SUC is during controller replication, so it is only necessary to check after a successful controller replication.

6.4.2   Asking For And Receiving Updates

If the secondary controller knows the SUC it can ask for updates from the SUC. Asking for updates is done using the **ZW_RequestNetWorkUpdate** function. If the call was successful the update process will start and the controller application will be notified about any changes in the network through calls to **ApplicationControllerUpdate**). Once the update process is completed the callback function provided in **ZW_RequestNetWorkUpdate** will be called.

If the callback functions returns with the status ZW_SUC_UPDATE_OVERFLOW then it means that there has been more that 64 changes made to the network since the last update of this secondary controller and it is therefore necessary to do a controller replication to get this secondary controller updated.

NOTE: The SUC can refuse to update the secondary controller for several reasons, and if that happens the callback function will return with a value explaining why the update request was refused.

WARNING: Consider carefully how often the topology of the network changes and how important it is for the application that the secondary controller is updated with the latest.

CONFIDENTIAL

6.5   Inclusion Controllers

When a SIS is present in a Z-Wave network then all the controllers that knows the SIS will change state
to Inclusion Controllers, and the concept of primary and secondary controllers will no longer apply for the
controllers. The Inclusion controllers has the functionality of a Secondary Controller so the functionality
described in section 6.4 also applies for secondary controllers, but Inclusion Controllers are also able to
include/exclude nodes to the network on behalf of the SIS. The application in a controller can check if a
SIS is present in the network by using the **ZW_GetControllerCapabilities** function call. This allows the
application to adjust the user interface according to the capabilities. If a SIS is present in the network
then the CONTROLLER_NODEID_SERVER_PRESENT bit will be set and the
CONTROLLER_IS_SECONDARY bit will not be set.
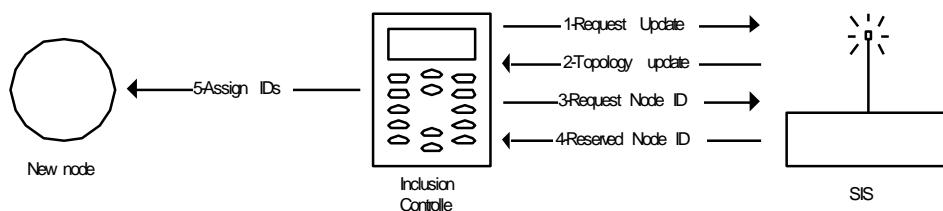
Figure 19. Inclusion of a node having a SIS in the network

6.6   Routing Slaves

The routing slave can request a update of its stored return routes from a SUC by using the
ZW_RequestNetWorkUpdate API call. There is no API call in the routing slave to check if the SUC is
known by the slave so the application must just try **ZW_RequestNetWorkUpdate** and then determine
from the return value if the SUC is known or not. If the SUC was known and the update was a success
then the routing slave would get a callback with the status SUC_UPDATE_DONE, the slave will not get
any notifications about what was changed in the network.

A static update controller (SUC) can help a battery-operated routing slave to be re-discovered in case it
is moved to a new location. The lost slave initiates the re-discovery process because it will be the first to
recognize that it is unable to reach the configured destinations and therefore can the application call
ZW_RediscoveryNeeded to request help from other nodes in the network.

The lost battery operated routing slave start to send "I'm lost" frames to each node beginning with node
ID = 1. It continue until it find a routing slave which can help it, i.e. the helping routing slave can obtain
contact with a SUC. Scanning through the node ID's is done on application level. Other strategies to
send the "I'm lost" frame can be implemented on the application level.

