Instruction

Z-Wave 400 Series Appl. Prg. Guide v6.02.00

Document No.:	INS12034
Version:	2
Description:	Guideline for developing 400 Series based applications using the application programming interface (API) based on Developer's Kit v6.0x
Written By:	JFR;JBU
Date:	2012-05-25
Reviewed By:	CHL;PSH;BBR
Restrictions:	Partners Only

Approved by:

Date	CET	Initials	Name	Justification
2012-05-25	14:07:39	NTJ	Niels Thybo Johansen	

REVISION RECORD

Doc. Rev	Date	By	Pages affected	Brief description of changes
1	20091206	JFR	All	Based on INS10682-1 Z-Wave Z-Wave 400 Series Appl. Prg. Guide v6.10.00
1	20120111	PSH JFR	5.1 5.4.3	Added API using guidelines Removed JP only API call ZW_SetOneChannelTransmit()
2	20120509	JFR	5.4.1.11 5.4.9.1	Added ApplicationRFNotify to support to external pow er amplifier (PA) Updated ZW_SetSleepMode description w rt. beamCount and POR
2	20120524	JBU	5.5.3	Documented TRANSMIT_COMPLETE_NOROUTE callback.

Table of Contents

List of Figures

List of Tables

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
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
CR	Carriage Return, move the position of the cursor to the first position on the same line
DLL	Dynamic Link Library
DUT	Device Under Test
ECB	Electronic CookBook (block cipher mode)
ERTT	Enhanced Reliability Test tool
EU	Europe
FET	Field-Effect Transistor
GNU	An organization devoted to the creation and support of Open Source software
HK	Hong Kong
HW	Hardware
IGBT	Insulated Gate Bipolar Transistor
IN	India
ISR	Interrupt Service Routines
LF	Line Feed, Move cursor to the next line
LRC	Longitudinal Redundancy Check
MTP	Many Times Programmable memory
MY	Malaysia
NAK	Not Acknowledged
NWI	Network Wide Inclusion
OTP	One time programmable memory
PA	Power Amplifier
POR	Power On Reset
PRNG	Pseudo-Random Number Generator
PWM	Pulse Width Modulator
R&D	Research and Development
RF	Radio Frequency
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 and how to build a complete program and load it on a 400 Series Z-Wave module. The document is also API reference guide for programmers.

2.2 Audience and Prerequisites

The audience is Sigma Designs R&D and external R&D software application programmers. The programmer should be familiar with the PK51 Keil Development Tool Kit for 8051 micro controllers and the GNU make utility.

3 Z-WAVE SOFTWARE ARCHITECTURE

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

The software is split into two groups of program modules: Z-Wave basis software and 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.

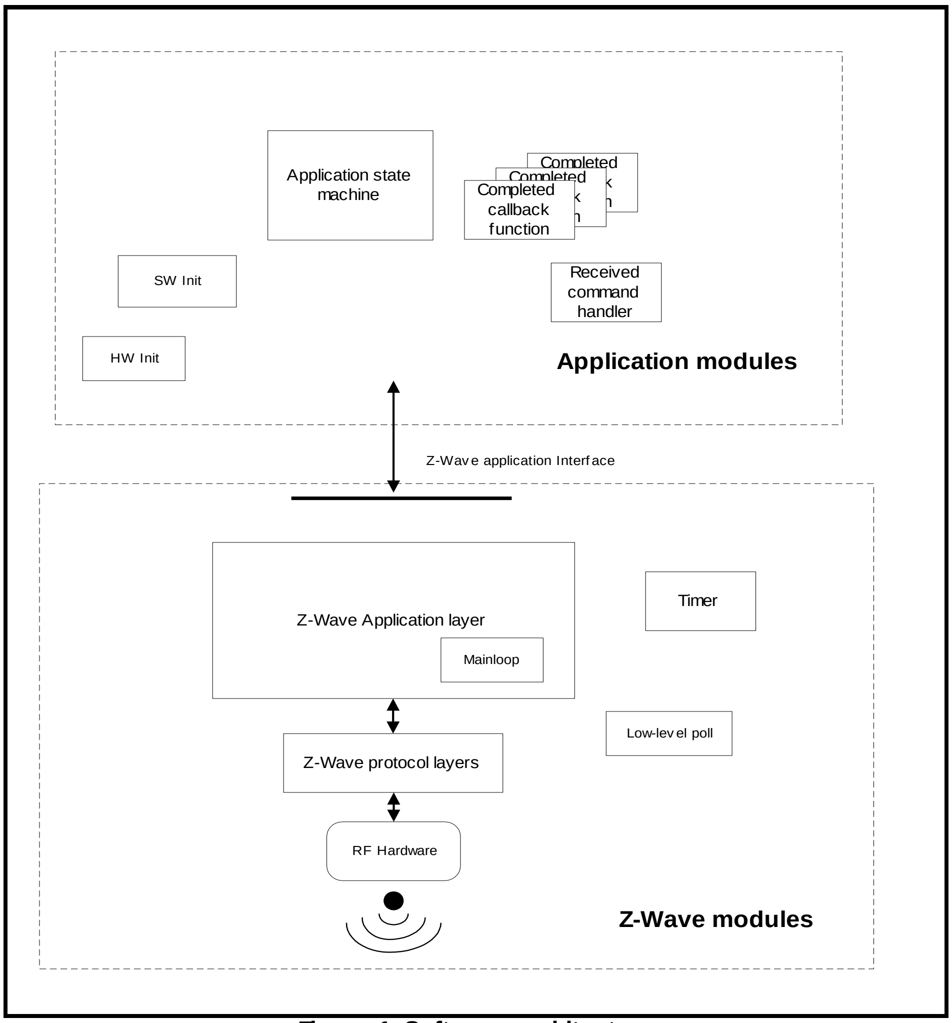

Figure 1. Software architecture

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 initializing the Z-Wave software (including the software timer used by the timer module) and finally calling the application software initialization function **ApplicationInitSW**. Execution then proceeds in the Z-Wave main loop.

3.2 Z-Wave Main Loop

The Z-Wave main loop will call the list of Z-Wave protocol functions, including the **ApplicationPoll** function. Hence, the functions must 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 RF data, transfer data via the UART, check user-activated buttons; “simultaneously” etc.

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 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 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 System Layer

The Systemt 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, a framework is 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. No precautions can unfortunately 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:

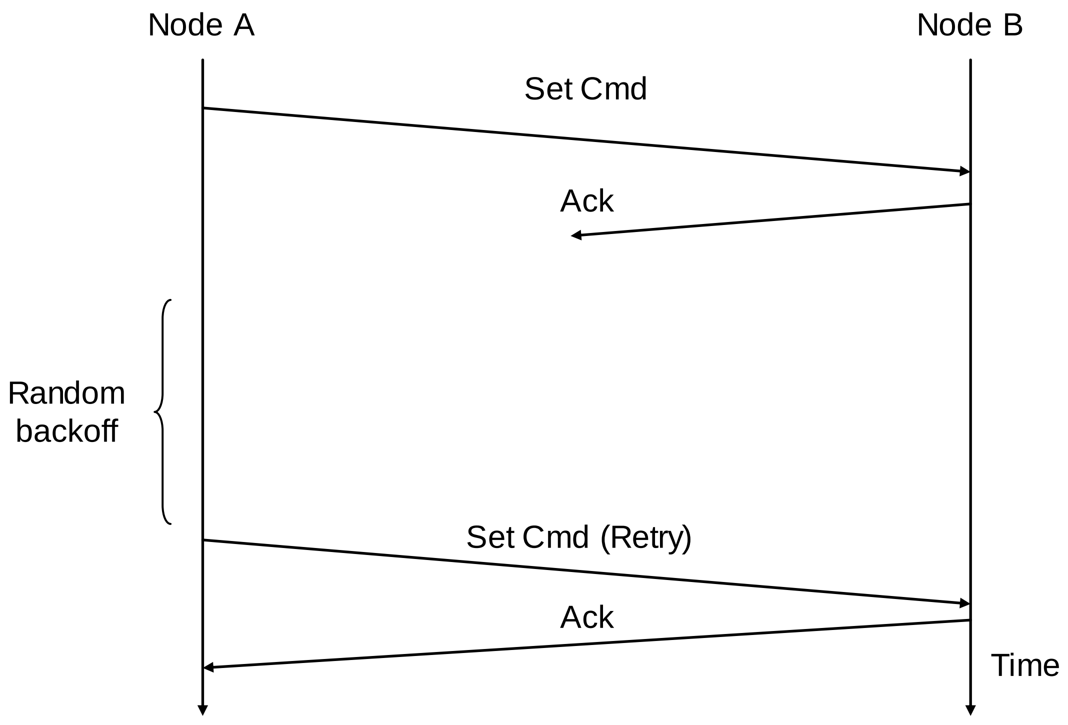

Figure 2. Multiple copies of the same Set frame

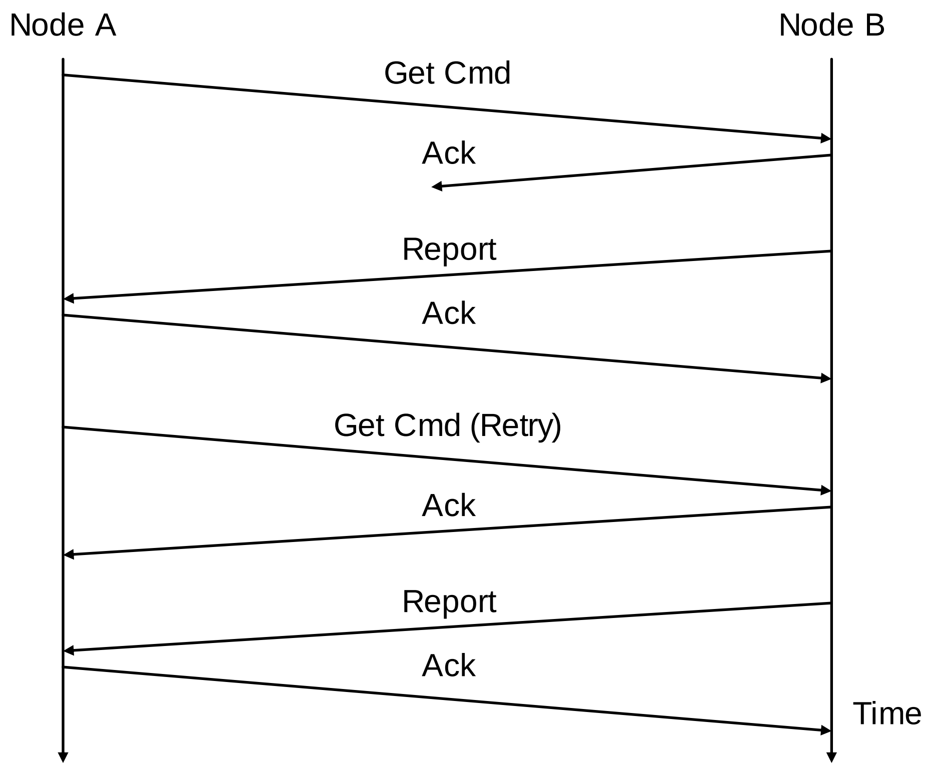

Figure 3. Multiple copies of the same Get/Report frame

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.

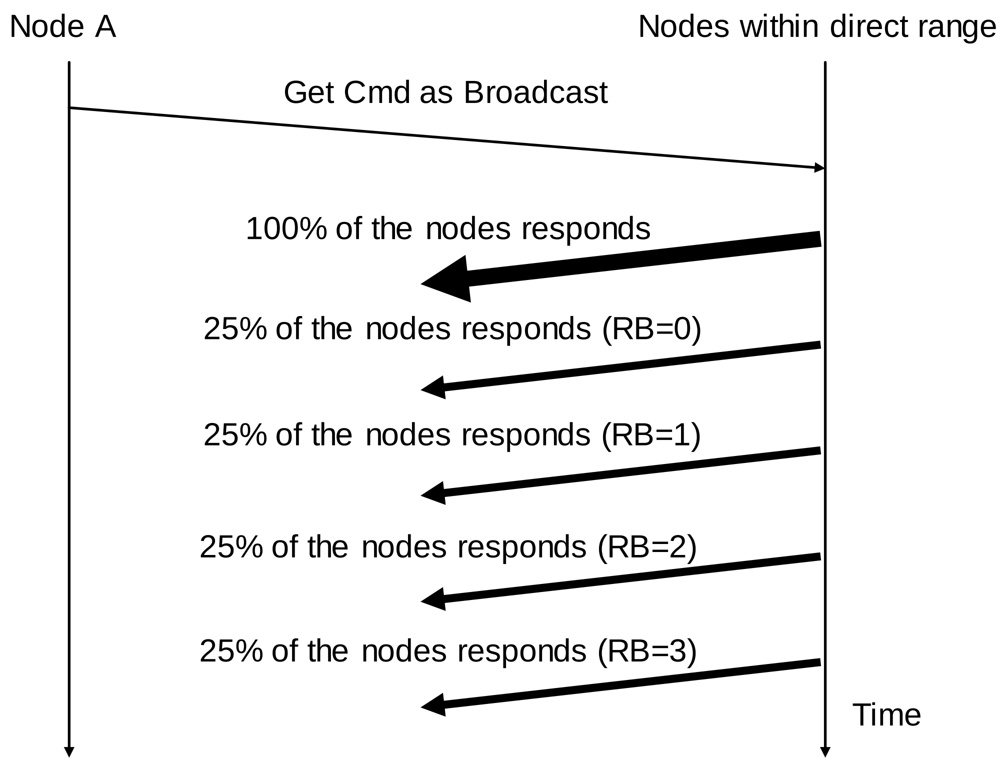
Node A
Nodes within direct range
Get Cmd as Broadcast
100% of the nodes responds
25% of the nodes responds (RB=0)
25% of the nodes responds (RB=1)
25% of the nodes responds (RB=2)
25% of the nodes responds (RB=3)
Time

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.

3.5 Z-Wave Software Timers

The Z-Wave timer module is designed to handle a limited number of simultaneous active software timers.

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 runs out.

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

3.6 Z-Wave Hardware Timers

The 100/200/300/400 Series Z-Wave Single Chips have 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. 100/200/300/400 Series Z-Wave Single Chips hardware timer allocation

	100 Series	200 Series	300 Series	400 Series
TIMER0	Available for the application	Protocol system clock	Protocol system clock	Available for the application
TIMER2	Available for the application	Available for the application	Available for the application	Available for the application
TIMER3	Protocol system clock	Not available	Not available	Not available

The TIMER0 and TIMER1 are standard 8051 timers/counters. TIMER1 is used by the protocol.

3.7 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. 100/200/300/400 Series Z-Wave Single Chip Application ISR availability

100 Series	200 Series	300 Series	400 Series
INUM_INT0 INUM_TIMER0 INUM_TIMER1 INUM_TIMER2 INUM_SERIAL0 INUM_ADC	INUM_INT0 INUM_INT1 INUM_TIMER1 INUM_SERIAL INUM_SPI INUM_TRIAC INUM_GP_TIMER INUM_ADC	INUM_INT0 INUM_INT1 INUM_TIMER1 INUM_SERIAL INUM_SPI INUM_TRIAC INUM_GP_TIMER INUM_ADC	INUM_INT0 INUM_INT1 INUM_TIMER0 INUM_SERIAL0 INUM_SPI0 INUM_TRIAC INUM_GP_TIMER INUM_ADC INUM_USB INUM_IR

Refer to ZW010x.h ZW020x, ZW030x.h and ZW040x.h header files with respect to ISR definitions. For an example, refer to UART ISR in serial API sample application.

3.8 Interrupt service routines.

When using interrupt service routines from one of the hardware interfaces such as ADC, GP timer or UART, then one should be aware about two things as described in the following two sections.

3.8.1 SFR pages

The 400 Series Z-Wave Single Chip uses multiple pages of 8051 SFR registers. The page selection is set using SFRPAGE. Consequently the SFRPAGE must be preserved when calling an Interrupt Service Routine (ISR) in your code. In order to do this the intrinsic functions _push_() and _pop_() must be called. Function _push_() must be called when the ISR starts, and _pop_() just before returning from the ISR.

For example, the ISR of the ADC should be look as follow:

```
void ADC_int(void) interrupt INUM_ADC
{
  _push_(SFRPAGE)†;

  call api's
 _pop_(SFRPAGE);
}
```

3.8.2 Calling functions from ISR

The 8051 core of the 400 Series Z-Wave Single Chip has no register-to-register move. Therefore, the compiler generates register to memory moves instead. Since the compiler knows the register bank, the physical address of a register in a register bank can be calculated. For example, when the compiler calculates the address of register R2 in register bank 0, the address is 0x02. If the register bank selected is not really 0, then the function overwrites this register. This might result in unpredictable behavior of the program.
This technique of accessing a register using its absolute address is called absolute register addressing.

In the Z-Wave system the system timer and RF interrupt use register bank 1. The default register bank used for non-interrupt code is register bank 0. Therefore, if a function is called from an ISR it might be looking in the wrong place for its register values.

To solve this problem, one of these solutions can be used:

1. Use the C51's REGISTERBANK directive to specify that a certain function uses the same register bank as the ISR that calls the function. Hence, no code is generated in the function to switch the register bank. For example:
   ```
   #pragma registerbank(1)
   void foo (void)
   {
   }
   ```
2. Use the NOAREGS directive to specify that the compiler should not use absolute register addressing. This make the function register bank independent so that it may be called from any function that uses a different register bank than the default.

† The _push_ and _pop_ functions are intrinsic functions and the header file INTRINS.H. Therefore, INTRINS.H should be included in order to be able to use them.

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 non-volatile memory.

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 header file “ZW_nvm_addr.h” contains a definition of the physical application memory offset NVM_APPL_OFFSET.

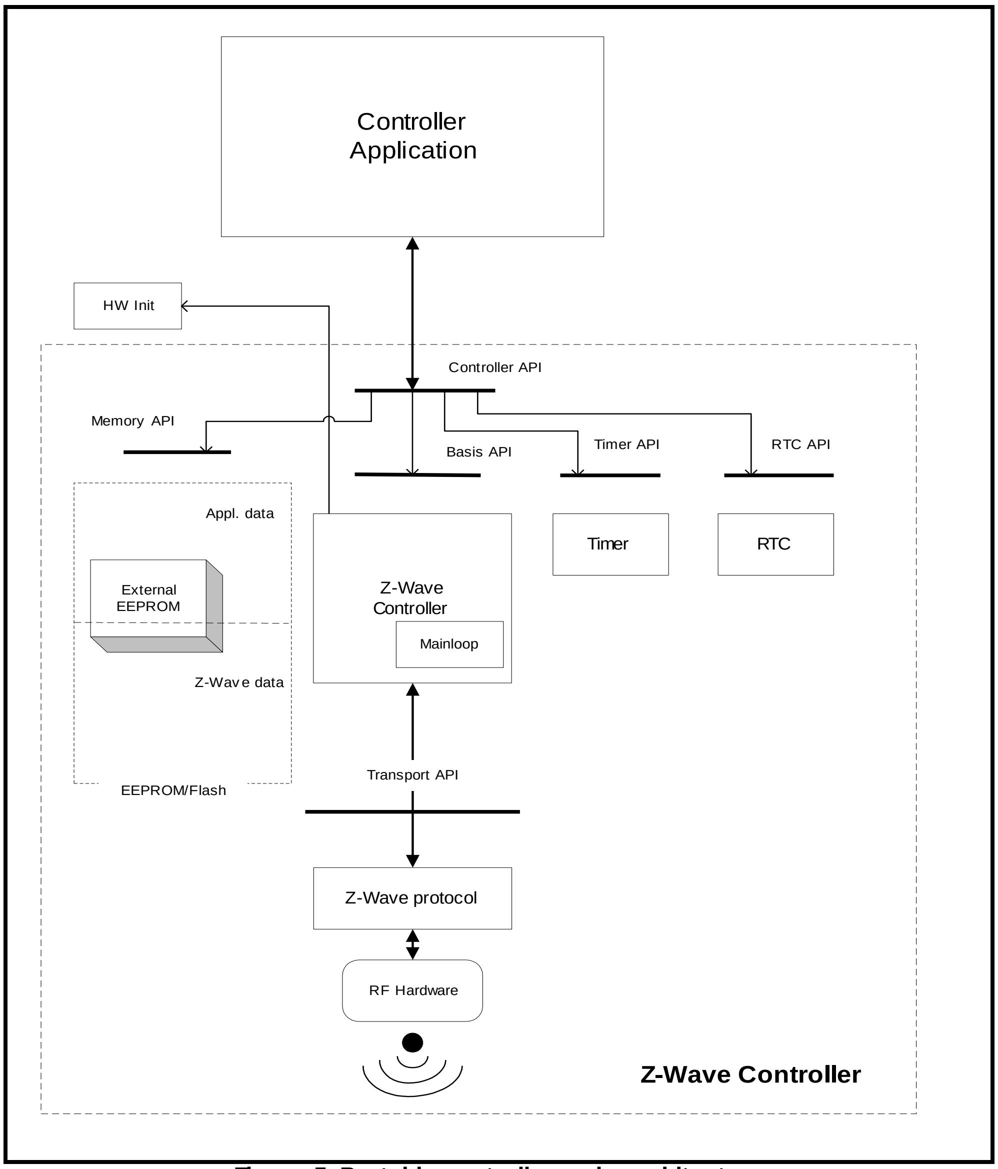

Figure 5. Portable controller node architecture

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.

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 algorithm in a portable controller tries to reach the destination depending on the transmit options as follows:

- If last working route does not exist and TRANSMIT_OPTION_ACK set. Try direct with retries.
- If last working route exist and TRANSMIT_OPTION_ACK set. Try direct without retries. In case it fails, try the last working route. In case the last working route 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. In case TRANSMIT_OPTION_EXPLORE set, a maximum number limits number of tries.
- 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 last working route comprises of 232 destinations having up to one route/direct each, which are stored in non-volatile memory. Last working route can also contain direct attempts. Updating last working route 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.

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
(∗ = 040X for 400 Series Z-Wave 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 non-volatile memory.

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 algorithm in a static controller tries to reach the destination depending on the transmit options as follows:

- If last working route does not exist and TRANSMIT_OPTION_ACK set. Try direct when neighbors.
- If last working route exist and TRANSMIT_OPTION_ACK set. Try the last working route. In case the last working route 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. In case TRANSMIT_OPTION_EXPLORE set, a maximum number limits number of tries.
- 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 last working route comprises of 232 destinations having up to one route/direct each, which are stored in non-volatile memory. Last working route can also contain direct attempts. Updating last working route 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.

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
(∗ = 040X for 400 Series Z-Wave modules, etc).

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 algorithm in an installer controller tries to reach the destination depending on the transmit options as follows:

- If last working route does not exist and TRANSMIT_OPTION_ACK set. Try direct with retries.
- If last working route exist and TRANSMIT_OPTION_ACK set. Try direct without retries. In case it fails, try the last working route. In case the last working route 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. In case TRANSMIT_OPTION_EXPLORE set, a maximum number limits number of tries.
- 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 last working route comprises of 232 destinations having up to one route/direct each, which are stored in non-volatile memory. Last working route can also contain direct attempts. Updating last working route 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 following define must be set when compiling the application: ZW_INSTALLER

The application must be linked with ZW_CONTROLLER_INSTALLER_ZW∗S.LIB
(∗ = 040X for 400 Series Z-Wave 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 algorithm in a bridge controller tries to reach the destination depending on the transmit options as follows:

- If last working route does not exist and TRANSMIT_OPTION_ACK set. Try direct when neighbors.
- If last working route exist and TRANSMIT_OPTION_ACK set. Try the last working route. In case the last working route 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. In case TRANSMIT_OPTION_EXPLORE set, a maximum number limits number of tries.
- 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 last working route comprises of 232 destinations having up to one route/direct each, which are stored in non-volatile memory. Last working route can also contain direct attempts. Updating last working route 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.

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
(∗ = 040X for 400 Series Z-Wave 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.

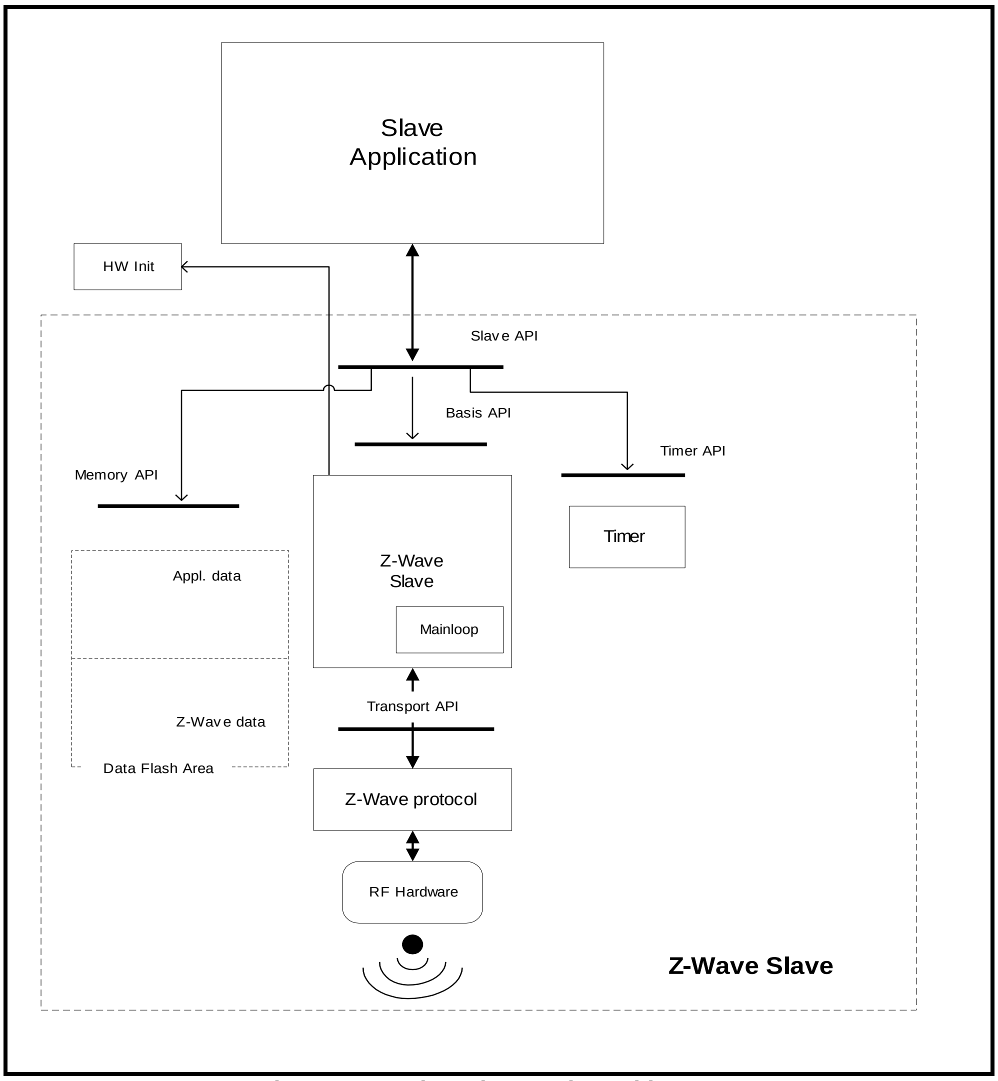

Figure 6. Routing slave node architecture

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 64 bytes MTP (Many Times Programmable memory) for storing data. The Z-Wave basis software reserves the first part of this area, and application data uses the remaning part. The header file "ZW_nvm_addr.h" contains a definition of the physical application memory offset NVM_APPL_OFFSET.

The home ID is set to a randomly generated value and node ID 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.

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 algorithm in a routing slave tries to reach the destination depending on the transmit options 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 one route each. Return routes can also contain direct attempts beside a full route.

The response route comprises of a FIFO containing up to two routes. New routes/direct via the response route or explore mechanism are not inserted into the return route. Updating response routes 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.

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
(∗ = 040X for 400 Series Z-Wave 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.

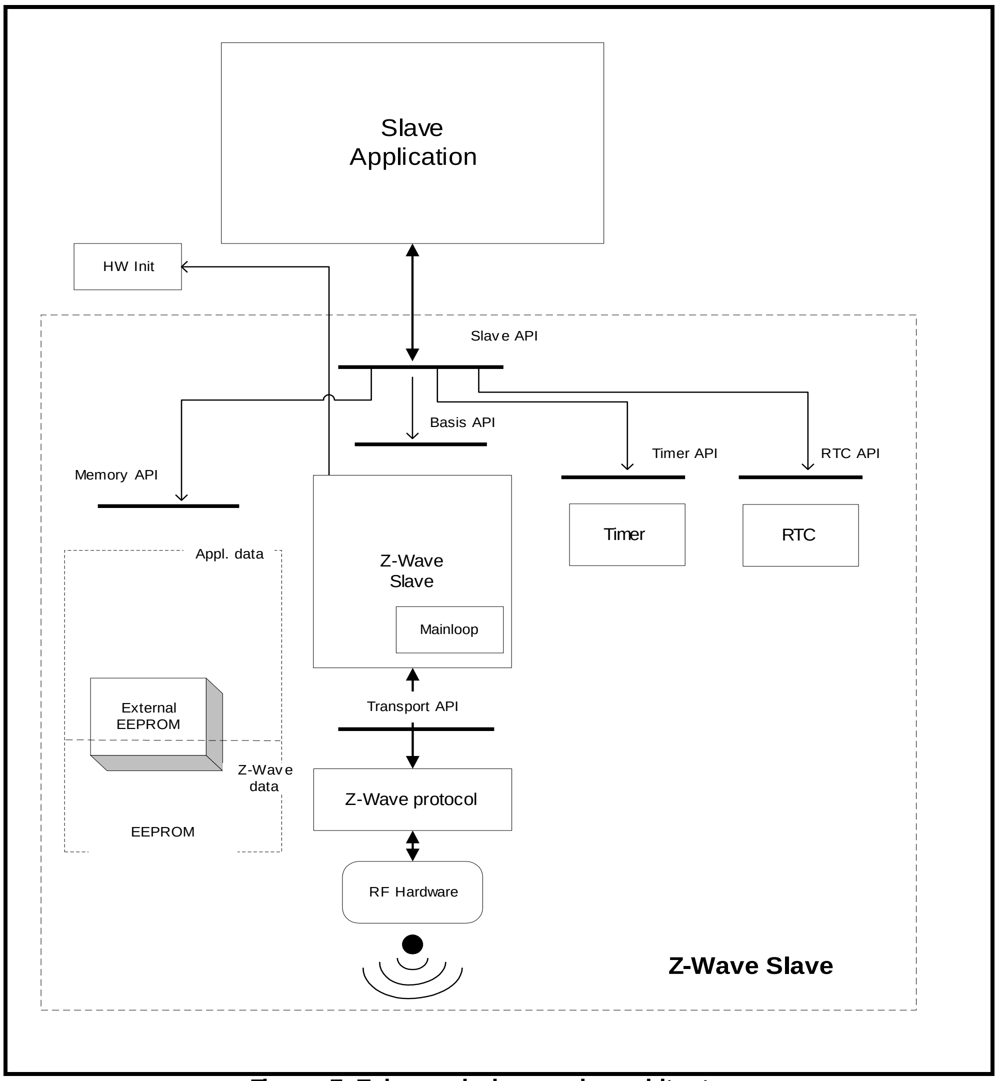

Figure 7. Enhanced slave node architecture

Enhanced slave nodes have an external EEPROM and a WUT. The external EEPROM is used as non-volatile memory instead of MTP. The Z-Wave basis software reserves the first area of the external EEPROM: The last area of the EEPROM is reserved for the application data. The header file "ZW_nvm_addr.h" contains a definition of the physical application memory offset NVM_APPL_OFFSET.

The routing algorithm in an enhanced slave tries to reach the destination depending on the transmit options 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 response route comprises of a FIFO containing up to two routes. New routes/direct are qualified for return route insertion (destination exist and route/direct does not exist). Insert the new route/direct in either an empty return route or the one having lowest priority. Updating response routes 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.

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
(∗ = 040X for 400 Series Z-Wave 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 algorithm in an enhanced 232 slave tries to reach the destination depending on the transmit options 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.

The response route comprises of a The FIFO contains up to one route. New routes/direct are qualified for return route insertion (destination exist and route/direct does not exist). Insert the new route/direct in either an empty return route or the one having lowest priority. Updating response routes 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.

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_232_ZW∗S.LIB
(∗ = 040X for 400 Series Z-Wave 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.

3.9.8.1 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.

3.9.8.2 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.

3.9.8.3 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.

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.

3.9.8.4 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.

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. It could cause secondary controllers and routing slaves to misbehave, if for instance a preferred repeater node is removed. Without automatic network update, 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 must be available in the network. This static controller is 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. In 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.

4 DEVELOPMENT ENVIRONMENT

The 400 Series Z-Wave Single Chip build environment is different compared to previous 100/200/300 Series because the chip contains 64KB OTP instead of 32KB Flash. However, the chip support a development mode enabling application development. Figure below show supported modes.

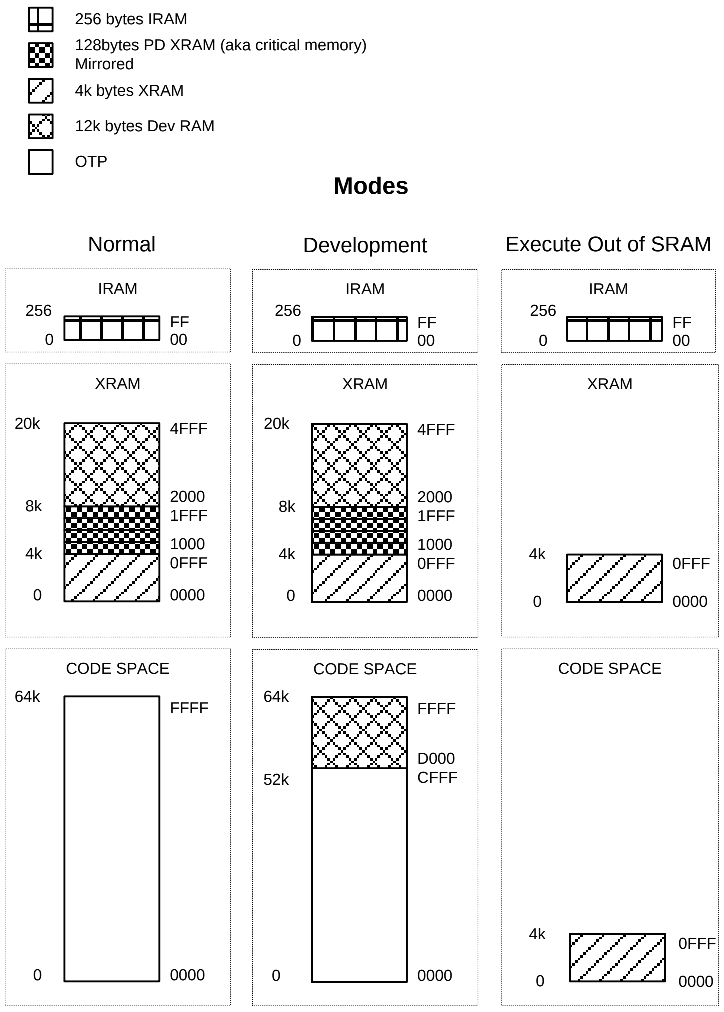

Figure 8. 400 Series Z-Wave Single Chip memory map in the different modes

In normal mode is the entire code space OTP based and this mode is therefore used when producing the final product.

Development mode is used during application development. The 12kB SRAM is mapped into the upper part of the OTP code space to allow modification of the code. For details, refer to [34].

Finally, “Execution Out of SRAM” mode provides a reduced code space of 4kB SRAM allowing execution of small test programs.

5 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.

5.1 API usage guidelines

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

5.1.1 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.

5.1.2 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.

5.2 Z-Wave Libraries

5.2.1 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
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 MTP	X	-	-	-	-	-
Non-volatile RAM in FLASH/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

5.2.1.1 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

5.2.1.2 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[‡‡]	-	X[3]
Request netw ork 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[§§]	X[3]	X[3]	X[***]	X[3]	X
Provide routing table info	-	-	X	X	X	X

[††] Only w hen primary controller and not SUC

[‡‡] Only w hen SUC and not primary controller

[§§] Only if "alw ays listening"

[***] The library w ithout repeater functionality cannot provide help or forward help requests.

5.2.1.3 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 netw ork 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.

5.2.1.4 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 4kB of additional flash and 60 bytes of RAM.

[1] Only w hen primary/inclusion controller

[2] Only w hen primary controller

[3] Only if "alw ays listening"

[4] The library w ithout repeater functionality cannot provide help or forward help requests.

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.

5.3 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].

Various header files	**Description**
ZW_adcdriv.h	ADC functions
ZW_appltimer_api.h	GPTimer/PWM and the 8051 timers functions and macros
ZW_mtp_api.h	MTP functions
ZW_nvm_addr.h	Application start address in non-volatile memory
ZW_pindefs.h	Macros for manipulating the GPIO's
ZW_rf0402_api.h	RF functions
ZW_SerialAPI.h	Serial API interface with function ID defines etc.
ZW_spi_api.h	SPI functions
ZW_sysdefs.h	CPU and clock defines.
ZW_timer_api.h	Software timer functions
ZW_triac_api.h	TRIAC controller functionality.
ZW_typedefs.h	Common used defines (BYTE, WORD…).
ZW_uart_api.h	UART functions
ZW040x.h	Inventra m8051w SFR and ISR C defines for the Z-Wave ZW040x RF transceiver.

5.4 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 does 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).

A serial API implementation provide an interface to the major part of interface functions via a serial port. The SDK contains a serial API application [34], which enables a host processor to control the interface functions via a serial port.

5.4.1 Required Application Functions

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

Warning: In order not to disrupt the radio communication and the protocol, no application function must execute code for more than 5ms without returning. It is not allowed to disable interrupt more than it takes to received 8 bits, which is around 0.8ms at 9.6kbps.

5.4.1.1 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 have to 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
	ZW_WAKEUP_WATCHDOG	Reset because of a watchdog timeout
	ZW_WAKEUP_EXT_INT	Woken up by external interrupt
	ZW_WAKEUP_POR	Reset by Power on reset circuit

Serial API (Not supported)

5.4.1.2 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. Notice that watchdog is enabled by default and must be kicked to avoid resetting the system (See ZW_WatchDogKick).

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)

5.4.1.3 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 equal to 0x00000000 and node ID as follows.

Device	**Node ID**
Slave	0x00
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)

5.4.1.4 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. If the application software executes CPU time consuming functions, without returning to the Z-Wave main loop, the **ZW_POLL** function must be called frequently (see **ZW_POLL**).