CONFIDENTIAL
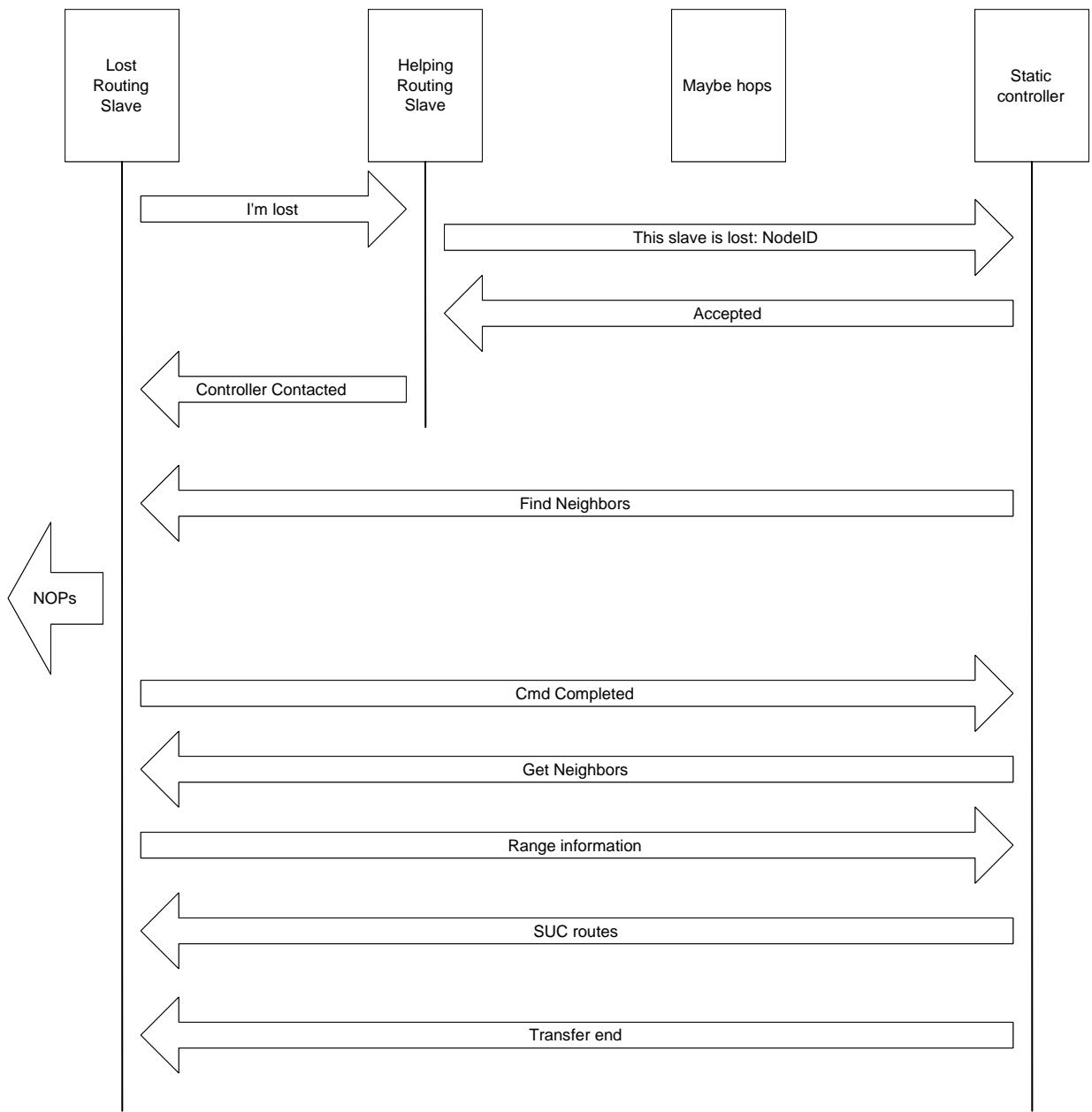
Figure 20. Lost routing slave frame flow

The helping routing slave must maximum use three hops to get to the controller, because it is the fourth hop when the controller issues the re-discovery to the lost routing slave. All handling in the helping slave is implemented on protocol level. In case a primary controller is found then it will check if a SUC exists in the network. In case a SUC is available it will be asked to execute the re-discovery procedure. When the controller receive the request "Re-discovery node ID x" it update the routing table with the new neighbor information. This allows the controller to execute a normal re-discovery procedure.

In case the **ZW_RediscoveryNeeded** was successful then the lost routing slave would get a callback with the status ZW_ROUTE_UPDATE_DONE and afterwards must the application call **ZW_RequestNetWorkUpdate** to obtain updated return routes from the SUC. See the Bin_Sensor_Battery sample code for an example of usage.

CONFIDENTIAL

7 APPLICATION NOTE: INCLUSION/EXCLUSION IMPLEMENTATION

This note describes the API calls the application layer needs to use when including new nodes to the network or excluding nodes from the network.

7.1   Including new nodes to the network

The API calls required by the including controller and the devices that is included are described. The callbacks as well as the steps the protocol takes without any application level involvement is also described. Finally it illustrates the frame flow between the two devices during the inclusion process.

The Z-Wave API calls **ZW_AddNodeToNetwork** and **ZW_SetLearnMode** are used to include nodes in a Z-Wave network. The primary/inclusion controller use the API call **ZW_AddNodeToNetwork** when including a node to the network and **ZW_SetLearnMode** is used by the controller or slave node that is to be included.