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	400 Series LED Dimmer
Minimum	58 us	7.2 us	7.2 us	TBD
Maximum	3.8 ms	2.4 ms	2.4 ms	TBD

Defined in: ZW_basis_api.h

Serial API (Not supported)

5.4.1.5 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, multicast, or broadcast frame). This is used to avoid flooding the network by responding on a multicast or broadcast.

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

5.4.1.6 ApplicationNodeInformation

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

The Z-Wave System 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 available on Routing Slaves. This option is not available on 3-channel systems (the JP frequency).
	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 available 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.
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)

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 9. 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.

5.4.1.7 ApplicationSlaveUpdate (All slave libraries)

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

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

All slave libraries requires this function implemented within the System 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[ ]

5.4.1.8 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 System 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 System 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 added to the network
	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.	

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.

5.4.1.9 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 System 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.	

Serial API:

ZW->HOST: REQ | 0xA8 | rxStatus | destNodeID | srcNodeID | cmdLength | pCmd[ ] | multiDestsOffset_NodeMaskLen | multiDestsNodeMask

5.4.1.10 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 System 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[ ]

5.4.1.11 ApplicationRfNotify

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].

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

5.4.2 Z-Wave Basis API

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

5.4.2.1 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

5.4.2.2 ZW_GetProtocolStatus

BYTE ZW_GetProtocolStatus(void)

Macro: ZW_GET_PROTOCOL_STATUS()

Report the status of the protocol.

This function returns 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

5.4.2.3 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 built-in 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.

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

5.4.2.4 ZW_Poll

void ZW_Poll(void)

Macro: ZW_POLL

This Z-Wave low-level poll function handles the transfer of bytes from the transmit buffer to the RF media and buffering of incoming frames in the receive buffer.

This function must be called while doing a busy loop or other time consuming execution in the application code to avoid loosing incoming frames and corrupting outgoing frames. This function does not service the Z-Wave protocol so no frames will be acknowledged or forwarded when the application is busy and calling ZW_Poll() so other nodes in the network will experience that the node is very difficult to communicate with. The use of this call should be limited to situations where it is impossible for the application to return from the function that it is currently running in or situations where radio communication is not necessary.

Defined in: ZW_basis_api.h

Serial API (Not supported)

5.4.2.5 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)

Serial API

HOST->ZW: REQ | 0x1D

ZW->HOST: RES | 0x1D | rndNo

5.4.2.6 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 normal Power 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

[†††] 400 Series support only -2dB power level steps

5.4.2.7 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

5.4.2.8 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 IN	Transmit complete call back.

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.

Serial API:

HOST->ZW: REQ | 0x53 | funcID

Notice: funcID is used to correlate callback with original request. Callback is disabled by setting funcID equal to zero in original request.

ZW->HOST: RES | 0x53 | retVal

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

5.4.2.9 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 neighbor 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

[‡‡‡] 400 Series support only -2dB power level steps

5.4.2.10 ZW_RFAbove3vSupplyGuaranteed

void ZW_RFAbove3vSupplyGuaranteed(BOOL above_3v_supply)

Use this function to disable\enable the OTP charge pump in RX mode. Switching the charge pump off in RX lowers the sensitivity variation. However the charge pump can only be switched off if the supply is guaranteed to be above 3V.

Defined in: ZW_basis_api.h

Parameters:

Above_3v_supply IN	Boolean	
	TRUE	The power supply can guarantee a voltage above 3v, thus disabling the OTP charge pump.
	FALSE	Power supply can't guarantee voltage above 3v, thus enabling the OTP charge pump.

Serial API:

Not supported yet

5.4.2.11 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.

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 might fail.

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 IN	Transmit completed call back function

Callback function Parameters:

txStatus IN	(see **ZW_SendData**)

Serial API:

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

ZW->HOST: RES | 0x12 | retVal

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

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:

txStatus IN (see **ZW_SendData**)

Return value:

BYTE	FALSE	If transmit queue overflow.

Serial API

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

ZW->HOST: REQ | 0xBE | retVal

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

[§§§] 400 Series support only -2dB pow er level steps

5.4.2.12 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.

5.4.2.13 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 P1.0)
	ZW_INT1	External interrupt 1 (Pin P1.1)
triggerLevel IN	The external interrupt trigger level:	
	TRUE	Set the interrupt trigger to high level /Rising edge
	FALSE	Set the interrupt trigger to low level /Falling edge

Serial API

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

5.4.2.14 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 System layer frames will be passed to the System 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 System layer.

Promiscuously received frames are delivered to the application via the ApplicationCommandHandler callback function (see section 5.4.1.5).

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 5.4.1.5 for callback syntax when a frame has been promiscuously received.

5.4.2.15 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

5.4.2.16 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

5.4.2.17 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

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
ZM0401	0x04	0x07
ZM4102	0x04	0x07
SD3402	0x04	0x07
ZW0403	0x05	0x07

Timer functions are: TimerStart, TimerRestart and TimerCancel.

5.4.2.18 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)

5.4.2.19 ZW_WatchDogEnable

void ZW_WatchDogEnable(void)

Macro: ZW_WATCHDOG_ENABLE()

Enables the 400 Series Z-Wave Single Chip built-in watchdog. By default, the watchdog is disabled. The watchdog timeout interval is 1 second.

The watchdog must be kicked at least one time per interval. Failing to do so will cause the 400 Series Z-Wave Single Chip to be reset.

Some software bugs can be difficult to diagnose when the watchdog is enabled because the application will reboot when the watchdog resets the 400 Series Z-Wave Single Chip. 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

5.4.2.20 ZW_WatchDogDisable

void ZW_WatchDogDisable(void)

Macro: ZW_WATCHDOG_DISABLE ()

Disable the 400 Series Z-Wave Single Chip built in watchdog.

Defined in: ZW_basis_api.h

Serial API

HOST->ZW: REQ | 0xB7

5.4.2.21 ZW_WatchDogKick

void ZW_WatchDogKick(void)

Macro: ZW_WATCHDOG_KICK ()

To keep the watchdog timer from resetting the 400 Series Z-Wave Single Chip, it has to be kicked 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

5.4.3 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.

5.4.3.1 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)

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.

When communicating to a Frequently Listening Routing Slave (FLiRS) will the API call automatically generate a wakeup beam to awake the FLiRS. ZW0102 does not support generation and detection of wakeup beams.

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.

Table 9. Transmit options behavior for portable and installer libraries

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

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 ZDK 4.5 controller uses the routing algorithm from 5.02 to address nodes from ZDK's not supporting explorer frame. The routing algorithm from 5.02 ignores the transmit option TRANSMIT_OPTION_EXPLORE flag and maximum number of source routing attempts value. 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.

In a bridge controller library, sending to a virtual node belonging to the bridge itself is not recommended.

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

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 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. In case it is a routed singlecast the source routing info takes up to 6 bytes depending on the number of hops leaving minimum 48 bytes for the payload. In case it is a singlecast, which piggyback on an explorer frame overhead is 8 bytes leaving minimum 46 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_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 a 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).

Serial API:

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

ZW->HOST: RES | 0x13 | RetVal

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

5.4.3.2 ZW_SendData_Bridge

```
BYTE ZW_SendData_Bridge( BYTE srcNodeID,
                         BYTE destNodeID,
                         BYTE *pData,
                         BYTE dataLength,
                         BYTE txOptions,
                         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, 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 ZDK 4.5 controller uses the routing algorithm from 5.02 to address nodes from ZDK's not supporting explorer frame. The routing algorithm from 5.02 ignores the transmit option TRANSMIT_OPTION_EXPLORE flag and maximum number of source routing attempts value. 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: Always 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.

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 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. In case it is a routed single cast the source routing info takes up to 6 bytes depending on the number of hops leaving minimum 48 bytes for the payload. In case it is a singlecast, which piggyback on an explorer frame overhead is 8 bytes leaving minimum 46 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_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).
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).
	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.

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

WARNING: Use pRoute[4] equal [0,0,0,0].

5.4.3.3 ZW_SendDataMeta_Bridge

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

Macro: ZW_SEND_DATA_META_BRIDGE(srcnodeid, nodeid, data, length, options, 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 does 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 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

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 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. In case it is a routed single cast the source routing info takes up to 6 bytes depending on the number of hops leaving minimum 48 bytes for the payload. In case it is a singlecast, which piggyback on an explorer frame overhead is 8 bytes leaving minimum 46 bytes for the payload. 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)
completedFunc	Transmit completed call back function	

Callback function Parameters:

txStatus IN	(see **ZW_SendData**)

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

WARNING: Use pRoute[4] equal [0,0,0,0].

5.4.3.4 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

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**)

Serial API:

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

ZW->HOST: RES | 0x14 | RetVal

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

5.4.3.5 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

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 for a multicast depends on the destination node IDs leaving between 25-53 bytes for the payload. The size of the protocol header and checksum for a multicast frame is: ((MaxNodeID - ((MinNodeID - 1) & 0xE0)+7) >> 3) + 10 where MaxNodeID is the largest node ID number and MinNodeID is the smallest.
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**)

Serial API:

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

ZW->HOST: RES | 0xAB | RetVal

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

5.4.3.6 ZW_SendDataAbort

void ZW_SendDataAbort(void)

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

5.4.3.7 ZW_SendConst

void ZW_SendConst(BYTE bStart, BYTE bChNo, BYTE bSignalType)

This function start/stop generating RF test signal.
The test signal can be on of the following:

- Test signal with only the carrier frequency.
- Test signal with a modulated carrier frequency; the signal will switch between sending logical 1 frequency and logical zero frequency

The function also selects which channel to send the test signal on.

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

The API should only be called when starting\stopping a test.

Parameters:

bStart	Start/Stop generating RF test signal	TRUE start sending RF test signal. FALSE stop sending RF test signal
bChNot	IN The number of channel to send the test signal on.	0..1 for 2 channels targets 0..2 for 3 channels targets
bSignalType	IN type of the RF test signal to generater	ZW_RF_TEST_SIGNAL_CARRIER ZW_RF_TEST_SIGNAL_CARRIER_MODULATED

Defined in: ZW_transport_api.h

Serial API (Not supported)

5.4.3.8 ZW_SetListenBeforeTalkThreshold

void ZW_SetListenBeforeTalkThreshold(BYTE bChannel, BYTE bThreshold)

This function sets the "Listen Before Talk" threshold used in the Japanese frequency band. The default threshold value is set to "49(dec)" and corresponds to a chip input power of -75dBm. The appropriate value range goes from 34(dec) to 78(dec) and each threshold step corresponds to a 1.5dB input power step.

For instance, if a SAW filter with an insertion loss of 3dB is inserted between the antenna feed-point and the chip, the threshold value should be set to 47(dec).

Parameters:

bChannel	IN	Channel number the Threshold should be set for. Valid channel numbers are 0,1 and 2
bThreshold	IN	The threshold the RSSI should use. Valid threshold range is from 34 to 78.

Defined in: ZW_transport_api.h

Serial API (Not supported)

NOTE: This function is avalible in Japan.

5.4.4 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.

5.4.4.1 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)

5.4.4.2 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)

5.4.4.3 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)

5.4.4.4 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)

5.4.4.5 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)

5.4.5 IO API

The 400 Series Z-Wave Single Chip has four ports: P0, P1, P2, and P3. All IO's can be set as either input or output. The IO cells are push/pull cells. When an IO is set as input, a pull-up can be enabled optionally on the input pin of that IO.

Port	ZM4101
P0	P0.0-P0.7
P1	P1.0-P1.7
P2	P2.0
P3	P3.0,P3.1,P3.4-P3.5

The IO's can be used either as a general purpose IO (GPIO) or for some of the IO's, it can be used by one or more of the built-in HW peripherals. The IO's are default set as GPIO's. This means that they are directly controlled by the MCU. If a built-in HW peripheral is enabled it can take over control of the IO, this means the direction of the IO, the pull-up state or the output state. In the case where several HW peripherals that it takes control over can use a particular IO, the control is prioritized as depicted in Table 10.

Table 10. IO functions (Some of the functions are not yet available)

IO	**Functions** (Listed with lowest priority first)
P0.4	GPIO, Key scanner Column 4 output, LED0 output
P0.5	GPIO, Key scanner Column 5 output, LED1 output
P0.6	GPIO, Key scanner Column 6 output, LED2 output
P0.7	GPIO, Key scanner Column 7 output, LED3 output
P1.0	GPIO, External Interrupt 0, Key scanner Row 0 input
P1.1	GPIO, External Interrupt 1, Key scanner Row 1 input
P1.2	GPIO, Key scanner Row 2 input
P1.3	GPIO, Key scanner Row 3 input
P1.4	GPIO, Key scanner Row 4 input
P1.5	GPIO, Key scanner Row 5 input
P1.6	GPIO, Key scanner Row 6 input
P1.7	GPIO, Key scanner Row 7 input
P2.0	GPIO, Key scanner Column 15 output, UART0 Rx input
P2.1	GPIO, Key scanner Column 14 output, UART0 Tx output
P2.2	GPIO, SPI1 master output
P2.3	GPIO, SPI1 master Input
P2.4	GPIO, SPI1 serial clock output
P2.5	GPIO
P2.6	GPIO
P3.1	GPIO, Key scanner Column 12 output, IR Rx input
P3.4	GPIO, ADC0 input, Key scanner Column 11 output, IR Tx0 output
P3.5	GPIO, ADC1 input, Key scanner Column 10 output, IR Tx1 output
P3.6	GPIO, ADC2 input, Key scanner Column 9 output, IR Tx2 output, Triac output
P3.7	GPIO, ADC3 input, Key scanner Column 8 output, Triac Zero-cross input, PWM output

The state of the IO's must be fixed before the 400 Series Z-Wave Single Chip is put into powerdown mode and must be enabled after the 400 Series Z-Wave Single Chip is powered-up. This is done to avoid unwanted glitches on the IO's when the 400 Series Z-Wave Single Chip is powered up.

5.4.5.1 ZW_IOS_set

```
void ZW_IOS_set( BYTE bPort,
                 BYTE bDirection,
                 BYTE bValue)
```

This function is used to set the state of the GPIO’s In **ApplicationInitHW()**.

Defined in: ZW_basis_api.h

Parameters:

bPort IN	0-3	Port number 0 => P0, 1 => P1, 2 => P2, 3 => P3
bDirection IN	bit pattern	Direction. 0b=output, 1b=input. E.g. 0xF0=> upper 4 IO's are inputs and the lower 4 IO's are outputs
bValue IN	bit pattern	Output setting / Pull-up state When an IO is set as output the corresponding bit in bValue will determine the output setting: 1b=high 0b=low When an IO is set as input the corresponding bit in bValue will determine the state of the pull-up resistor in the IO cell: 1b=pull-up disabled 0b=pull-up enabled

Serial API (Not supported)

5.4.6 GPIO macros

In order to be able to control the GPIO individually a set of helper macros has been defined. These macros can set a GPIO as input/output, set the state of the output GPIO or read the value of an input GPIO.

The GPIO name will be a parameter in all the macros. The format of the pin name is as follow:

P(port number)(IO number)

thus IO pin 3 in port 1 name will be P13.

Note: The actual change of the IO settings first takes place after leaving **ApplicationInitHW()**.

5.4.6.1 PIN_OUT

PIN_OUT(pin)

This macro sets a GPIO as an output IO. Notice that this macro first works after **ApplicationInitHW()** has been executed.

Defined in: ZW_pindefs.h

Parameters:

pin IN	Pxy	Name of a GPIOr x = port number; y = IO number

Example:

```
PIN_OUT(P12);
```

5.4.6.2 PIN_IN

PIN_IN(pin, pullup)

This macro sets a GPIO as an input and determines whether the internal pullup is enabled or disabled. Notice that this macro first works after **ApplicationInitHW()** has been executed..

Defined in: ZW_pindefs.h

Parameters:

pin IN	Pxy	Name of a GPIO x = port number; y = IO number
pullup IN	Boolean	Pull-up state. 0b=disabled, 1b=enabled.

Example:

```
PIN_IN(P30,TRUE);
```

5.4.6.3 PIN_LOW

PIN_LOW(pin)

This macro sets the state of an output GPIO to low. Notice that this macro first works after **ApplicationInitHW()** has been executed.

Defined in: ZW_pindefs.h

Parameters:

pin IN	Pxy	Name of a GPIOr x = port number; y = IO number

Example:

```
PIN_LOW(P12);
```

5.4.6.4 PIN_HIGH

PIN_HIGH(pin)

This macro sets the state of an output GPIO to HIGH. Notice that this macro first works after **ApplicationInitHW()** has been executed.

Defined in: ZW_pindefs.h

Parameters:

pin IN	Pxy	Name of a GPIOr x = port number; y = IO number

Example:

```
PIN_HIGH(P12);
```

5.4.6.5 PIN_TOGGLE

PIN_TOGGLE(pin)

This macro toggle the state of an output GPIO from high to low or low to high. Notice that this macro first works after **ApplicationInitHW()** has been executed.

Defined in: ZW_pindefs.h

Parameters:

pin IN	Pxy	Name of a GPIOr x = port number; y = IO number

Example:

```
PIN_TOGGLE(P12);
```

5.4.6.6 PIN_GET

PIN_GET(pin)

This macro gets the state of the pin of a GPIO.

Defined in: ZW_pindefs.h

Parameters:

pin IN	Pxy	Name of a GPIOr x = port number; y = IO number

Return value

BOOL	TRUE FALSE	The pin is high The pin is low

Example:

```
a=PIN_GET(P12);
```

5.4.7 Z-Wave Memory API

The memory application interface handles accesses to the application data area in non-volatile memory.

Routing slave nodes use MTP for storing application data. Enhanced slave and all controller nodes use an external non-volatile memory for storing application data. The Z-Wave protocol uses the first part of the external non-volatile memory for home ID, node ID, routing table etc. The external non-volatile memory is accessed via the SPI1 interface and using P2.5 as chip select. Alternative chip select pins, refer to [34].

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.

5.4.7.1 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 non-volatile memory to the specified RAM addresses.

NOTE: A NULL pointer can be given as the pHomeID parameter if the application is only interested 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

5.4.7.2 MemoryGetByte

BYTE MemoryGetByte(WORD offset)

Macro: ZW_MEM_GET_BYTE(offset)

Read one byte from the non-volatile memory allocated for the application.

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

5.4.7.3 MemoryPutByte

BYTE MemoryPutByte(WORD offset, BYTE data)

Macro: ZW_MEM_PUT_BYTE(offset,data)

Write one byte to the application area of the non-volatile memory.

On controllers and enhanced slaves this function is based on external non-volatile memory and a long write time (2-5 msec.) must be taken into consideration when implementing the application.

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 it is more than 200ms ago the buffer was accessed.

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

5.4.7.4 MemoryGetBuffer

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

Macro: ZW_MEM_GET_BUFFER(offset,buffer,length)

Read a number of bytes from the non-volatile memory allocated for the application.

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[ ]

5.4.7.5 MemoryPutBuffer

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

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

Copy a number of bytes from a RAM buffer to the application area of the non-volatile memory.

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 it is more than 200ms ago 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.

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	

Serial API:

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

ZW->HOST: RES | 0x24 | RetVal

ZW->HOST: REQ | 0x24 | funcID

5.4.7.6 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)

5.4.7.7 ZW_MemoryFlush

void ZW_MemoryFlush(void)

Macro: ZW_MEM_FLUSH()

This call writes data immediately to the application area of the non-volatile memory.

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 it is more than 200ms ago 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 non-volatile memory . Data is written directly to the EEPROM.

Defined in: ZW_mem_api.h

Serial API (Not supported)

5.4.8 Z-Wave Timer API

The software timer is based on a “tick-function” every 10 ms. 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 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 continue from their current state.

Global tick counter is stored in the global variable:

```
WORD tickTime
```

5.4.8.1 TimerStart

```
BYTE TimerStart( VOID_CALLBACKFUNC(func)(),
                 BYTE bTimerTicks,
                 BYTE bRepeats)
```

Macro: ZW_TIMER_START(func, bTimerTicks, bRepeats)

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 "bRepeats" times before the timer is stopped. It’s possible to have up to 5 timers running simultaneously.

Defined in: ZW_timer_api.h

Return value:

BYTE	Timer handle (timer table index). 0xFF is returned if the timer start operation failed. The timer handle is used when calling other timer functions such as TimerRestart, etc.

Parameters:

pFunc IN	Timeout function address (not NULL).
bTimerTicks IN	Timeout value (value * 10 ms). Predefined values: TIMER_ONE_SECOND
bRepeats IN	Number of function calls. Maximum value is 253. Predefined values: TIMER_ONE_TIME

TIMER_FOREVER

Serial API (Not supported)

5.4.8.2 TimerRestart

BYTE TimerRestart(BYTE bTimerHandle)

Macro: ZW_TIMER_RESTART(BYTE bTimerHandle)

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 been canceled.

Defined in: ZW_timer_api.h

Return value:

BYTE	TRUE	Timer restarted

Parameters:

bTimerHandle	IN	Timer to restart

Serial API (Not supported)

5.4.8.3 TimerCancel

BYTE TimerCancel(BYTE bTimerHandle)

Macro: ZW_TIMER_CANCEL(bTimerHandle)

Stop and unregister 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	Timer cancelled

Parameters:

bTimerHandle IN	Timer number to stop	

Serial API (Not supported)

5.4.9 Power Control API

The 400 Series Z-Wave Single Chip has two types of power down mode: WUT (Wake Up Timer) mode and Stopped mode. Stopped mode is the lowest power mode of the chip where all circuitry is shut down except for a small basic block that keeps the IO states and allows for wakeup by external interrupt. It is also possible to have a small part of the RAM, denoted Critical Memory, powered. WUT mode is identical to Stopped mode except for enabling of a low power ring oscillator that ticks every second or 1/128 second. The WUT can wake up the chip after a programmable amount of time.

The sample applications are executed out of Development RAM. In this mode the chip will actually not shut down as it would erase the RAM that stores the program. Instead, it will halt the CPU and disable the watchdog. The chip is awoken as if it was reset.

5.4.9.1 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. 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 chip.

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

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.

Parameters:

mode IN Specify the type of power save mode:

	ZW_STOP_MODE	The whole chip is turned down. The chip can be wakened up again by Hardware reset or by the external interrupt INT1.
	ZW_WUT_MODE	The chip 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**. The chip 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 s. The maximum timeout value is 2 s. 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 1000 ms or 250 ms and check for radio transmissions to the module (See 5.4.1.6 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.

intEnable IN	Interrupt enable bit mask. If a bit mask is 1, the corresponding interrupt is enabled and this interrupt will wakeup the chip from power down. Valid bit masks are:	
	ZW_INT_MASK_EXT1	External interrupt 1 (PIN P1_1) is enabled as interrupt source
	0x00	No external Interrupts will wakeup. Useful 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

5.4.10 SPI interface API

The 400 Series Z-Wave Single Chip has a SPI controller that operates as a SPI master, SPI1, and in some versions it also has a SPI controller that can operate as a SPI master or as a SPI slave, SPI0.

The SPI master,SPI1, is reserved by the Z-Wave protocol, if the 400 Series Z-Wave Single Chip is programmed as one of the following Z-Wave nodes types: : Portable Controller, Static Controller, Installer Controller, Bridge Controller, or Enhanced Slave.

5.4.10.1 ZW_SPI1_init

void ZW_SPI1_init(BYTE bSpiInit)

Initializes the 400 Series Z-Wave Single Chip built-in SPI master controller, SPI1. The function sets the SPI clock speed, the signaling mode and the data order. E.g.:

```
ZW_SPI1_init(SPI_SPEED_8_MHZ|SPI_SIG_MODE_1|SPI_MSB_FIRST)
```

Sets clock speed to 8MHz, SPI clock idle to low, data sampled at rising edge and clocked at falling edge, and sends most significant bit first.

Defined in: ZW_spi_api.h

Parameters:

bSpiInit IN	bit mask:	
	Speed of the SPI clock	
	SPI_SPEED_8_MHZ	SPI clock runs at @8MHz
	SPI_SPEED_4_MHZ	SPI clock runs at @4MHz
	SPI_SPEED_2_MHZ	SPI clock runs at @2MHz
	SPI_SPEED_1_MHZ	SPI clock runs at @1MHz
	SPI signaling modes	
	SPI_SIG_MODE_1	SPI clock idle low, data sampled at rising edge and clocked at falling edge
	SPI_SIG_MODE_2	SPI clock idle low, data sampled at falling edge and clocked at rising edge
	SPI_SIG_MODE_3	SPI clock idle high, data sampled at falling edge and clocked at rising edge
	SPI_SIG_MODE_4	SPI clock idle high, data sampled at rising edge and clocked at falling edge
	Data order	
	SPI_MSB_FIRST	send MSB bit first
	SPI_LSB_FIRST	send LSB bit first

Serial API (Not supported)

5.4.10.2 ZW_SPI1_rx_get

BYTE ZW_SPI1_rx_get(void)

This function returns a previously received byte from SPI1.

This function does not wait until data has been received.

Defined in:	ZW_spi_api.h

Return value:

BYTE	Received data.

Serial API (Not supported)

5.4.10.3 ZW_SPI1_active_get

BYTE ZW_SPI1_active_get(void)

Read the SPI1 send data status.

Defined in: ZW_spi_api.h

Return value:

BYTE	non-zero	SPI1 Transmitter is busy
	zero (0x00)	SPI1 Transmitter is idle

Serial API (Not supported)

5.4.10.4 ZW_SPI1_coll_get

BYTE ZW_SPI1_coll_get(void)

This function returns the state of the SPI1 collision flag and then clears the collision flag.

Defined in: ZW_spi_api.h

Return value:

BYTE	non-zero	SPI1 data collided
	zero (0x00)	SPI1 no collisions

Serial API (Not supported)

5.4.10.5 ZW_SPI1_tx_set

void ZW_SPI1_tx_set(BYTE data)

Function starts transmission over the SPI1. The received data can be read with **ZW_SPI_active_get()** when the SPI1 controller is done.

This function waits until SPI1 transmitter is idle before it sends the new data. The function does not wait until the new data has been sent. See also **ZW_SPI1_tx_data_set().**

Defined in: ZW_spi_api.h

Parameters:

data IN	Data to be send.

Serial API (Not supported)

5.4.10.6 ZW_SPI1_enable

void ZW_SPI1_enable(BYTE bState)

Function enables the SPI1 master and allocates the pins MISO1, MOSI1, and SCK1.

Defined in: ZW_spi_api.h

Parameters:

bState IN	TRUE	enable the SPI1 controller
	FALSE	disable the SPI1 controller

Serial API (Not supported)

5.4.10.7 ZW_SPI0_init

void ZW_SPI0_init(BYTE bSpiInit)

Initializes the 400 Series Z-Wave Single Chip built-in SPI0 master/slave controller. Notice that not all 400 Series Z-Wave Single Chip/modules has this SPI available on the pin-out.

This function sets the SPI clock speed, the signaling mode and the data order. E.g.:

```
ZW_SPI1_init(SPI_SPEED_8_MHZ|SPI_SIG_MODE_1|SPI_MSB_FIRST)
```

Sets clock speed to 8MHz, SPI clock idle to low, data sampled at rising edge and clocked at falling edge, and sends most significant bit first.

Defined in: ZW_spi_api.h

Parameters:

bSpiInit IN bit mask:

Speed of the SPI clock (master mode only)

SPI_SPEED_8_MHZ	SPI clock runs at @8MHz
SPI_SPEED_4_MHZ	SPI clock runs at @4MHz
SPI_SPEED_2_MHZ	SPI clock runs at @2MHz
SPI_SPEED_1_MHZ	SPI clock runs at @1MHz

SPI signaling modes

SPI_SIG_MODE_1	SPI clock idle low, data sampled at rising edge and clocked at falling edge
SPI_SIG_MODE_2	SPI clock idle low, data sampled at falling edge and clocked at rising edge
SPI_SIG_MODE_3	SPI clock idle high, data sampled at falling edge and clocked at rising edge
SPI_SIG_MODE_4	SPI clock idle high, data sampled at rising edge and clocked at falling edge

Data order

SPI_MSB_FIRST	send MSB bit first
SPI_LSB_FIRST	send LSB bit first

Master/Slave

SPI_MASTER	enable SPI master mode
SPI_SLAVE	enable SPI slave mdoe

Slave Select (Slave mode only)

SPI_SS_N_SS	use io SS_N IO as the slave select when the 400 Series Z-Wave Single Chip is in SPI slave mode
SPI_SS_N_GPIO	slave controller is always enabled when the 400 Series Z-Wave Single Chip is in SPI slave mode. The IO, SS_N, can be used as a GPIO.

Serial API (Not supported)

5.4.10.8 ZW_SPI0_rx_get

BYTE ZW_SPI0_rx_get(void)

Function returns a previously received byte from SPI0.

This function does not wait until data has been received.

Defined in: ZW_spi_api.h

Return value:

BYTE	Received data.

Serial API (Not supported)

5.4.10.9 ZW_SPI0_active_get

BYTE ZW_SPI0_active_get(void)

Read the SPI0 send data status.

Defined in: ZW_spi_api.h

Return value:

BYTE	non-zero	SPI0 Transmitter is busy
	zero (0x00)	SPI0 Transmitter is idle

Serial API (Not supported)

5.4.10.10 ZW_SPI0_coll_get

BYTE ZW_SPI0_coll_get(void)

This function returns the state of the SPI0 collision flag and then clears the collision flag.

Defined in: ZW_spi_api.h

Return value:

BYTE	non-zero	SPI0 data collided
	zero (0x00)	SPI0 no collisions

Serial API (Not supported)

5.4.10.11 ZW_SPI0_int_get

BYTE ZW_SPI0_int_get(void)

This function returns the state of the SPI0 interrupt/transmission done flag.

Defined in: ZW_spi_api.h

Return value:

BYTE	non-zero	SPI0 interrupt/transmission flag is set
	zero (0x00)	SPI0 interrupt/transmission flag is cleared

Serial API (Not supported)

5.4.10.12 ZW_SPI0_tx_set

void ZW_SPI0_tx_set(BYTE data)

Function starts transmission over the SPI0. The received data can be read with **ZW_SPI0_active_get()** when the SPI0 controller is done.

This function waits until SPI0 transmitter is idle before it sends the new data. The function does not wait until the new data has been sent. See also **ZW_SPI0_tx_data_set().**

Defined in:	ZW_spi_api.h

Parameters:

data IN	Data to be send.

Serial API (Not supported)

5.4.10.13 ZW_SPI0_enable

void ZW_SPI0_enable(BYTE bState)

Function enables the SPI0 master and allocates the pins MISO0, MOSI0, and SCK0. If SPI_SS_N_SS is set in ZW_SPI0_init() then also SS_N0 is allocated.

Defined in: ZW_spi_api.h

Parameters:

bState IN	TRUE	enable the SPI0 controller
	FALSE	disable the SPI0 controller

Serial API (Not supported)

5.4.10.14 ZW_SPI0_int_clear

void ZW_SPI0_int_clear(void)

Function clears the SPI0 interrupt/transmission done flag

Defined in: ZW_spi_api.h

Serial API (Not supported)

5.4.11 ADC interface API

The ADC is an 8/12-bit ADC with a 4 input multiplexer. The ADC can be used for monitoring battery-levels, voltages across various sensors etc. The ADC releases an interrupt if the measured voltage is above, below or equal to a threshold depending on the configuration settings. The ADC uses up to 4 GPIO as inputs depending on its configuration. Input pins that are not enabled can be used as GPIO

Three sources can work as voltage-references for the ADC, namely either the power-supply for the chip, an internal 1.2V voltage-reference or the P3.7 pin. The sample rate when in continuous conversion mode is 21k sample/s for 8 bit conversions and 9k sample/s for 12 bit conversions.

The figures below show when the ADC interrupt is released dependent on, how the ADC threshold gradient is set:

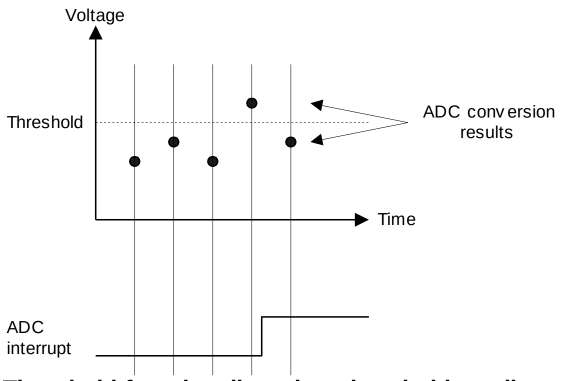

Figure 10. Threshold functionality when threshold gradient set to high

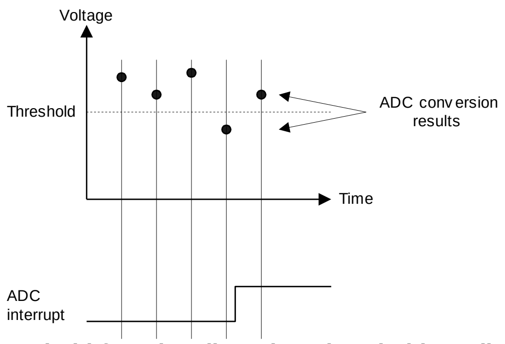

Figure 11. Threshold functionality when threshold gradient set to low

The figure below shows how the connections to the ADC can be configured:

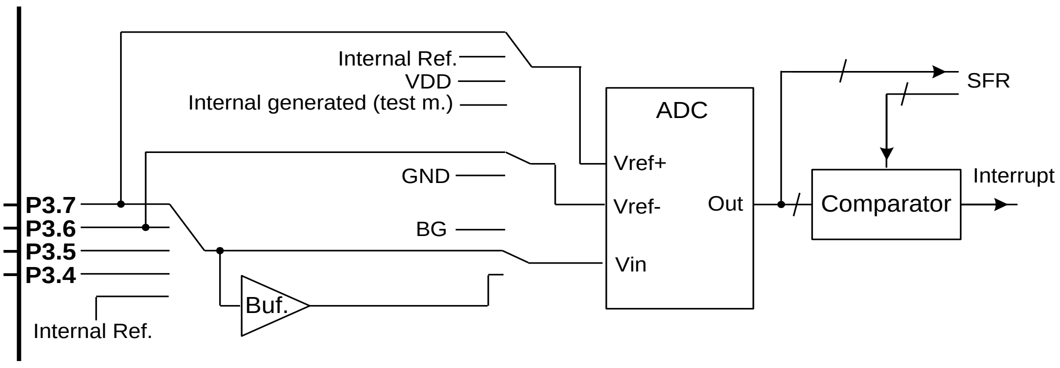

Figure 12. Configuration of input pins

The below are description of the API available to use the ADC.

5.4.11.1 ZW_ADC_init