For the primary/inclusion controller that is including a node the **ZW_AddNodeToNetwork** is called with either:

ADD_NODE_ANY   Add any type of node to the network

ADD_NODE_SLAVE   Only add a node based on slave libraries

ADD_NODE_CONTROLLER   Only add a node based on controller libraries

ADD_NODE_EXISTING   Node is already in the network

To avoid the need to differentiate on the user interface whether it is a controller or slave the ADD_NODE_ANY can be used. The application can decide which actions to take based on the callback values.

ADD_NODE_SLAVE and ADD_NODE_CONTROLLER are available to support backward compatibility in case they are used on devices with separate slave and controller inclusion procedures.

ADD_NODE_EXISTING is useful when the controller application want the Node Information Frame from a node already included in the network.

The figure below illustrates the inclusion process between a primary/inclusion controller and a node that the user wishes to include in the network.

CONFIDENTIAL
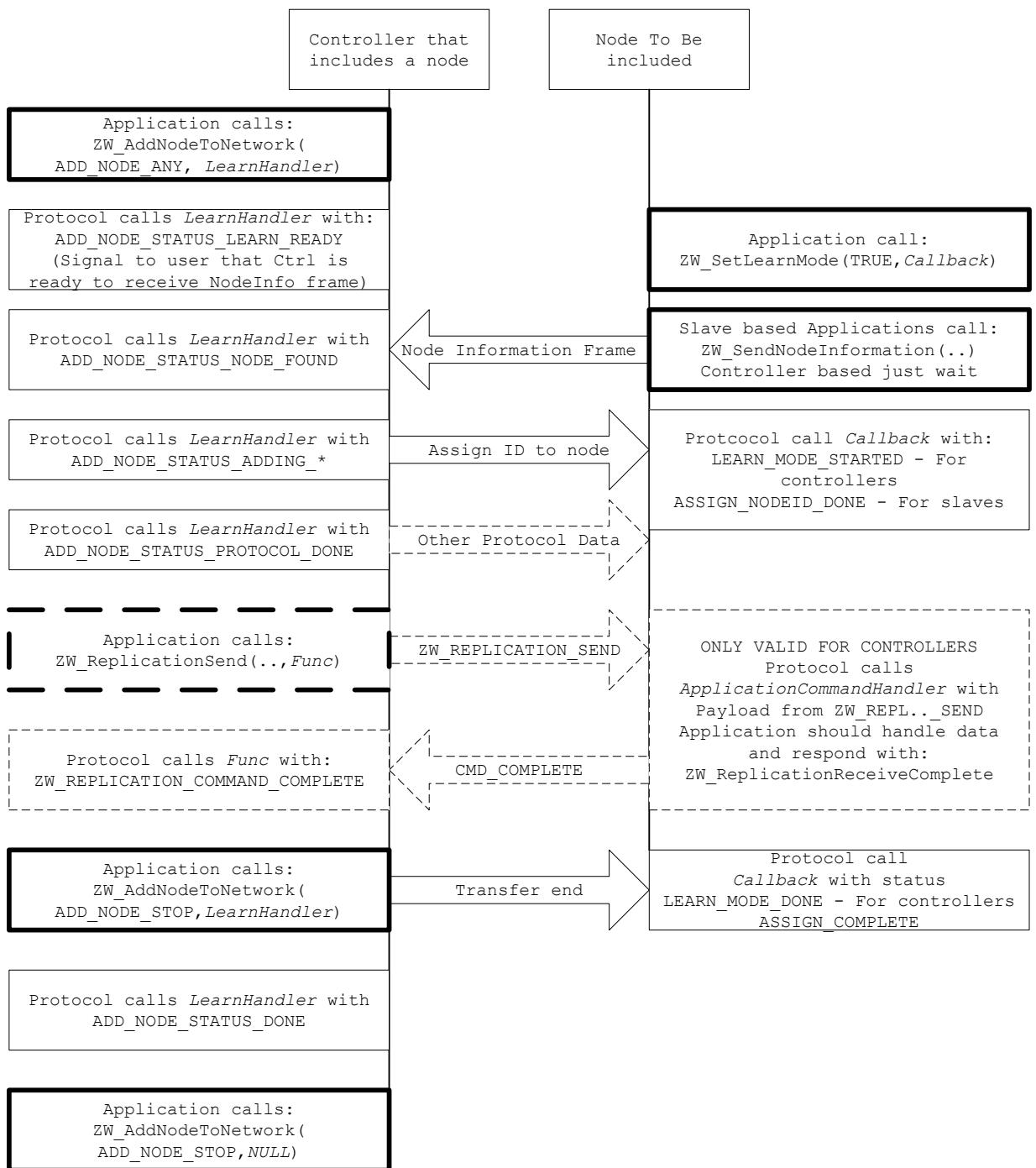
Figure 21. Node inclusion frame flow

Legend:
1. Bold frames indicate that the Application initiates an action.
2. Dashed frames indicate optional steps and frame flows.
3. *Italic* indicates a callback function specified by the application.