```
void ZW_ADC_init ( BYTE bMode,
                   BYTE bUpper_ref,
                   BYTE bLower_ref,
                   BYTE bPin_en)
```

Initialize the ADC.

Defined in: ZW_adcdriv_api.h

Parameters:

bMode IN	ADC_MULTI_CON_MODE	Sets the ADC in multi conversion mode The ADC will continue converting until it is stopped.
	ADC_SINGLE_CON_MODE	Sets the ADC in single conversion mode The ADC will convert one time then stops.
	ADC_BATT_MON_MODE	Sets the ADC in battery monitoring mode. When ADC is in this mode the chip supply voltage (VDD) will be selected as upper reference. The GND will be selected as lower reference voltage. The ADC input will be the band gap circuit.
bUpper_ref	ADC_REF_U_VDD	Select the chip power supply (VDD) as the upper reference voltage. Ignored when ADC in battery monitor mode.
	ADC_REF_U_EXT	Select IO P3.7 as the upper reference voltage. Ignored when ADC in battery monitor mode.
	ADC_REF_U_BGAB	Select the band gab circuit as the upper reference voltage. Ignored when ADC in battery monitor mode.
bLower_ref	ADC_REF_L_VSS	Select the ground (VSS) as the lower reference voltage. Ignored when ADC in battery monitor mode.
	ADC_REF_L_EXT	Select IO P3.6 as lower reference voltage. Ignored when ADC in battery monitor mode.
bPin_en	Bits mask	Select which IO to enable as ADC inputs. Selected pins cannot be used as GPIOs
	ADC_PIN_1	Select I/O P3.7 as an ADC input
	ADC_PIN_2	Select I/O P3.6 as an ADC input
	ADC_PIN_3	Select I/O P3.5 as an ADC input
	ADC_PIN_4	Select I/O P3.4 as an ADC input

Serial API (Not supported)

5.4.11.2 ZW_ADC_power_enable

void ZW_ADC_power_enable(BOOL boEnable)

Turn on/off the ADC power

Defined in: ZW_adcdriv_api.h

Parameters:

boEnable IN	TRUE	Turn the ADC power on
	FALSE	Turn the ADC power off. The ADC will cancel any activity immediately.

Serial API (Not supported)

5.4.11.3 ZW_ADC_enable

void ZW_ADC_enable(BOOL boStart)

Start / stop the ADC.
When stopping the ADC while it's running in continuous mode. The ADC will continue running until the last conversion is finished.

Defined in: ZW_adcdriv_api.h

Parameters:

boStart IN	TRUE	Start the ADC and begin converting
	FALSE	Stop the ADC.

Serial API (Not supported)

5.4.11.4 ZW_ADC_pin_select

void ZW_ADC_pin_select(BYTE bAdcPin)

Select the IO to use the current input. The IO should be already enabled as an ADC input.

Defined in: ZW_adcdriv_api.h

Parameters:

bAdcPin IN	ADC_PIN_1	Select IO P37 as the current ADC input
	ADC_PIN_2	Select IO P36 as the current ADC input
	ADC_PIN_3	Select IO P35 as the current ADC input
	ADC_PIN_4	Select IO P34 as the current ADC input

Serial API (Not supported)

5.4.11.5 ZW_ADC_threshold_mode_set

void ZW_ADC_threshold_mode_set(BYTE bThresMode)

Set the ADC threshold mode.
.

Defined in: ZW_adcdriv_api.h

Parameters:

bThresMode	ADC_THRES_UPPER	The ADC fires when input is above/equal to the threshold value
	ADC_THRES_LOWER	The ADC fires when input is below/equal to the threshold value

Serial API (Not supported)

5.4.11.6 ZW_ADC_threshold_set

void ZW_ADC_threshold_set(WORD wThreshold)

Set the ADC threshold value. Depending on the threshold mode (set by **ZW_ADC_threshold_mode_set()**) , the threshold value is used to trigger an interrupt when the sampled value is above/equal or below/equal the threshold value.

Defined in: ZW_adcdriv_api.h

Parameters:

wThreshold IN	Threshold value ranges from 0 to 4095	When ADC is running in 8 bit resolution, the threshold value range is from 0 to 255. When ADC is running in 12 bit resolution, the threshold value range is from 0 to 4095. The API **ZW_ADC_resolution_set** should be called before calling this API

Serial API (Not supported)

5.4.11.7 ZW_ADC_int_enable

void ZW_ADC_int_enable(BOOL boEnable)

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 returning from the interrupt routine by calling **ZW_ADC_int_clear**.
.
Defined in: ZW_adcdriv_api.h

Parameters:

boEnable IN	TRUE	Enables the ADC interrupt.
	FALSE	Disable the ADC interrupt.

Serial API (Not supported)

5.4.11.8 ZW_ADC_int_clear

void ZW_ADC_int_clear(void)

Clear the ADC interrupt flag.
.
Defined in: ZW_adcdriv_api.h

Serial API (Not supported)

5.4.11.9 ZW_ADC_is_fired

BOOL ZW_ADC_is_fired(void)

Check if the ADC conversion crossed over the threshold value.
.
Defined in: ZW_adcdriv_api.h

Retrun value:

BOOL	TRUE	The current conversion result meet the threshold condition.
	FALSE	The ADC is not finished or the current conversion result doesn't meet the threshold condition.

Serial API (Not supported)

5.4.11.10 ZW_ADC_result_get

WORD ZW_ADC_result_get(void)

Get the value of an ADC conversion. The return value is an 8-bit or 12-bit integer 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

Retrun value:

WORD	Unsigned 16-bit value representing the result of the ADC conversion	If ADC resolution is 8 bit, the last 8 bit of the return value will be ignored (zeros). If the ADC resolution is 12-bit then the last 4 bit of the return value will be ignored (zeros).

Serial API (Not supported)

5.4.11.11 ZW_ADC_buffer_enable

void ZW_ADC_buffer_enable(BOOL boEnable)

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:

boEnable	TRUE	Enable the input buffer.
	FALSE	Disable the input buffer.

Serial API (Not supported)

5.4.11.12 ZW_ADC_auto_zero_set

void ZW_ADC_auto_zero_set(BYTE bAzpl)

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:

bAzpl	ADC_AZPL_1024	Set the autozero period to 1024 clocks. Valid for high impedance input sources.
	ADC_AZPL_512	Set the autozero period to 512 clocks. Valid for medium to high impedance input sources.
	ADC_ZPL_256	Set the autozero period to 256 clocks. Valid for medium to low impedance input sources.
	ADC_ZPL_128	Set the autozero period to 128 clocks. Valid for low impedance input sources.

Serial API (Not supported)

5.4.11.13 ZW_ADC_resolution_set

void ZW_ADC_resolution_set(BYTE bReso)

Set the resolution of the ADC.
NOTE: When changing the ADC resolution, the threshold value should also be changed.
.
Defined in: ZW_adcdriv_api.h

Parameters:

bReso	ADC_12_BIT	Set the ADC resolution to 12 bits
	ADC_8_BIT	Set the ADC resolution to 8 bits

Serial API (Not supported)

5.4.11.14 ZW_ADC_batt_monitor_enable

void ZW_ADC_batt_monitor_enable(BOOL boEnable)

Enable / disable the battery monitor mode.
When in battery monitor mode, the ADC will automatically be configured to have the VDD as upper reference voltage, VSS as lower reference voltage and the band gap as the ADC input.
If the ADC is running in 8-bit mode then the supply voltage (the input) can be calculated as follow:
supply = (Vref * 256)/(ADC conversion result).
If the ADC is running in 12-bit mode then the supply voltage can be calculated as follow:
supply = (Vref * 4096)/(ADC conversion result)
.

Defined in: ZW_adcdriv_api.h

Parameters:

bEnable	TRUE	The ADC is set in battery monitor mode.
	FALSE	The ADC is set in normal conversion mode

Serial API (Not supported)

5.4.12 UART interface API

The UART (Universal Asynchronous Receiver Transmitter) interface is for serial communication with external devices such as PC's, host controllers etc. The two UART interfaces transmits data in an asynchronous way, and is a two-way communication protocol, using 2 pins each as a communications means: TxD and RxD. The two pins can be enabled and disabled individually. If only using RX mode the TxD pin can be used as general IO pins and vice versa. The UART's use dedicated timers and do not take up any 8051 timer resources.

Since the two UART's are identical the description of each function is collapsed using the notation UARTx, where x is either 0 or 1.

The UARTx supports full duplex and can operate with the baud rates between 9.6kbaud and 230.4 kbaud. (See under **ZW_UARTx_init**)

The interface operates with 8 bit words, one start bit (low), one stop bit (high) and no parity. This setup is hardwired and can not be changed.

The UARTx shifts data in/out in the following order: start bit, data bits (LSB first) and stop bit. The figure below gives the waveform of a serial byte.

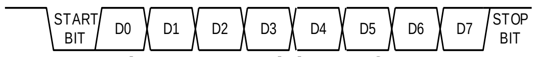

Figure 13. Serial Waveform

5.4.12.1 Transmission

An interrupt is released when D7 has been sent on the TxD pin. A new byte can be written to the buffer when the interrupt has been released.

5.4.12.2 Reception

The reception is activated by a falling edge on RxD. If the falling edge is not verified by the majority voting on the start bit, then the serial port stops reception and waits for another falling edge on RxD. When the MSB of the byte has been received a stop bit is expected. The first 2/3 of the stop bit is sampled and a majority decision is made on these samples. The interrupt will be released if the stop bit is recognized as high.

When 2/3 of the stop bit has been received the serial port waits for another high-to-low transition (start bit) on the RxD pin.

5.4.12.3 RS232

Connecting a RS232 level converter to the 2 pins of a UART interface makes the 400 Series Z-Wave Single Chip able to communicate according to the RS232 standard.

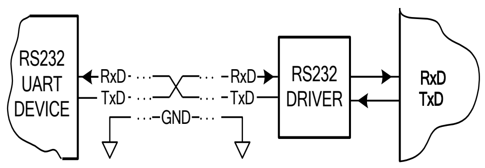

Figure 14. RS232 Setup

5.4.12.4 Integration to the Protocol

Before using the UARTx the UART should be initialized and mapped to the IO pins. This initialization should be performed in the initialization function **ApplicationInitHW**. The initialization and IO mapping is performed using the **ZW_UARTx_init** functions and if needed the **ZW_UART0_zm4102_mode_enablet** function. There are no requirements to the order of calling these functions.

The use of the UART is typically performed in the **ApplicationPoll**. The UART is then polled and characters are received / transmitted. Alternatively, the UART can be serviced in an ISR, but this approach is often to slow for higher baudrates.

A UART application typically writes a character or string to a teminal. This can be performed by initializing the modem as described above in **ApplicationInitHW** and then calling **ZW_UARTx_tx_data_wait_set(BYTE data)** for a character or **ZW_UARTx_tx_send_str(BYTE *str)** for an entire string. Both functions wait until the UART is ready before sending each character. However in some cases it is not desirable to wait until the UART is ready before continuing code execution. In this case it is better to poll to see if the UART is ready and then transmit characters when the UART is ready. In this case a different set of functions are needed as given below.

```
if (!ZW_UART0_tx_active_get())
{
    ZW_UART0_tx_data_wait_set('A');
}
```

Another possibility is to use the interrupt flags:

```
if (ZW_UART0_tx_int_get())
{
    ZW_UART0_tx_int_clear();
    ZW_UART0_tx_data_wait_set('A');
}
```

However the latter method has the disadvantage that it requires an initial write to the UART or else the first interrupt flag will not go high and the writing will never start.

Another typical UART application is to receive a character to the 400-series Z-Wave Single Chip. Similarly as for the TX setup, it is possible to call a function that waits until a character is ready, **ZW_UART0_rx_data_wait_get()**, or to poll for a new character before reading it. But in the receive case it is apparent that the wait function should be used with extreme caution as it cause a system freeze if a character is never received.

An example of the preferred solution to receive characters is given below:

```
if (ZW_UART0_rx_int_get())
{
    ZW_UART0_rx_int_clear(); // Clear flag right after detection
    ch = ZW_UART0_rx_data_get(); // Where ch is of the type BYTE
     ...
}
```

Note: It is important to clear the interrupt flag as fast as possible after detecting the interrupt flag (even before reading data). Omitting to do this may lead to loss of data as the interrupt flag may trigger again before the flag is cleared. This is especially a concern at high baudrates.

5.4.12.5 Serial Interface API

The serial interface API handles transfer of data via the serial interfaces using the 400 Series Z-Wave Single Chip built-in UART0 and UART1. This serial API supports transmissions of either a single byte, or a data buffer. The received characters are read by the application one-by-one.

5.4.12.6 ZW_UART0_init / ZW_UART1_init

void ZW_UART0_init(WORD bBaudRate, BOOL bEnableTx, BOOL bEnableRx) /
void ZW_UART1_init(WORD bBaudRate, BOOL bEnableTx, BOOL bEnableRx)

Initializes the 400 Series Z-Wave Single Chip built-in UARTx to support ZM4101 and SD3402. Using ZM4102 requires an additional call **ZW_UART0_zm4102_mode_enablet** to map to correct pin configuration. The order of calling these functions are optional but the functions should be called in the **ApplicationInitHW()** so the ports are mapped correctly when the chip starts up.

The init functions optionally enable/disable UARTx transmit and/or receive, clears the rx and tx interrupt flags and sets the specified baud rate.

Defined in: ZW_uart_api.h

Parameters:

bBaudRate IN	Baud Rate / 100	Valid values: 96 $\Rightarrow$ 9.6kbaud, 144 $\Rightarrow$ 14.4kbaud, 192 $\Rightarrow$ 19.2kbaud, 384 $\Rightarrow$ 38.4kbaud, 576 $\Rightarrow$ 57.6kbaud, 1152 $\Rightarrow$ 115.2kbaud, 2304 $\Rightarrow$ 230.4kbaud
bEnableTx IN	TRUE	Enable UARTx transmitter and allocate TxD pin. (UART0 TxD is allocated on P2.1) (UART1 TxD is allocated on P3.1)
	FALSE	Disable UARTx Transmitter and de-allocate TxD pin
bEnableRx IN	TRUE	Enable UARTx receiver and allocate RxD pin (UART0 RxD is allocated on P2.0) (UART1 RxD is allocated on P3.0)

	FALSE	Disable UARTx receiver and de-allocate RxD pin

Serial API (Not supported)

5.4.12.7 ZW_UART0_zm4102_mode_enable

void ZW_UART0_zm4102_mode_enable(BOOL bState)

Map pins of the 400 Series Z-Wave Single Chip built-in UART0 to support ZM4102. Only available on UART0. Refer also to **ZW_UART0_init / ZW_UART1_init**

Defined in: ZW_uart_api.h

Parameters:

bState IN	TRUE	Support ZM4102 built-in UART0 by mapping TxD to IO P3.5 and RxD to IO P3.4
	FALSE	Support ZM4101/SD3402 built-in UART0 by mapping TxD to IO P2.1 and RxD to IO P2.0

Serial API (Not supported)

5.4.12.8 ZW_UART0_rx_data_get / ZW_UART1_rx_data_get

BYTE ZW_UART0_rx_data_get(void) / BYTE ZW_UART1_rx_data_get(void)

This function returns the last received byte from UARTx. The UART should be polled using the **ZW_UART0_rx_int_get / ZW_UART1_rx_int_get** to see whether a new byte is ready before calling this function.

The function does not wait for a byte to be received but returns immediately. The alternative functions **ZW_UART0_rx_data_wait_get / ZW_UART1_rx_data_wait_get** waits until a byte is received before returning.

Defined in: ZW_uart_api.h

Return value:

BYTE	Received data.

Serial API (Not supported)

5.4.12.9 ZW_UART0_rx_data_wait_get / ZW_UART1_rx_data_wait_get

BYTE ZW_UART0_rx_data_wait_get(void) / BYTE ZW_UART1_rx_data_wait_get(void)

Returns a byte from the UARTx receiver. If no byte is available the function waits until data has been received. This function should be used with *extreme caution* as it may freeze the system if no character is received. In normal cases it is better to use polling, **ZW_UART0_rx_int_get / ZW_UART1_rx_int_get**, to check if a new byte is received and then **ZW_UART0_rx_data_get / ZW_UART1_rx_data_get** to get the byte.

Defined in: ZW_uart_api.h

Return value:

BYTE	Received data.

Serial API (Not supported)

5.4.12.10 ZW_UART0_tx_active_get / ZW_UART1_tx_active_get

BYTE ZW_UART0_tx_active_get(void) / BYTE ZW_UART1_tx_active_get(void)

Read the UARTx send data status. The function returns TRUE if the UART is currently busy transmitting data. The function is typically used in a polled TX setup to check whether the UART is ready before sending the next character using **ZW_UART0_tx_data_set / ZW_UART1_tx_data_set**.

Alternatively to creating the poll-transmit loop it is simpler to use **ZW_UART0_tx_data_wait_set** / ZW_UART1_tx_data_wait which automatically waits for the transmitter before sending the data.

Defined in: ZW_uart_api.h

Return value:

BYTE	non-zero	UARTx Transmitter is busy
	zero (0x00)	UARTx Transmitter is idle

Serial API (Not supported)

5.4.12.11 ZW_UART0_tx_data_wait_set / ZW_UART1_tx_data_wait_set

void ZW_UART0_tx_data_wait_set(BYTE data) / void ZW_UART1_tx_data_wait_set(BYTE data)

(Earlier this function was called ZW_UARTx_tx_send_byte(BYTE data). The function name has been changed for consistency in naming)

The function transmits a byte over the UARTx.

This function waits until UARTx transmitter is idle before it sends the new data. The function does not wait until the new data has been sent before returning.

An alternative function is the **ZW_UART0_tx_data_set / ZW_UART1_tx_data_set** which returns immediately but requires polling to check for readiness.

Defined in:	ZW_uart_api.h

Parameters:

data IN	Data to send.

Serial API (Not supported)

5.4.12.12 ZW_UART0_tx_data_set / ZW_UART1_tx_data_set

void ZW_UART0_tx_data_set(BYTE data) / void ZW_UART1_tx_data_set(BYTE data)

Function sets the transmit data register

This function does not wait until UARTx transmitter is idle before it sends the new data. The function should not be called unless the UART is ready. To check if the UART is ready is done using the **ZW_UART0_tx_active_get / ZW_UART1_tx_active_get**. Data send to the UART when it is not ready will be ignored.

The function does not wait until the new data has been sent before returning. An simpler alternative is to use the **ZW_UART0_tx_data_wait_set** / ZW_UART1_tx_data_wait function.

Defined in:	ZW_uart_api.h

Parameters:

data IN	Data to send.

Serial API (Not supported)

5.4.12.13 ZW_UART0_tx_send_num / ZW_UART1_tx_send_num

void ZW_UART0_tx_send_num(BYTE data) / void ZW_UART1_tx_send_num(BYTE data)

Converts a byte to a two-byte hexadecimal ASCII representation, and transmits it over the UART. This function waits until UARTx transmitter is idle before it sends the new data. The function does not wait until the last data byte has been sent.

See also: **ZW_UART0_tx_send_str / ZW_UART1_tx_send_str** and **ZW_UART0_tx_send_nl / ZW_UART1_tx_send_nl**

Defined in: ZW_uart_api.h

Parameters:

data IN	Data to convert and send.

Serial API (Not supported)

5.4.12.14 ZW_UART0_tx_send_str / ZW_UART1_tx_send_str

void ZW_UART0_send_str(BYTE *str) / void ZW_UART1_send_str(BYTE *str)

Transmit a null terminated string over UARTx. The null data is not transmitted. This function waits until UARTx transmitter is idle before it sends the first data byte data. The function does not wait until the last data byte has been sent.

See also: **ZW_UART0_tx_send_num / ZW_UART1_tx_send_num** and **ZW_UART0_tx_send_nl / ZW_UART1_tx_send_nl**

Defined in: ZW_uart_api.h

Parameters:

str IN	String pointer.

Serial API (Not supported)

5.4.12.15 ZW_UART0_tx_send_nl / ZW_UART1_tx_send_nl

void ZW_UART0_tx_send_nl(void) / void ZW_UART1_tx_send_nl(void)

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

See also: **ZW_UART0_tx_send_num / ZW_UART1_tx_send_num** and **ZW_UART0_tx_send_str / ZW_UART1_tx_send_str**

Defined in: ZW_uart_api.h

Serial API (Not supported)

5.4.12.16 ZW_UART0_tx_int_clear / ZW_UART1_tx_int_clear

void ZW_UART0_tx_int_clear(void) / void ZW_UART1_tx_int_clear(void)

Clear the UARTx transmit interrupt/done flag.

See also: **ZW_UART0_tx_int_get / ZW_UART1_tx_int_get**

Defined in: ZW_uart_api.h

Serial API (Not supported)

5.4.12.17 ZW_UART0_rx_int_clear / ZW_UART1_rx_int_clear

void ZW_UART0_rx_int_clear(void) / void ZW_UART1_rx_int_clear(void)

Clear the UARTx receiver interrupt/ready flag.

See also: **ZW_UART0_rx_int_get / ZW_UART1_rx_int_get**

Defined in: ZW_uart_api.h

Serial API (Not supported)

5.4.12.18 ZW_UART0_tx_int_get / ZW_UART1_tx_int_get

BYTE ZW_UART0_tx_int_get(void) / BYTE ZW_UART1_tx_int_get(void)

Returns the state of the Transmitter done/interrupt flag. This function has limited used and in practice it is preferred to check if the UART is ready using **the ZW_UART0_tx_active_get / ZW_UART1_tx_active_get** function in a polled configuration. The **ZW_UART0_tx_active_get / ZW_UART1_tx_active_get** does not require the interrupt flag to be cleared.

See also : **ZW_UART0_tx_int_clear / ZW_UART1_tx_int_clear**

Defined in: ZW_uart_api.h

Return value:

BYTE	non-zero	UARTx Transmitter done/interrupt flag is set
	zero (0x00)	UARTx Transmitter done/interrupt flag is cleared

Serial API (Not supported)

5.4.12.19 ZW_UART0_rx_int_get / ZW_UART1_rx_int_get

BYTE ZW_UART0_rx_int_get(void) / BYTE ZW_UART1_rx_int_get(void)

Returns the state of the receiver data ready/interrupt flag. The flag goes high when a new byte has been received. The flag should be cleared as soon as possible after detection in order to minimize risk of data loss (especially at high baud rates). Clearing the interrupt flag is done using the function **ZW_UART0_rx_int_clear / ZW_UART1_rx_int_clear**. When a new byte is detected the byte can be read using the **ZW_UART0_rx_data_get / ZW_UART1_rx_data_get** function.

See also: **ZW_UART0_rx_int_clear / ZW_UART1_rx_int_clear** and **ZW_UART0_rx_data_get / ZW_UART1_rx_data_get**

Defined in: ZW_uart_api.h

Return value:

BYTE	non-zero	UARTx Receiver data ready/interrupt flag is set
	zero (0x00)	UARTx receiver data ready/interrupt flag is cleared

Serial API (Not supported)

5.4.13 Application HW Timers/PWM interface API

The 400 Series Z-Wave Single Chip has two built-in HW timers available for the application:

1. Timer0
2. GPTimer or PWM generator.

Timer	bits	Clocked by	Count up/down
Timer0	8/13/16	32MHz / 2 or P1.0	Counts up
GPTimer	16	32MHz / 8 or 32MHz / 1024	Counts down

Timer0 is a standard 8051 timer that can be configured to:

- be enabled/disabled
- use the system clock divided by 2 (16MHz) or use a pin as clock source
- use a pin as clock gate as an option
- generate an interrupt at overflow

Refer to figure below for principle diagrams of how the clock control works for Timer0.

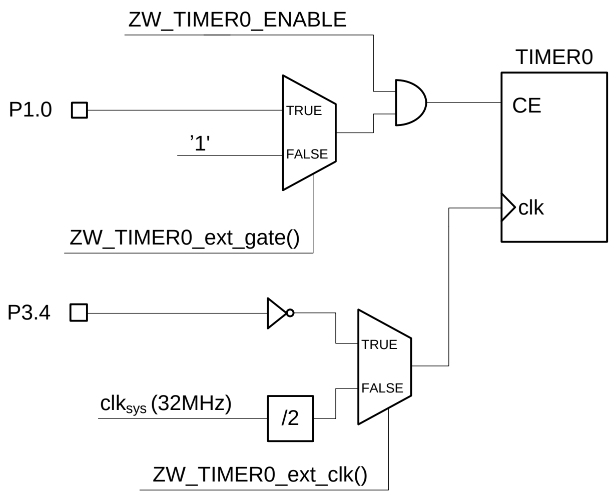

Figure 15. Principle of Timer0's clock control

Timer0 can operate in three different modes. Refer to the description of ZW_TIMER0_init. The protocol uses the standard 8051 Timer1.

The GPTimer can optionally be used as a PWM generator instead of a timer.

5.4.13.1 ZW_TIMER0_init

void ZW_TIMER0_init(BYTE bValue)

This function initializes Timer0. However,

Defined in: ZW_appltimer_api.h

Parameters:

bValue	Timer0 Mode:	
	TIMER_MODE_0	13 bit mode. The 5 lower bits of the low register acts as a 5 bit prescaler for the high byte
	TIMER_MODE_1	16 bit mode (no reload)
	TIMER_MODE_2	8bit - auto reload mode. The 8bit timer runs in the high byte register. After an overflow the low byte register value is loaded into the high byte register

Serial API (Not supported)

5.4.13.2 ZW_TIMER0_INT_CLEAR

ZW_ TIMER0_INT_CLEAR

This macro clears the TIMER0 receiver interrupt/overflow flag

Defined in: ZW_appltimer_api.h

Serial API (Not supported)

5.4.13.3 ZW_TIMER0_INT_ENABLE

ZW_TIMER0_INT_ENABLE(BYTE bState)

This macro enables or disables the Timer0 interrupt.

Defined in: ZW_ appltimer _api.h

Parameters:

bState IN	TRUE	enable TIMER0 interrupt
	FALSE	disable TIMER0 interrupt

Serial API (Not supported)

5.4.13.4 ZW_TIMER0_ENABLE

ZW_TIMER0_ENABLE(BYTE bState)

This macro enables or halts the Timer0.

Defined in: ZW_appltimer_api.h

Parameters:

bState IN	TRUE	TIMER0 runs
	FALSE	TIMER0 is halted

Serial API (Not supported)

5.4.13.5 ZW_TIMER0_ext_clk

ZW_TIMER0_ext_clk(BOOL bState)

This function set the clock source for Timer0

Defined in: ZW_appltimer_api.h

Parameters:

bState IN	TRUE	Timer0 runs on the signal on pin P3.4 (synchronized to the system clock)
	FALSE	Timer0 run on the system clock divided by 2

Serial API (Not supported)

5.4.13.6 ZW_TIMER0_ext_gate

ZW_TIMER0_ext_gate(BOOL bState)

This function enables/disables to use of P1.0 as external clock gate for Timer0

Defined in: ZW_appltimer_api.h

Parameters:

bState IN	TRUE	The clock for Timer0 can be gated by P1.0. The clock runs when this function is called with the parameter set to true, ZW_TIMER0_ENABLE has be called with the parameter set to TRUE and when P1.0 is high
	FALSE	The clock for Timer0 is not gated by P1.0. The clock is only controlled by the macro ZW_TIMER0_ENABLE.

Serial API (Not supported)

5.4.13.7 ZW_TIMER0_LOWBYTE_SET

ZW_TIMER0_LOWBYTE_SET (BYTE bValue)

This macro sets the timer0 value, see below.

Defined in: ZW_appltimer_api.h

Parameters:

bValue IN The input value depends on the chosen mode:

Mode0:	Lower 5 bits sets the prescaler value for the 13 bit timer
Mode1:	Sets the lower 8 bits of the 16 bit timer
Mode2:	N.A.
Mode3:	Sets the 8 bit timer of the 8 bit timer0

Serial API (Not supported)

5.4.13.8 ZW_TIMER0_HIGHBYTE_SET

ZW_TIMER0_HIGHBYTE_SET (BYTE bValue)

This macro sets the timer0 value, see below.

Defined in: ZW_appltimer_api.h

Parameters:

bValue IN	The input value depends on the chosen mode: Mode0: Sets the 8 bit timer value Mode1: Sets the upper 8 bits of the 16 bit timer Mode2: Sets the 8 bit reload value of the 8 bit timer0 Mode3: Sets the 8 bit timer of the 8 bit timer1

Serial API (Not supported)

5.4.13.9 ZW_TIMER0_HIGHBYTE_GET

ZW_TIMER0_HIGHBYTE_GET

This macro returns the Timer 0 timer high register value

Defined in: ZW_appltimer_api.h

Return value:

BYTE	The return value depends on the chosen mode: Mode0: 8 bit timer value Mode1: upper 8 bits of the 16 bit timer Mode2: 8 bit timer value Mode3: 8 bit timer value (timer 1 interrupt)

Serial API (Not supported)

5.4.13.10 ZW_TIMER0_LOWBYTE_GET

ZW_TIMER0_LOWBYTE_GET

This function returns the Timer 0 timer low register value

Defined in: ZW_appltimer_api.h

Return value:

BYTE	The return value depends on the chosen mode: Mode0: 5 bit prescaler value for the 13 bit timer. (lower 5 bits)

Mode1: lower 8 bits of the 16 bit timer
Mode2: 8 bit timer value
Mode3: 8 bit timer value

Serial API (Not supported)

5.4.13.11 ZW_TIMER0_word_get

WORD ZW ZW_TIMER0_word_get (void)

This function returns the two 8 bit Timer 0 register values as one 16 bit value. Used when timer0 is set in mode 1.

Defined in: ZW_appltimer_api.h

Return value:

WORD	16bit timer value

Serial API (Not supported)

5.4.13.12 ZW_GPTIMER_init

void ZW_GPTIMER_init(BYTE bValue)

This function initializes the GPTimer. Calling ZW_GPTIMER_init() will disable the PWM, since the GP Timer and the PWM share hardware. The GPTimer counts down.

Defined in: ZW_appltimer_api.h

Parameters:

bValue IN	Bit mask:	
	Prescaler setting	
	PRESCALER_BIT	When set: Timer counter runs @ 32MHz / 1024 = 31.25kHz When nor set: Timer counter runs @ 32MHz / 8 = 4MHz
	Reload Timer	
	RELOAD_BIT	When set: The GPTimer counter registers are reloaded with the reload register value upon underrun. When not set: The GPTimer stops upon underrun.
	Immediate write	
	IMWR_BIT	When set: The GP Timer counters will be loaded with the value of the reload register when it is disabled or immediately when the reload values are set. When not set: The GP Timer counters will be loaded with the value of the reload register when it is disabled or when it times out (underrun).

Serial API (Not supported)

5.4.13.13 ZW_GPTIMER_int_clear

void ZW_GPTIMER_int_clear (void)

This function clears the GP Timer interrupt flag.

Defined in: ZW_appltimer_api.h

Serial API (Not supported)

5.4.13.14 ZW_GPTIMER_int_get

BYTE ZW_GPTIMER_int_get (void)

This function returns the state of the GP Timer interrupt flag.

Defined in: ZW_appltimer_api.h

Return value:

BYTE	0x00: interrupt flag is not set mom-0x00: Interrupt is set

Serial API (Not supported)

5.4.13.15 ZW_GPTIMER_int_enable

void ZW_GPTIMER_int_enable(BYTE bState)

This function enables or disables the GPTimer interrupt

Defined in: ZW_appltimer_api.h

Parameters:

bState IN	TRUE	enable GPTimer interrupt
	FALSE	disable GPTimer interrupt

Serial API (Not supported)

5.4.13.16 ZW_GPTIMER_enable

void ZW_GPTIMER_enable(BYTE bState)

This function enables or disables the GPTimer and clears the interrupt flag. The GPTimer counters are reset when the GPTimer is disabled.

Defined in: ZW_appltimer_api.h

Parameters:

bState IN	TRUE	enable GPTimer.
	FALSE	disable GPTimer.

Serial API (Not supported)

5.4.13.17 ZW_GPTIMER_pause

void ZW_GPTIMER_pause(BYTE bState)

This function enters or leaves GPTimer pause state. When entering the pause state, the GPTimer counters stops counting. When leaving the pause state the counters will start counting from the state they were in when the pause state was entered.

Defined in: ZW_appltimer_api.h

Parameters:

bState IN	TRUE	Enter GPTimer pause state.
	FALSE	Leave GPTimer pause state.

Serial API (Not supported)

5.4.13.18 ZW_GPTIMER_reload_set

void ZW_GPTIMER_reload_set(WORD wReloadValue)

This function sets the 16 bit GPTimer reload register. This value sets the time from where the GPTimer is enabled or it reloaded until the it under-runs (issues an interrupt). E.g. if the GPtimers reload value is set to 0x0137 and the prescaler is set to 1024, the underrun will happen after 0x137 * 1024 * $(32MHz)^{-1}$ = 9.95ms.

The value 0x0000 equals a timer reload value of 0x10000. E.g. if the GPtimers reload value is set to 0x0000 and the prescaler is set to 8, the underrun will happen after 0x10000 * 8 * $(32MHz)^{-1}$ = 16.38ms.

Defined in: ZW_appltimer_api.h

Parameters:

wReloadValue	IN	16 bit reload value

Serial API (Not supported)

5.4.13.19 ZW_GPTIMER_reload_get

WORD ZW_GPTIMER_reload_get(void)

This function returns the 16 bit GPTimer reload register value.

Defined in: ZW_appltimer_api.h

Return value:

WORD	16 bit reload value

Serial API (Not supported)

5.4.13.20 ZW_GPTIMER_get

WORD ZW_GPTIMER_get(void)

This function returns the 16 bit GPTimer counter register value. That is it returns a value in the range [reload_value-1;0]. E.g. if the reload value is set to 0x2A40, ZW_GPTIMER_reload_get() will return a value in the range [0x2A3F;0].

Defined in: ZW_appltimer_api.h

Return value:

WORD	16 bit counter value

Serial API (Not supported)

5.4.13.21 ZW_PWM_init

void ZW_PWM_init(BYTE bValue)

This function initializes the pulse width modulator. Calling ZW_PWM_init() will disable the GPTimer function, since the PWM and the GP Timer share hardware.

Defined in: ZW_appltimer_api.h

Parameters:

bValue IN	Bit mask:	
	Prescaler setting	
	PRESCALER_BIT	When set: PWM counter runs @ 32MHz / 1024 = 31.25kHz When nor set: PWM counter runs @ 32MHz / 8 = 4MHz
	Invert signal	
	IMWR_BIT	When set: PWM signal is inverted. When not set: The signal is not inverted
	Immediate write	
	IMWR_BIT	When set: The PWM counters will be loaded with the value of the waveform registers when it is disabled or immediately when the waveform values are set. When not set: The PWM counters will be loaded with the value of the waveform registers when it is disabled or at the end of a PWM signal period.

Serial API (Not supported)

5.4.13.22 ZW_PWM_enable

void ZW_PWM_enable(BYTE bState)

This function enables or disables the PWM and clears the interrupt flag. The PWM counters are reset when it is disabled.

Defined in: ZW_appltimer_api.h

Parameters:

bState IN	TRUE	enable PWM.
	FALSE	disable PWM.

Serial API (Not supported)

5.4.13.23 ZW_PWM_int_clear

void ZW_PWM_int_clear (void)

Function clears the PWM interrupt flag.

Defined in: ZW_appltimer_api.h

Serial API (Not supported)

5.4.13.24 ZW_PWM_int_get

BYTE ZW_PWM_int_get (void)

Function returns the state of the PWM interrupt flag.

Defined in: ZW_appltimer_api.h

Return value:

BYTE	0x00: interrupt flag is not set non-0x00: Interrupt is set

Serial API (Not supported)

5.4.13.25 ZW_PWM_int_enable

void ZW_PWM_int_enable(BYTE bState)

This function enables or disables the PWM interrupt
The PWM interrupt is triggered on the rising edge of the PWM signals (or at the falling edge of the PWM signal if PWMINV_BIT is set in ZW_PWM_init()).

Note: The Interrupt should be disabled when either the high or low value in ZW_PWM_waveform_set() is set to zero.

Defined in: ZW_appltimer_api.h

Parameters:

bState IN	TRUE	enable PWM interrupt
	FALSE	disable PWM interrupt

Serial API (Not supported)

5.4.13.26 ZW_PWM_waveform_set

void ZW_PWM_waveform_set (BYTE bHigh,
BYTE bLow)

This function sets the low and high time of the PWM signal. Refer to figure below.

High time of PWM signal:	t_{hPWM} = (bValueHigh * PRESCALER_BIT)/f_{sys}
Low time of PWM signal	t_{lPWM} = (bValueLow * PRESCALER_BIT) /f_{sys}
Total period of PWM signal:	$T_{PWM} = t_{hPWM} + t_{lPWM}$

where f_{sys} is 32MHz and
PRESCALER_BIT is 1 when set and 0 when not set.

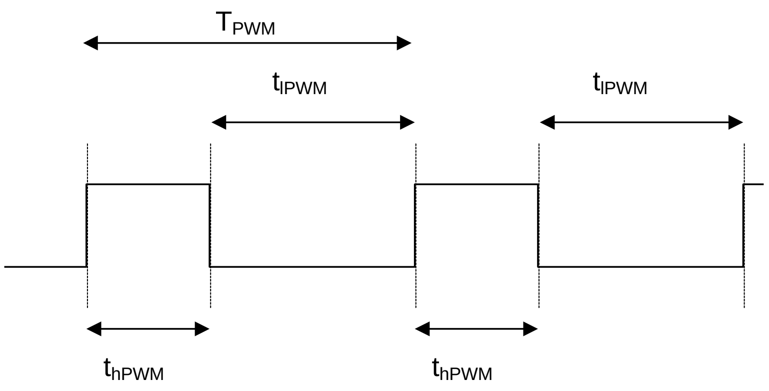

Figure 16. PWM waveform

Defined in: ZW_appltimer_api.h

Parameters:

bHigh IN	high time
bLow IN	low time

Serial API (Not supported)

5.4.13.27 ZW_PWM_waveform_get

void ZW_waveform_get(BYTE *bValue,
BYTE *bLow)

This function returns the values of the waveform registers.

Defined in: ZW_appltimer_api.h

Parameters:

bHigh OUT	high time
bLow OUT	low time

Serial API (Not supported)

5.4.14 AES API (Only available in a secure SDK)

The built-in AES-128 hardware engine is a NIST standardized AES 128 block cipher. The cipher engine is used by the Z-Wave Protocol to encrypt/decrypt Z Wave frame payload and to authenticate Z Wave frames. In addition this AES-128 encryption engine can also be used to encrypt a 128bit data block (Using ECB - Electronic CookBook mode) by the application.

The input and output data and key for the AES API's are 16 bytes long char arrays. **ZW_AES_ecb_set** is used to set the input data (plaintext and key) and the function **ZW_AES_ecb_get** is used to return the cipher data from the AES engine. The ECB process is started using the function **ZW_AES_ecb_enable(TRUE)** and it lasts about 24µs. The process can be canceled by calling **ZW_AES_ecb_enable(FALSE).** The AES engine must be polled, using the function **ZW_AES_ecb_active** to check when a ECB process is done. Figure below gives an example of how the AES engine functions are called.

```
/* Example of ECB ciphering. Vectors are from FIPS-197 */

void ApplicationPoll()
{
    :
   switch (mainState)
   {
    :
   case START_AES_TEST:
      keybuffer[15] =  0x00;
      keybuffer[14] =  0x01;
      keybuffer[13] =  0x02;
      keybuffer[12] =  0x03;
      keybuffer[11] =  0x04;
      keybuffer[10] =  0x05;
      keybuffer[9 ] =  0x06;
      keybuffer[8 ] =  0x07;
      keybuffer[7 ] =  0x08;
      keybuffer[6 ] =  0x09;
      keybuffer[5 ] =  0x0A;
      keybuffer[4 ] =  0x0B;
      keybuffer[3 ] =  0x0C;
      keybuffer[2 ] =  0x0D;
      keybuffer[1 ] =  0x0E;
      keybuffer[0 ] =  0x0F;

      plainbuffer[15] = 0x00;
      plainbuffer[14] = 0x11;
      plainbuffer[13] = 0x22;
      plainbuffer[12] = 0x33;
      plainbuffer[11] = 0x44;
      plainbuffer[10] = 0x55;
      plainbuffer[9]  = 0x66;
      plainbuffer[8]  = 0x77;
      plainbuffer[7]  = 0x88;
      plainbuffer[6]  = 0x99;
      plainbuffer[5]  = 0xAA;
      plainbuffer[4]  = 0xBB;
      plainbuffer[3]  = 0xCC;
      plainbuffer[2]  = 0xDD;
      plainbuffer[1]  = 0xEE;
      plainbuffer[0]  = 0xFF;

      cipherbuffer[15] = 0x69;
      cipherbuffer[14] = 0xC4;
      cipherbuffer[13] = 0xE0;
      cipherbuffer[12] = 0xD8;
      cipherbuffer[11] = 0x6A;
      cipherbuffer[10] = 0x7B;
      cipherbuffer[9]  = 0x04;
      cipherbuffer[8]  = 0x30;
      cipherbuffer[7]  = 0xD8;
      cipherbuffer[6]  = 0xCD;
      cipherbuffer[5]  = 0xB7;
      cipherbuffer[4]  = 0x80;
      cipherbuffer[3]  = 0x70;
      cipherbuffer[2]  = 0xB4;
      cipherbuffer[1]  = 0xC5;
      cipherbuffer[0]  = 0x5A;
      /* Set AES ECB input data pointers */
      ZW_AES_ecb_set(plainbuffer,keybuffer);
      /* Start AES ECB function */
      ZW_AES_enable(TRUE);
      mainState= WAIT_AES_ECB;
      break;
    case WAIT_AES_ECB:
      /* Check to se if AES ECB procedure is done */
      if (ZW_AES_active_get()==FALSE)
      {
        ZW_AES_ecb_get(plainbuffer);
        /* check against proven data */
        fail=FALSE;
        for (i=0;i<16;i++)
        {
          if (plainbuffer[i]!=chipherbuffer[i])
          {
            fail=TRUE;
            break;
          }
        }
        if (fail) report();

        mainState= IDLE;
      }
        :
      break;
    }
}
```

Figure 17. Example of ECB ciphering. Vectors are from FIPS-197.

5.4.14.1 ZW_AES_ecb_set

```
void ZW_AES_ecb_set (BYTE *bData,
                     BYTE *bKey)
```

Call this function to run the AES in ECB mode (Electronic Cookbook mode).

Defined in: ZW_AES_api.h

Parameters:

bData	Array of 16 bytes	Pointer to byte array containing the data to be encrypted.
bKey	Array of 16 bytes	Pointer to byte array containing the encryption key

Serial API (Not supported)

5.4.14.2 ZW_AES_ecb_get