To allow the primary/inclusion controller in a Z-Wave network to include all kind of nodes, it is necessary to have a frame that describes the capabilities of a node. Some of the capabilities will be protocol related and some will be application specific. All nodes will automatically send out their Node Information Frame

CONFIDENTIAL

when the action button on the node is pressed. Once a node is included into the network it can always at a later stage get the node information from a node by requesting it with the API call **ZW_RequestNodeInfo**).

All slave nodes will per default start with Home ID is 0x00000000 and Node ID 0x00. All controllers will per default start with a unique Home ID and Node ID 0x01. Both have to be changed before the node can be included into a network. Furthermore the node must enter a learn mode state in order to accept assignment of new ID's. That state is communicated from the node by sending out a Node Information Frame as described. The primary/inclusion controller can now assign a Home and Node ID to the node to be included in the Z-Wave network. In case the node is already included to a network then the primary/inclusion controller refuses to include it.

During "Other protocol data" the network topology is discovered and updated. The primary/inclusion controller request the new node to check which of the current nodes in the network it can communicate directly with. In case a SUC/SIS is present in the network then the new node is informed about its presence and SUC return routes are transferred automatically. In case the SUC/SIS is created at a later stage, then the API call **ZW_AssignSUCReturnRoutes** can be used to allow the node to communicate with the SUC/SIS.

In case a controller is included then it's optional to transfer groups and scenes on application level using the Controller Replication command class [1]. This option is very handy, as it will save the user a lot of time reconfiguring the groups and scenes in the new controller. The Controller Replication command class must only be used in conjunction with a controller shift or when including a new controller to the network. The API call **ZW_ReplicationSend** must be used by the sending controller when transferring the group and scene command classes to another controller. The API call **ZW_ReplicationReceiveComplete** must be used by the receiving controller as acknowledge on application level because the data must first be stored in NVM before it can receive the next group or scene data.

A controller not supporting the Controller Replication Command Class must implement the acknowledge on application level when receiving Controller Replication commands to avoid that the sending controller is locked due to a missing acknowledge on application level. The receiving controller will then ignore the content of the Controller Replication commands but acknowledge on application level using the API call **ZW_ReplicationReceiveComplete**.

The following code sample shows how add node functionality is implemented on a controller capable of adding nodes to the network:

```
/* Call to be performed when user/application wants to include a node to the network
*/
ZW_AddNodeToNetwork(ADD_NODE_ANY, LearnHandler);

/*========================= LearnHandler =============================
**    Function description
**        Callback function to ZW_ADD_NODE_TO_NETWORK
**-------------------------------------------------------------------------*/
void LearnHandler(LEARN_INFO  *learnNodeInfo)
{
  if (learnNodeInfo->bStatus == ADD_NODE_STATUS_LEARN_READY)
  {
     /* Application should now signal to the user that we are ready to add a node.
     User may still choose to abort */
  }
  else if (learnNodeInfo->bStatus == ADD_NODE_STATUS_NODE_FOUND)
  {
     /* Protocol is busy adding node. User interaction should be disabled */
  }
  else if (learnNodeInfo->bStatus == ADD_NODE_STATUS_ADDING_SLAVE)
  {
     /* Protocol is still busy, this is just an information that it is a slave based
     unit that is being added */
  }
  else if (learnNodeInfo->bStatus == ADD_NODE_STATUS_ADDING_CONTROLLER)
  {
     /* Protocol is still busy, this is just an information that it is a controller
     based unit that is being added */
  }
  else if (learnNodeInfo->bStatus == ADD_NODE_STATUS_PROTOCOL_DONE)
  {
     /* Protocol is done. If it was a controller that was added, the application can
     now transfer information with ZW_ReplicationSend if any applications specific
     data that needs to be transferred to the included controller at inclusion time
     */

     /* When application is done it informs the protocol */
     ZW_AddNodeToNetwork(ADD_NODE_STOP, LearnHandler);
  }
  else if (learnNodeInfo->bStatus == ADD_NODE_STATUS_FAILED)
  {
     /* Add node failed - Application should indicate this to user */
     ZW_AddNodeToNetwork(ADD_NODE_STOP_FAILED, NULL);
  }
  else if (learnNodeInfo->bStatus == ADD_NODE_STATUS_DONE)
  {
     /* It is recommended to stop the process again here */
     ZW_AddNodeToNetwork(ADD_NODE_STOP, NULL);

     /* Add node is done. Application can move on Now is a good time to check if the
     added node should be set as SUC or SIS */
  }
}
```

CONFIDENTIAL

The following code samples show how an application typically implement the code needed in order to be able to include itself in an existing network.