```
void ZW_AES_ecb_get(BYTE *bData)
```

After calling **ZW_AES_ecb_set** , use **ZW_AES_active_get** to see if the AES process is done. When this is the case, call **ZW_AES_ecb_set** to transfer the result of a AES ECB process to the array bData.

Defined in: ZW_AES_api.h

Parameters:

bData	Array of 16 bytes	Pointer to byte array buffer to store the data in.

Serial API (Not supported)

5.4.14.3 ZW_AES_enable

void ZW_AES_enable(BOOL bState)

Call **ZW_AES_enable(TRUE)** to enable the AES engine and start the ECB process. The AES engine will automatically be disabled when a ECB process is done. Call **ZW_AES_enable(FALSE)** if a ECB process is to be canceled.

Defined in: ZW_AES_api.h

Parameters:

bState	TRUE	Enable the AES and start the ECB mode.
	FALSE	Disable the AES.

Serial API (Not supported)

5.4.14.4 ZW_AES_active_get

BYTE ZW_AES_active_get (void)

Returns the active/idle state of the AES engine. Use this function to see when a ECB process is done.

Defined in: ZW_AES_api.h

Return value:

BOOL	TRUE	The AES is busy.
	FALSE	The AES is idle.

Serial API (Not supported)

5.4.15 TRIAC Controller API

The built-in TRIAC Controller is targeted at controlled light / power dimming applications. The Triac Controller is able to drive both TRIAC's and FET's/IGBT's. The Triac Controller can dim the load with a precision of 1000 steps in each half-period.

When controlling TRIAC's the Triac Controller will generate one or more fire pulses in each half period of the mains to turn on the Triac. The fire angle if set by the specified dim level. The Triac will turn off when the current is close to zero at the end of the half period. The Triac Controller will generate multiple pulses if the fire angle is less than 90º. The multiple pulses ensure that at least one pulse is located after the middle of a half-period, thereby ensuring that the Triac will be fired even with fully inductive loads, and still limiting the current consumption.

When controlling a FET/IGBT the Triac Controller will turn on the FET/IGBT at a Zero-X of the mains and turn off the FET/IGBT later in the half-period according to the specified dim level.

The Triac Controller can operate in both 50Hz and 60Hz environments.

The application software can use the following TRIAC API calls to control the 400 Series Z-Wave Single Chip TRIAC Controller.

5.4.15.1 ZW_TRIAC_init

BYTE ZW_TRIAC_init(BOOL bMode,
WORD wPulseLength,
BYTE bPulseRepLength,
BYTE bZeroXMode,
BYTE bInitMask,
BYTE bInvZerox,
BYTE bMainsFreq,
WORD wCorrection,
BYTE bCorPrescale
BYTE bKeepOff)

ZW_TRIAC_init initializes the 400 Series Z-Wave Single Chip's integrated TRIAC controller. Refer to the section after the function parameter list for a description of the setup of the different zero-cross modes (page 161). Place this function call in **ApplicationInitHW.**

Defined in: ZW_triac_api.h

Parameters:

bMode IN	Mode of operation: FALSE: Triac Mode TRUE: FET/IGBT Mode	
wPulseLength IN	Triac Fire pulse length Legal values: 1-1023 Each step equals $\frac{n}{32MHz}$, where n is 265 in 60Hz systems 318 in 50Hz systems i.e. setting this parameter to 40 in a 50Hz system gives a Triac pulse length of $40 \cdot \frac{318}{32MHz} = 397.5us$	Not applicable in FET Mode Set this parameter so that is equals the minimum Triac gate high time according to the datasheet of the specific Triac in use.

bPulseRepLength IN	Triac fire pulse repetition period	Not applicable in FET Mode
	Legal values: 18-255 and bPulseRepLength **must** be larger than wPulseLength/4:	This parameter sets the period from the rising edge of one fire pulse to the rising edge of the next fire pulse.
	Each step equals $\frac{4 \cdot n}{32MHz}$, where n is 265 in 60Hz systems 318 in 50Hz systems	The Triac Controller will generate multiple fire pulses when the fire angle is less than 90°
	i.e. setting this parameter to 20 in a 50Hz system gives a Triac pulse length of $20 \cdot \frac{4 \cdot 318}{32MHz} = 795us$	
bZeroXMode IN	Bridge types:	
	TRIAC_FULLBRIDGE	The TRIAC signal is triggered only on the rising edges of the ZEROX signal.
	TRIAC_HALFBRIDGE_A	The TRIAC signal is triggered on the rising and the falling edge of the ZEROX signal.
	TRIAC_HALFBRIDGE_B	The TRIAC signal is triggered on the rising[****] edge of the ZEROX signal in every second halfperiod.
bInitMask IN	Initial zero-cross mask:	
	TRUE	Mask out noise impulse noise on the mains from the point of a detected zero-cross to the start of the Triac fire pulse
	FALSE	Do not Mask out impulse noise on the mains from the point of a detected zero-cross to the start of the Triac fire pulse
bInvZerox IN	Inverse zero-cross signal:	
	TRUE	Inverse zero-cross signal
	FALSE	Do not inverse zero-cross signal
bMainsFreq IN	AC mains frequency:	
	FREQUENCY_50HZ	Using 50Hz AC mains supply
	FREQUENCY_60HZ	Using 60Hz AC mains supply

[****] If bInitMask is set to TRUE, the Triac controller will trigger on a falling edge in every second half-period

wCorrection IN	ZeroX Duty-Cycle correction Legal values: 0-1023. The Triac controller has a timer, that can compensate for a non-50/50 duty cycle of the ZeroX signal. The timer can run on a prescalered clock (see bCorPrescale below) I.e. setting this parameter to 300 and bCorPrescale to '1' gives a correction of $300 \cdot \frac{3}{32MHz} = 28.1\mu s$.	Half Bridge Mode A: The parameter is used to compensate from a ZeroX signal duty-cycle that is not exactly 50/50, in half bridge mode A. Typically, the high time of a ZeroX signal in half bridge mode is shorter than the low time. In this case, setting this parameter to value greater than 0, can correct this mismatch. If the high time is N ns longer than the low time, this parameter should be set so that it equals N/2 ns. Half Bridge Mode B: 60Hz systems: This value should be set to $\left\lceil 26 + 88\frac{1}{3} \cdot \text{bKeepOff} \right\rceil$ 50Hz systems: This value should be set to $31 + 106 \cdot \text{bKeepOff}$ Full Bridge Mode: N.A.

bCorPrescale	Correction prescaler Legal values: 0: Prescaler disabled 1: Prescaler enabled	When this parameter is set to 1, the clock signal that is used for the correction timer (see under wCorrection above) is prescaled by a factor of 3. That is, the timer clock will run at 32MHz/3~10.67MHz When this parameter is set to 0, the correction timer will run using the system clock (32.00Mz). Half Bridge Mode A: Set this parameter to 1 if the needed correction has to be longer than 1023*(32MHz)$^{-1}$ = 31.97 µs Half Bridge Mode B: N.A. Full Bridge Mode: N.A.
bKeepOff	KeepOff distance Legal values: 0-9 Each step equals $\frac{n}{32MHz}$, where n is 265 in 60Hz systems 318 in 50Hz systems i.e. setting this parameter to 3 in a 50Hz system gives a distance of $3 \cdot \frac{318}{32MHz} = 29.8\mu s$	Use this parameter to specify the minimum distance from the falling edge of the Triac pulse to the zero cross of the mains signal (ZeroX). This parameter will also specify the distance from where the Triac controller starts looking for a new ZeroX to the nominal ZeroX point. That is, use this parameter in regions where the mains frequency has large deviations.

Return values:

BYTE	0x01	bPulseRepLength is less than wPulseLength/4:
	0x02	bPulseRepLength is less than 18 (legal values 18-255)

Serial API (Not supported)

Half bridge A:

In this mode, the Triac Controller uses both edges on the zero-cross signal for each period of the mains signal. That is, the zero-cross signal is expected to go high at the beginning of the mains period and the go low at the next zero-cross, as depicted in figure below. Since this is not usually the case, because of input threshold level the duty cycle, the rising edge is delayed, and the falling edge is too early. This results in a non-50/50 duty cycle, which again will result in a DC voltage over the Triac load. Use the parameters wCorrection and bCorPrescale to correct the duty-cycle, and thereby to get rid of the DC voltage. Setting these parameters will “delay” the falling edge in the Triac controller, as depicted in figure below.

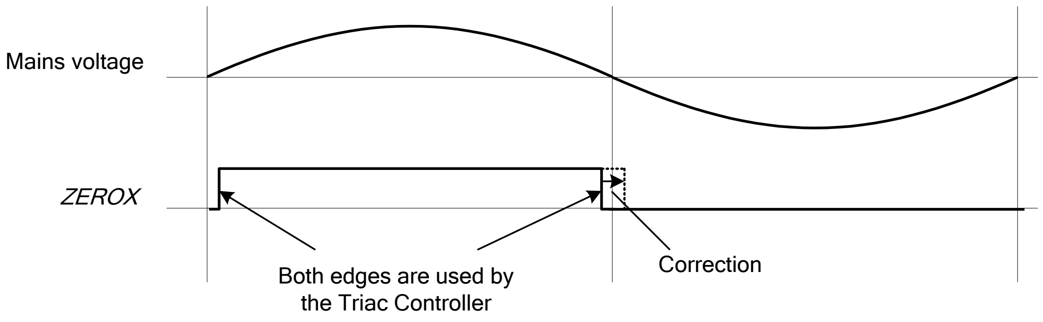

Figure 18. Half-bridge A zero-x signal

Half bridge B:

In this mode, the Triac Controller only uses one edge on the zero-cross signal for each period of the mains signal. That is, the zero-cross signal is expected to go high at the beginning of the mains period and the go low before the beginning of the next period, as depicted in figure below.

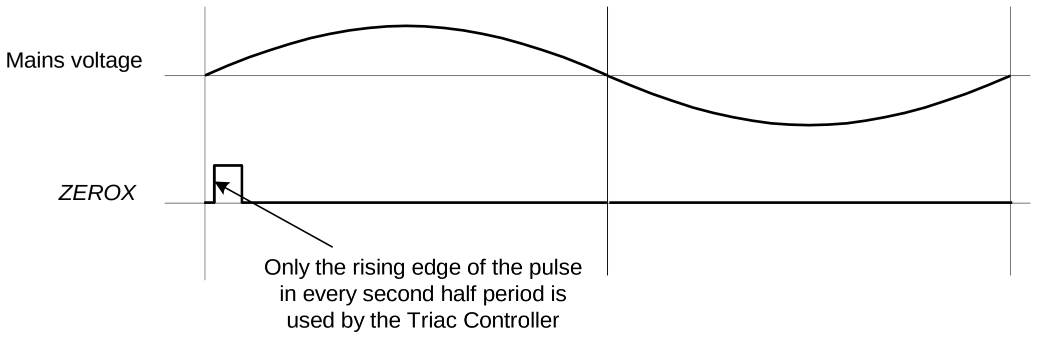
Mains voltage
ZEROX
Only the rising edge of the pulse in every second half period is used by the Triac Controller

Figure 19. Half-bridge B zero-x signal

In the positive halfperiod the triac pulse is generated

Full Bridge:

In this mode, the Triac Controller uses two rising edges on the zero-cross signal for each period of the mains signal. That is, the zero-cross signal is expected to go high at the beginning of the mains period and the go low before the beginning of the next half-period, then high again after the following zero-cross, and finally low again before the end of the period, as depicted in the two figures below.

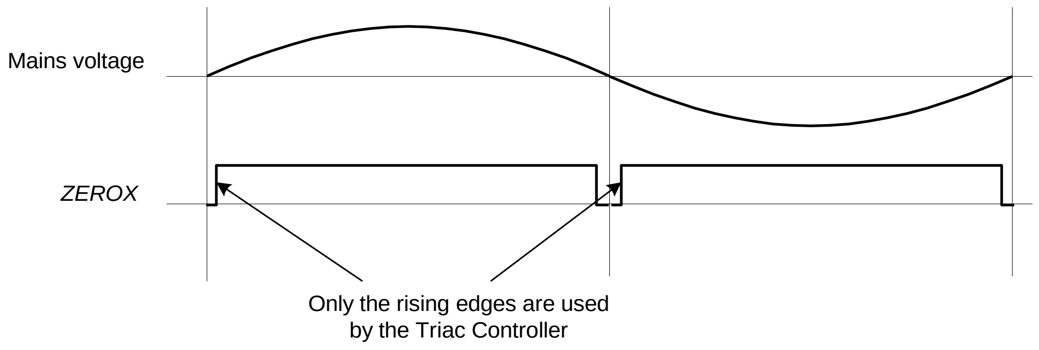

Figure 20. Example 1 of a full bridge zero-x signal

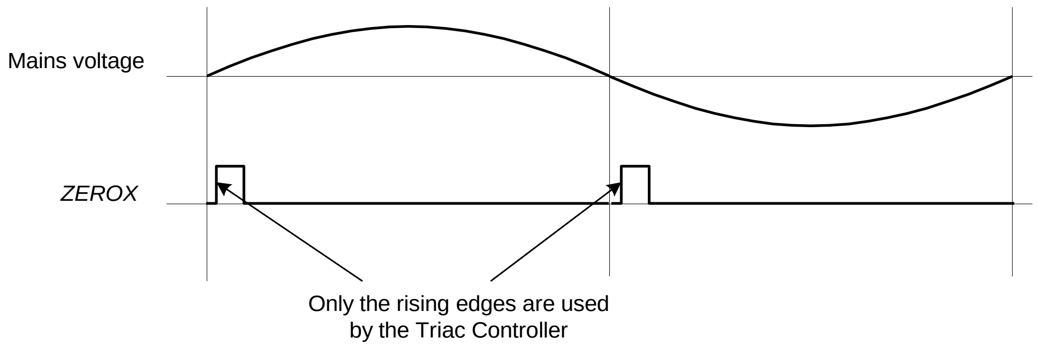

Figure 21. Example 2 of a full bridge zero-x signal

Once the Triac Controller is started, the Zero-cross signal is masked off the whole half period, except for a short period just before the next zero-x. This period can be adjusted using the parameter bKeepOff. See figure below

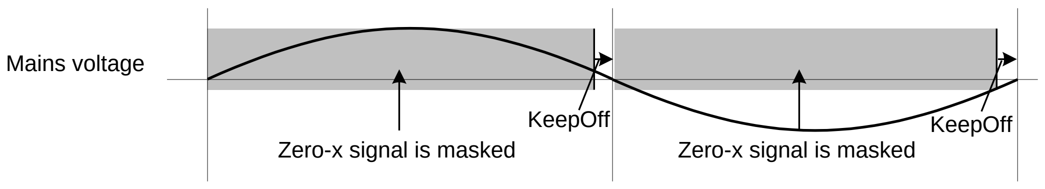

Figure 22. Masked Zero-X signal

In Triac Mode the Triac Controller will generate multiple pulses if the fire angle is less than 90º. The length of each of the pulses is set by the parameter wPulseLength and the repetition length is set by the parameter wPulseReplength. See figure below.

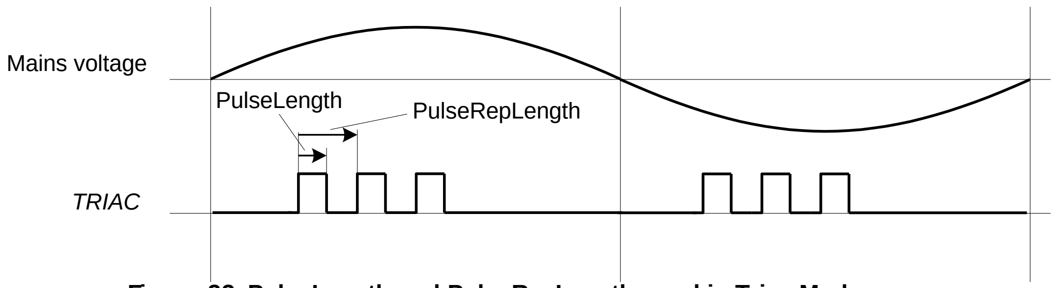

Figure 23. PulseLength and PulseRepLength used in Triac Mode

5.4.15.2 ZW_TRIAC_enable

void ZW_TRIAC_enable(BOOL boEnable)

ZW_TRIAC_enable enables/disables the Triac Controller. When enabled the Triac controller takes control over the TRIAC (P3.6) and the ZEROX[††††] (P3.7) pins. **ZW_TRIAC_init** must have been called before the Triac Controller is enabled.

Defined in: ZW_triac_api.h

Parameters:

boEnable	IN	TRUE or FALSE	TRUE: enables the Triac Controller

Serial API (Not supported)

[††††] If the PWM is enabled, see ZW_PWM_enable(), then the PWM will control the P3.7 pin

5.4.15.3 ZW_TRIAC_dimlevel_set

BOOL ZW_TRIAC_dimlevel_set(WORD wLevel)

ZW_TRIAC_dimlevel_set turns the Triac controller on and sets the dimming level. **ZW_TRIAC_init** must have been called before the Triac Controller is started.

Defined in: ZW_triac_api.h

Parameters:

wLevel IN	Dimming level (0-1000), where 0 is shut off and 1000 is full on

Return values:

BOOL	TRUE	The new dim level has been accepted by the Triac Controller
	FALSE	The Triac Controller has not yet read in the previous dim level. Wait up to one half period of the mains signal (50Hz: 10ms, 60 Hz 8.33ms) and try again

Serial API (Not supported)

5.4.15.4 ZW_TRIAC_int_enable

void ZW_TRIAC_int_enable(BOOL boEnable)

ZW_TRIAC_int_enable enables/disables the zero cross (ZeroX) interrupt. The ZeroX interrupt is issued when the TRIAC controller detects a zero cross on the ZEROX signal. Hence, the Triac Controller will take control of the ZEROX pin (P3.7) when **ZW_TRIAC_int_enable(TRUE)** has been called.

The ZeroX interrupt can be used to implement a SW based TRIAC controller where the TRIAC signal is controlled by the SW. The Triac Controller will generate the ZeroX interrupt when it detects a zero cross on the ZEROX signal, even if the Triac Controller has been disabled (by calling **ZW_TRIAC_enable(FALSE)**) as long as **ZW_TRIAC_int_enable(TRUE)** has been called.

Defined in: ZW_triac_api.h

Parameters:

boEnable IN	TRUE	Enable the interrupt. The Triac controller will issue an interrupt when a zero cross is detected on the ZEROX signal.
	FALSE	Disable the Triac interrupt.

Serial API (Not supported)

The interrupt number is set by the define, INUM_TRIAC, as described in ZW040x.h

5.4.15.5 ZW_TRIAC_int_get

BOOL ZW_TRIAC_int_get(void)

ZW_TRIAC_int_get returns the state of the Triac Controller interrupt flag. Call **ZW_TRIAC_int_enable(TRUE)** to enable the interrupt.

Defined in: ZW_triac_api.h

Return values:

BOOL	TRUE	The Triac Controller interrupt flag is set.
	FALSE	The Triac Controller interrupt flag is cleared.

Serial API (Not supported)

5.4.15.6 ZW_TRIAC_int_clear

void ZW_TRIAC_int_clear(void)

ZW_TRIAC_int_get clears the Triac Controller interrupt flag. Call **ZW_TRIAC_int_enable(TRUE)** to enable the interrupt and use **ZW_TRIAC_int_get** to see whether the interrupt has been set.

Defined in: ZW_triac_api.h

Serial API (Not supported)

5.4.16 LED Controller API

The built-in LED Controller is targeted at LED light dimming applications. The controller can control up to four individual channels in 3 different modes of operation. The application software can use the following LED API calls to control the 400 Series Z-Wave Single Chip LED Controller.

5.4.16.1 ZW_LED_init

void ZW_LED_init(BYTE bMode,
BYTE bChannelEn)

ZW_LED_init initializes the 400 Series Z-Wave Single Chip's integrated LED controller by setting the desired mode of operation and the desired number of active channels. Should be called in **ApplicationHWInit**.

Defined in: ZW_led_api.h

Parameters:

bMode IN	Mode of operation type:	
	LED_MODE_NORMAL	In this mode, the LED controller will generate a pulse width modulated signal for each active channel. The PWM signals has no phase skew. The frequency of all of the PWM signals is $32MHz/2^{16}$ = 488.28Hz. The duty-cycle of the PWM signals is set by the **ZW_LED_waveforms_set** function.
	LED_MODE_SKEW	The SKEW mode is same as the NORMAL mode except that phase of the channels are skewed. That is, the signal of channel 1 is skewed ¼ of a period compared to the signal of channel 0, the signal of channel 2 is skewed ¼ of a period compared to the signal of channel 1, etc.
	LED_MODE_PRBS	In this mode, the LED controller uses a PRBS signal generator to generate to LED signals. The total high time in this mode equals the total high time in the other modes.

bChannelEn	IN	Bit mask of one of the 4 channels to be enabled	
		LED_CHANNEL0	Enable channel 0. The LED Controller takes control of the P0.4 pin.
		LED_CHANNEL1	Enable channel 1. The LED Controller takes control of the P0.5 pin.
		LED_CHANNEL2	Enable channel 2. The LED Controller takes control of the P0.6 pin.
		LED_CHANNEL3	Enable channel 3 The LED Controller takes control of the P0.7 pin.

Serial API (Not supported)

5.4.16.2 ZW_LED_waveforms_set

void ZW_LED_waveforms_set(WORD *pwLevel)

ZW_LED_waveforms_set sets the high time of signals of each of the 4 LED Controller channels. This API will set the waveform for all channels even though not all of them are enabled.
ZW_LED_waveforms_set waits until the previous waveform setting has been adopted by the LED controller before setting the new waveform values. This can take up to 2.048ms. Alternatively, first poll **ZW_LED_busy** to see when the LED controller is ready to adopt new settings and then call **ZW_LED_waveforms_set** or call **ZW_LED_waveform** to set one channel at a time.

Defined in: ZW_led_api.h

Parameters:

pwLevel IN	A pointer to an array with 4 16-bits values.	
	0x0000-0xFFFF	Duty cycle times of the LED controller channels. The first 16 bit element in the array determines the value for channel 0. The next 16 bit element determines the value for channel 1, etc.

Serial API (Not supported)

5.4.16.3 ZW_LED_waveform_set

void ZW_LED_waveform_set(BYTE bChannel
WORD wLevel)

ZW_LED_waveform_set: set the duty cycle time of one of the LED controller 4 channels. The API will not wait until the LED controller is ready to use the new value. The LED controller can only accept the new value just after the end of a period. Poll **ZW_LED_busy** to see when the LED controller is ready to adopt new settings.

Alternatively, call **ZW_LED_waveforms** to set all channels in one call.

Defined in: ZW_led_api.h

Parameters:

bChannel IN	The channel ID LED_CHANNEL0 LED_CHANNEL1 LED_CHANNEL2 LED_CHANNEL3
wLevel	The duty cycle of the channel 0x0000-0xFFFF

Serial API (Not supported)

5.4.16.4 ZW_LED_busy

BOOL ZW_LED_busy(void)

ZW_LED_busy is used to check to see if the LED controller is ready to accept new waveform values.

Defined in: ZW_led_api.h

Return values:

BOOL	TRUE	The LED controller can accept new waveform values
	FALSE	The LED controller cannot accept new waveform values, since it has not yet read in the previous data set. Wait up to 2^{16}/32MHz = 2.048ms and check again.

Serial API (Not supported)

5.4.17 Infrared Controller API

The built-in Infrared (IR) Controller is targeted at IR remote applications. The IR controller can operate either as an IR transmitter or as an IR receiver. When operating as a transmitter one or more of the three outputs (P3.4, P3.5, and P3.6) can be enabled as IR outputs that drive an IR LED, as depicted in figure below. Each output can drive 12mA. Hence, using three outputs give a drive strength of 36mA. If 36mA is insufficient you will have to implement an external driver.

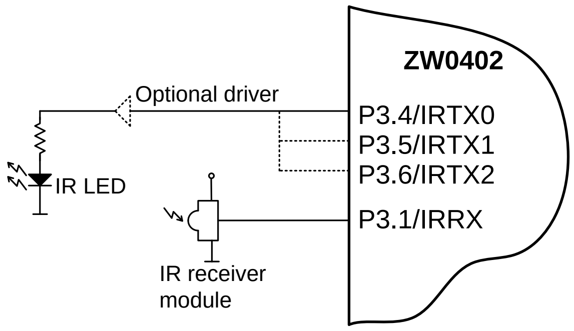

Figure 24. External IR hardware

An external IR receiver module or an IR transistor must be connected to Pin P3.1 when operating in Receive mode. An IR receiver module has a built-in photo transistor and preamplifier with automatic gain control and gives a digital TTL/CMOS output signal. The IR receivers can be found in two versions, with and without demodulator. The versions without demodulator (like Vishay TSOP 98200) generates an output signal with carrier (as depicted in the upper part of figure below), whereas the versions with demodulator (like Vishay TSOP322xx) generates an output signal without the carrier (as depicted in the lower part of figure below). Therefore, the one without demodulator is best for code learning applications, where you want to be able to detect the carrier frequency. The one with modulator has improved immunity against ambient light such as fluorescent lamps.

Using an photo transistor, where the transistor is connected directly to the 400 Series Z-Wave Chip requires that the transmitting IR LED is placed within a short range (2"-4") of the IR transistor, since the IR transistor signal is analog and isn't amplified. This circuit is also sensitive also to ambient light.

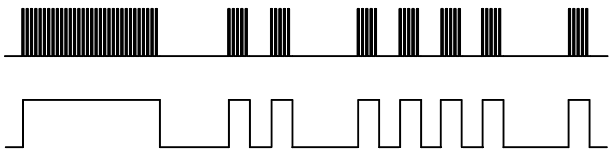

Figure 25. IR signal with and without carrier

In both cases, the IR Receiver detects widths of the marks (high/carrier on) and spaces (low) of a coded IR message, as seen in figure below. The mark/space width data is stored in SRAM using DMA. While running, the IR Controller requires very little CPU processing. The IR receiver is able to detect the waveform of the carrier[‡‡‡‡].

The IR Transmitter generates a carrier and the marks and spaces for an IR message. The widths of the marks and the spaces are read from SRAM using DMA.

[‡‡‡‡] Note that the IR receiver module can distort the duty cycle of the carrier.

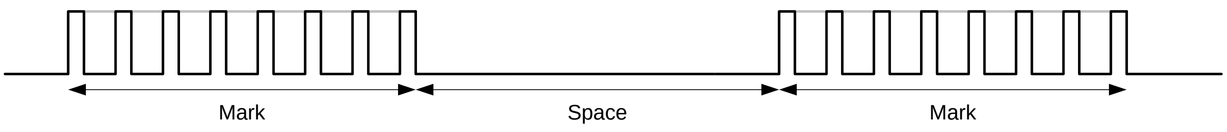

Figure 26. IR Coded message with carrier

Both the IR Receiver and the IR Transmitter can be configured to detect/generate a wide range of IR coding formats.

5.4.17.1 Carrier Detector/Generator

The carrier detector can detect the carrier waveform (high and low periods) of a carrier modulated IR signal.

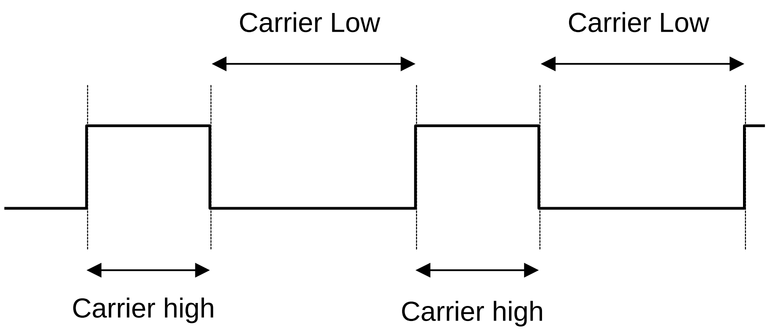

Figure 27. Carrier waveform

The following bullets provide a short feature list of the Carrier Detector/generator.

1. IR Carrier Generator frequency range: 7.8kHz - 16MHz (50/50 duty cycle) or 10.4kHz -10.7MHz (33/66 duty cycle)
2. A built-in Glitch Remover is able to remove glitches on the incoming IR signal.
3. For each detection process the IR Carrier Detector can calculate an average of the “high” duration and an average of the “low” duration over 1 (no averaging), 2, 4, or 8 periods.

5.4.17.2 Organization of Mark/Space data in Memory

Both the IR Receiver engine and the IR Transmitter engine use SRAM to store mark/space data information. The data is stored in the same format for both engines, as depicted in figure below. The width of a mark is stored in 1 to 3 bytes – likewise the width of a space is stored in 1-3 bytes.

Bit 7 in each byte is used to differentiate the mark and space bytes. That is, bit 7 of all “mark”-bytes are set to 1 and bit 7 of all “space”-bytes are set to 0.

The maximum number of bits used to describe a mark or space width is 16. The means that 3 bytes are needed to store a 16 bit value (the upper 6 bits of the 3rd byte are unused); whereas 2 bytes are needed to store a 14 bit value, and only one byte is needed to store a 7 bits value.

Refer to the example as depicted in figure below, where:

4. 3 bytes are used for the start mark (PS0)
5. 3 bytes are used for the start space (PS1)
6. 2 bytes are used for each of the rest of the mark/spaces

The maximum size of the mark/space data information is 511 bytes. The data can be stored anywhere in the lower 4K XRAM in the 400 series Z-Wave Single Chip.

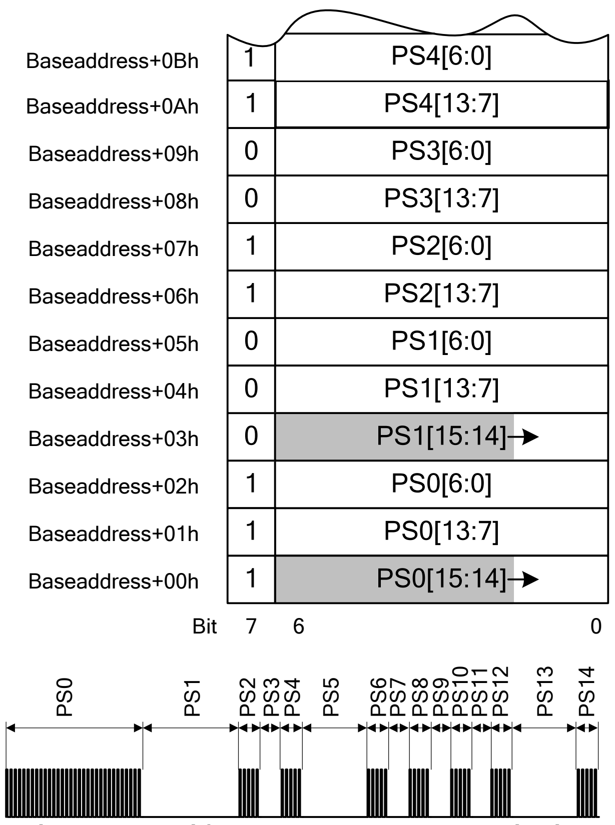

Figure 28. Mark/Space Data Memory Organization

The width is described as a certain count of prescaled clock periods. E.g if the prescaler is set to 1/16 and the width of a mark is 889us, the width will be stored as

$$\frac{period}{\frac{prescaler}{f_{sys}}} = \frac{889us}{\frac{16}{32MHz}} = 1798LSB$$

That is,

$$\frac{\ln(1798)}{\ln(2)} = 10.81$$

11 bits are needed $\Rightarrow$ 2 bytes.

Since the maximum number of bits used to store each mark or space width is 16. It results in a maximum mark or space width of:

- 262ms using clock divider of 128 or
- 1.7s using the Carrier Generator @ 36kHz

5.4.17.3 IR Transmitter

Before the IR transmitter can start generating the IR stream the IR Transmitter must have been initialized, the Mark/Space data must have been built in a buffer in the lower 4kB XRAM, and the IR interrupt is optionally enabled. The organization of the Mark/Space data is described in section 5.4.17.2. Additionally, the Carrier Generator must be initialized.

The function **ZW_IR_tx_init** must be called to initialize all the needed parameters as described below and in section 5.4.17.5:

7. The prescaler value for generating the carrier signal can be either: 1, 2, 3, 4, 5, 6, 7, or 8 resulting in a clock speed of either 32MHz, 16MHz, 32/3MHz, 8MHz, 32/5MHz, 16/3MHz, 32/7MHz, or 4MHz
8. The IR transmitter can use a prescaler that use the 32MHz clock divided by 1, 2, 4, 8, 16, 32, 64, or 128. It can also use the rising edge of the carrier generated by the Carrier Generator.
9. The output(s) can be inverted as an option
10. The Idle state of the IR signal can be either high or low
11. One, two, or three IO's can be used in parallel for driving an IR LED. Each output buffer can drive 12mA.
12. The carrier wave form is set by the carrier prescaler and two parameters that sets the low and high period of the carrier signal.

If only one IR coding style is used in a application the **ZW_IR_tx_init** function can be placed in **ApplicationInitHW,** otherwise it can be placed in other parts of the code, typically in **ApplicationPoll**

The function **ZW_IR_tx_data** must be called when a certain IR stream is to be transmitted. The parameters for this function sets is described below and in section 5.4.17.6

13. The address of the buffer in lower 4kB XRAM.
14. Size of IR data buffer in XRAM. The maximum size of the XRAM buffer is 511 bytes.

The IR Transmitter takes over control of the enabled IO's (P3.4, P3.5, and/or P3.6) when the function **ZW_IR_tx_data** is called and releases the control of the enabled IO's when the IR signal has been transmitted. Therefore, to make sure that IO's used by the IR transmitter (P3.4, P3.5, and/or P3.6) are output(s) and at the correct idle state, the GPIO must be set as outputs and the state must be set accordingly.

An IR interrupt routine is supplied with the ZW_phy_infrared_040x library. A variable `ir_tx_flag` (BOOL) is set TRUE when an IR message has been transmitted after **ZW_IR_tx_data** has been called. The `ir_tx_flag` variable is cleared when calling **ZW_IR_tx_data**.

Once the IR Transmitter is started, use the function **ZW_IR_disable** to cancel the operation.

An example of how to initialize and run the IR Transmitter is shown in Figure 29.