Sample code for controller based devices:

```
/* Call to be performed when a controller wants to be include in the network */
ZW_SetLearnMode (TRUE, InclusionHandler);
/*Controller based devices just wait for the learn process to start*/

/*=========================== InclusionHandler ===========================
**                    Callback function to ZW_SetLearnMode
**----------------------------------------------------------------------*/
void InclusionHandler(
LEARN_INFO *learnNodeInfo)
{
  if ((*learnNodeInfo).bStatus == LEARN_MODE_STARTED)
  {
     /* The user should no longer be able to exit learn mode.
     ApplicationCommandHandler should be ready to handle ZW_REPLICATION_SEND_DATA
     frames if it supports transferring of Application specific data* /
  }
  else if ((*learnNodeInfo).bStatus == LEARN_MODE_FAILED)
  {
     /* Something went wrong - Signal to user */
  }
  else if ((*learnNodeInfo).bStatus == LEARN_MODE_DONE)
  {
     /* All data have been transmitted. Capabilities may have changed. Might be a
     good idea to read ZW_GET_CONTROLLER_CAPABILITIES() and to check that
     associations still are valid in order to check if the controller have been
     included or excluded from network*/
  }
}
```

CONFIDENTIAL

Sample code for slave based devices:

```
/* Call to be performed when a slave wants to be include in the network */
ZW_SetLearnMode(TRUE, InclusionHandler);
ZW_SendNodeInformation(NODE_BROADCAST, TRANSMIT_OPTION_LOW_POWER,....);

/*=========================== InclusionHandler ===========================
**                    Callback function to ZW_SetLearnMode
**------------------------------------------------------------------------*/
void InclusionHandler
  BYTE bStatus /* IN Current status of Learnmode*/
  BYTE nodeID) /* IN resulting nodeID - If 0x00 the node was removed from network*/
{
  if(bStatus == ASSIGN_RANGE_INFO_UPDATE)
  {
     /* Application should make sure that it does not send out NodeInfo now that we
     are updating range */
  }
  if(bStatus == ASSIGN_COMPLETE)
  {
     /* Assignment was complete. Check if it was inclusion or exclusion and maybe
     tell user we are done */
     if (nodeID != 0)
     {
        /* Node was included in a network*/

     }
     else
     {
        /* Node was excluded from a network. Reset any associations */
     }

  }
  else if (bStatus == ASSIGN_NODEID_DONE)
  {
     /* ID is assigned. Protocol will call with bStatus=ASSIGN_COMPLETE when done */
  }
}
```

7.2   Excluding nodes from the network

The API calls required by the controller that exclude and the device that is to be excluded is described. The callbacks as well as the steps the protocol takes without any application level involvement is also described. Finally it illustrates the frame flow between the two devices during the exclusion process.

The Z-Wave API calls **ZW_RemoveNodeFromNetwork** and **ZW_SetLearnMode** are used to exclude nodes from a Z-Wave network. The primary/inclusion controller use the API call ZW_RemoveNodeFromNetwork when removing a node from a network and **ZW_SetLearnMode** is used by the controller or slave node that is to be removed.

For the primary/inclusion controller that is including a node the **ZW_RemoveNodeFromNetwork** is called with either:

REMOVE_NODE_ANY   - Remove any type of node from the network

REMOVE_NODE_SLAVE   - Only remove a node based on slave libraries

REMOVE_NODE_CONTROLLER - Only remove a node based on controller libraries

CONFIDENTIAL

To avoid the need to differentiate on the user interface whether it is a controller or slave the REMOVE_NODE_ANY can be used. The application can decide which actions to take based on the callback values.

REMOVE_NODE_SLAVE and REMOVE_NODE_CONTROLLER are available to support backward compatibility in case they are used on devices with separate slave and controller exclusion procedures.

The figure below illustrates the exclusion process between a primary/inclusion controller and a node that the user wishes to exclude from the network.
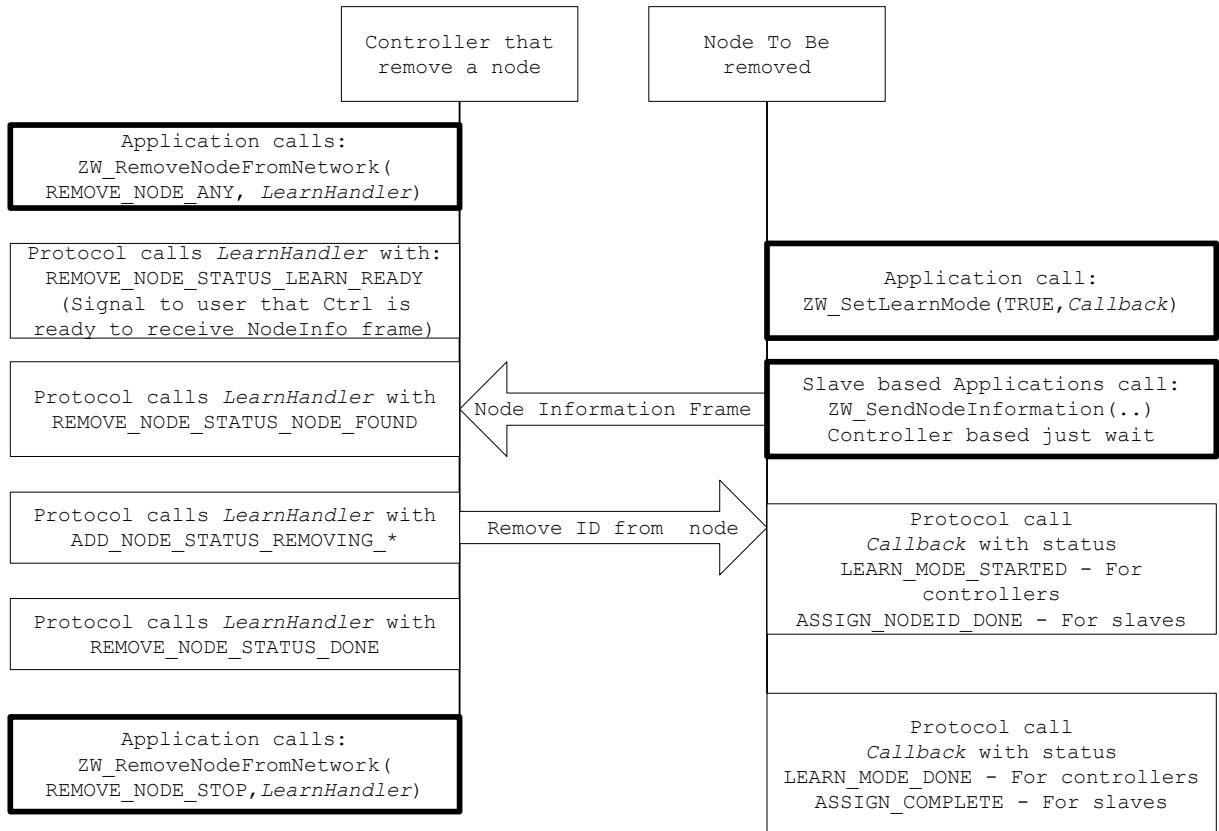
Figure 22. Node exclusion frame flow

CONFIDENTIAL

The following code sample shows how remove node functionality is implemented on a controller capable of removing nodes from the network:

```
/* Call to be performed when user/application wants to remove a node from the network
*/
ZW_RemoveNodeFromNetwork(REMOVE_NODE_ANY, LearnHandler);

/*========================= LearnHandler =============================
**    Function description
**        Callback function to ZW_RemoveNodeFromNetwork
**-------------------------------------------------------------------*/
void LearnHandler(LEARN_INFO  *learnNodeInfo)
{
  if (learnNodeInfo->bStatus == REMOVE_NODE_STATUS_LEARN_READY)
  {
     /* Application should now signal to the user that we are ready to remove a node.
     User may still choose to abort */
  }
  else if (learnNodeInfo->bStatus == REMOVE_NODE_STATUS_NODE_FOUND)
  {
     /* Protocol is busy removing node. User interaction should be disabled */
  }
  else if (learnNodeInfo->bStatus == REMOVE_NODE_STATUS_REMOVING_SLAVE)
  {
     /* Protocol is still busy, this is just an information that it is a slave based
     unit that is being removed*/
  }
  else if (learnNodeInfo->bStatus == REMOVE_NODE_STATUS_REMOVING_CONTROLLER)
  {
     /* Protocol is still busy, this is just an information that it is a controller
     based unit that is being removed */
  }
  else if (learnNodeInfo->bStatus == REMOVE_NODE_STATUS_ DONE)
  {
     /* Node is no longer part of the network*/

     /* When done - stop the process with */
     ZW_RemoveNodeFromNetwork(REMOVE_NODE_STOP, NULL);
  }
  else if (learnNodeInfo->bStatus == ADD_NODE_STATUS_FAILED)
  {
     /* Remove node failed - Application should indicate this to user */
     ZW_RemoveNodeFromNetwork(REMOVE_NODE_STOP, NULL);
  }
}
```

For the device that is excluded, the process is no different from an inclusion See paragraph 7 for sample code.

Applications based on Controller libraries should most likely check which capabilities the application should enable once the learn process is over. This includes reading **ZW_GetControllerCapabilities**.

Applications based on slave libraries should check the node ID returned to the callback function during the learn process if this node ID is zero the device is being excluded from the network and the application should most likely remove its network specific settings, such as associations.

CONFIDENTIAL

8   APPLICATION NOTE: CONTROLLER SHIFT IMPLEMENTATION

This note describes how a controller is able to include a new controller that after the inclusion will become the primary controller in the network. The controller that is taking over the primary functionality should just enter learn mode like when it is to be included in a network. The existing primary controller makes the controller change by calling **ZW_ControllerChange**(CONTROLLER_CHANGE_START,..).)

After a successfull change the controller that called **ZW_ControllerChange** will be secondary and no longer able to include devices.
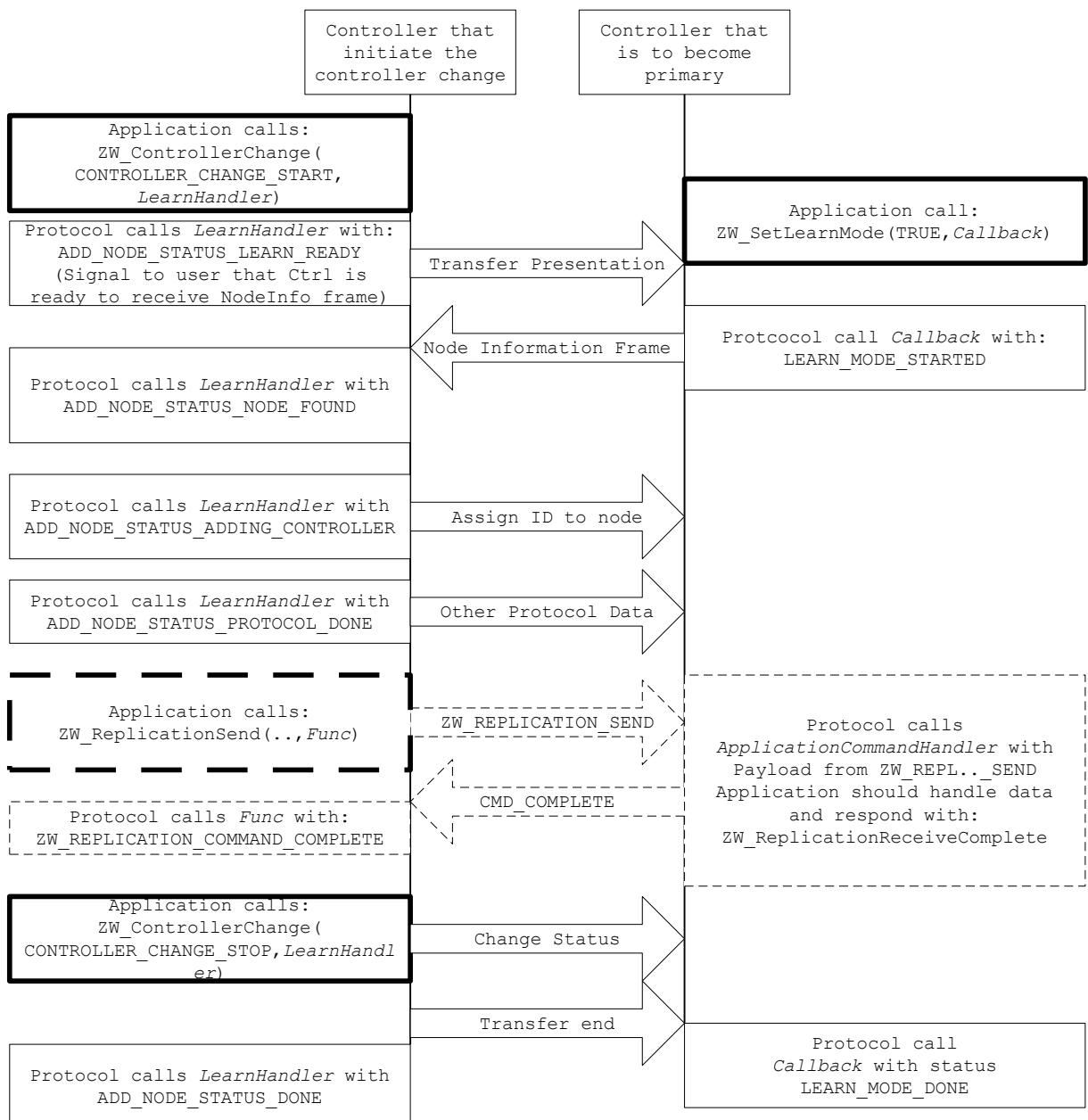
Figure 23. Controller shift frame flow

CONFIDENTIAL

9   REFERENCES

[1]   SD, SDS10242, Software Design Specification, Z-Wave Device Class Specification.
[2]   SD, DSH10086, Datasheet, ZW0x0x Z-Wave Interface Module.
[3]   SD, DSH10087, Datasheet, ZW0x0x Z-Wave Development Module.
[4]   SD, DSH10033, Datasheet, ZM1220 Z-Wave Module.
[5]   SD, DSH10034, Datasheet, ZM1206 Z-Wave Module.
[6]   SD, INS10240, Instruction, PC Based Controller User Guide.
[7]   SD, INS10241, Instruction, PC Installer Tool Application User Guide.
[8]   SD, INS10245, Instruction, Z-Wave Bridge User Guide.
[9]   SD, INS10029, Instruction, ZW0102 Single Chip Implementation Guideline.
[10]   SD, APL10312, Application Note, Programming the 200 and 300 Series Z-Wave Single Chip
        Flash.
[11]   SD, INS10336, Instruction, Z-Wave Reliability Test Guideline.
[12]   SD, INS10249, Instruction, Z-Wave Zniffer User Guide.
[13]   SD, INS10250, Instruction, Z-Wave DLL User's Manual.
[14]   SD, INS10679, Instruction, Z-Wave Programmer User Guide.
[15]   SD, INS10236, Instruction, Development Controller User Guide.
[16]   SD, INS10579, Instruction, Programming the ZW0102 Flash and Lock Bits.
[17]   SD, DSH10088, Datasheet ZMxx06 Converter Module.
[18]   SD, DSH10230, Datasheet, ZM2106C Z-Wave Module.
[19]   SD, INS10326, Instruction, ZW0201 Single Chip Implementation Guidelines.
[20]   SD, SRN11886, Software Release Note, ZW0201/ZW0301 Developer's Kit v4.53.00.
[21]   SD, APL10512, Application Note, Battery Operated Applications Using the ZW0201/ZW0301.
[22]   SD, DSH10856, Datasheet, ZM3106C Z-Wave Module.
[23]   SD, DSH10275, Datasheet, ZM2120C Z-Wave Module.
[24]   SD, DSH10857, Datasheet, ZM3120C Z-Wave Module.
[25]   SD, APL10292, Application Note, ZW0102 Triac Controller Guideline.
[26]   SD, APL10370, Application Note, ZW0201/ZW0301 Triac Controller Guideline.
[27]   SD, APL10514, Application Note, The ZW0201/ZW0301 ADC.
[28]   SD, INS10680, Instruction, Z-Wave XML Editor.
[29]   SD, INS11018, Instruction, Secure PC Based Controller User Guide (OBSOLETE, see INS10240).
[30]   SD, INS10681, Instruction, Secure Development Controller (AVR) User Guide.
[31]   SD, DSH10704, Datasheet, ZDP02A Z-Wave Development Platform.
[32]   SD, DSH11243, Datasheet, ZDP03A Z-Wave Development Platform.
[33]   SD, SDS11060, Software Design Specification, Z-Wave Command Class Specification.
[34]   SD, INS11072, Instruction, Z-Wave Programmer Communication Protocol.
[35]   SD, APL10742, Application Note, ZM3102N with External PA and Switch.
[36]   SD, INS11552, Instruction, 400 Series Crystal Calibration User Guide.
[37]   IETF RFC 2119, Key words for use in RFC's to Indicate Requirement Levels,
        http://tools.ietf.org/pdf/rfc2119.pdf
[38]   SD, INS12351, Instruction, Z-Wave ZW0201/ZW0301 Series Developer's Kit v4.54.01 Contents.

CONFIDENTIAL

INDEX

CONFIDENTIAL

CONFIDENTIAL

CONFIDENTIAL

CONFIDENTIAL

CONFIDENTIAL

CONFIDENTIAL

CONFIDENTIAL