```
void ApplicationInitHW()
{
  EA=1;
  EIR=1;
         :
   /* Carrier freq = 8MHz/(74+148)=36kHz, Carrier duty cycle 33/66 */
   ZW_IR_tx_init(FALSE, // Use Mark/Space prescaler
              3,     // Prescaler: 32MHz/(2^3)=4MHz
              FALSE, // Output is not inverted
              FALSE, // Output state is low
              0x03,  // Enable P3.4 and P3.5
              3,     // Carrier prescaler set to 4 (32MHz/4=8MHz)
              74,    // Carrier low 74/8MHz = 9.25us
              148);  // Carrier high 148/8MHz = 18.5us
         :
}

void ApplicationPoll()
{
  BYTE bIrBuffer[16];

   switch (mainState)
   {
           :
     case SEND_PLAY:
       // This IR message send a "PLAY" command
       bIrBuffer[0]=0xA0;
       bIrBuffer[1]=0x20;
       bIrBuffer[2]=0xBF;
       bIrBuffer[3]=0x20;
       bIrBuffer[4]=0xA0;
       bIrBuffer[5]=0x20;
       bIrBuffer[6]=0xA0;
       bIrBuffer[7]=0x20;
       bIrBuffer[8]=0xA0;
       bIrBuffer[9]=0x20;
       bIrBuffer[10]=0xA0;
       bIrBuffer[11]=0x20;
       bIrBuffer[12]=0xBF;
       bIrBuffer[13]=0x20;
       bIrBuffer[14]=0xDF;
       bIrBuffer[15]=0x80;
       ZW_IR_status_clear();          // Clear all IR status flags
```

```
        /* Use IrBuffer as buffer, size 16 bytes */
        ZW_IR_tx_data((WORD)bIrBuffer, // Address of buffer
                       16);            // Size of buffer
        mainState=WAIT_IR_DONE;
        break;
      case WAIT_IR_DONE:
        if (ir_tx_flag==TRUE)       // Wait until IR TX flag is set
        {
          mainState=IDLE;
        }

        break;
    }
}
```

Figure 29. Code example on use of IR transmitter

5.4.17.4 IR Receiver

Before the IR Receiver can be used to learn an incoming IR stream the IR receiver must have been initialized, the Mark/Space data buffer must have been allocated in the lower 4kB XRAM, and the IR interrupt *must* be enabled.

The organization of the Mark/Space data is described in section 5.4.17.2.

The function **ZW_IR_learn_init** must be called to initialize all the needed parameters as described below and in section 5.4.17.5:

15. The Rx SRAM buffer size is configurable. (1-511 bytes)
16. The Mark/Space detector in the IR Receiver can use either a prescaler that use the 32MHz clock divided by 1, 2, 4, 8, 16, 32, 64, or 128.
17. If the IR Receiver requires more SRAM space for the incoming IR stream, the CPU is interrupted and an error flag is set
18. The IR Receiver can be configured to remove glitches on the incoming IR signal
19. The IR Receiver can be configured to average the detected duration of the low/high periods of the Carrier
20. The IR input signal can be inverted as an option.
21. The IR Receiver can detect that the trailing space after the last mark of a received IR message is longer that a specific size. This size must be set and this works at the same time as a timeout if the message for some reason is shorter than expected.

Call the function **ZW_IR_learn_data** to start the learn process. The function is described below and in section 5.4.17.9:

22. When the learn process starts the IR receiver will start out using the highest possible prescaler value for the Carrier detector. When it then detects a carrier, it will measure the duration of the low and high periods of the carrier and, if possible, rescale the prescaler to a lower value and rerun the carrier measurement. This is done to achieve the highest precision of the carrier measurement while preventing timer overflow.
23. The learn process will terminate when the IR Receiver has detected at least one Mark and then a Space larger than a configurable amount of time, as described above.

An IR interrupt routine is supplied with the ZW_phy_infrared_040x library. A variable `ir_rx_flag` (BOOL) is set TRUE when an IR message has been received. The `ir_rx_flag` variable is cleared when calling **ZW_IR_learn_data.**

Call the function **ZW_IR_rx_status_get** to get the size of the received mark/space data, the detected carrier characteristics and error state (status flags). The function **ZW_IR_status_clear** clears the status flag.

Once the IR Receiver is started, use the function **ZW_IR_disable** to cancel the operation.

An example of how to initialize and run the IR Receiver is shown below.

```
BYTE bIrBuffer[256];

void ApplicationInitHW()
{

  EA=1;
  EIR=1;

  ZW_IR_learn_init((WORD)bIrBuffer // Buffer address
                    256,           // Buffer Size
                    4,             // Prescaler: 32MHz/(2^4)=2MHz
                    7,             // Trailing space min 2^16/2MHz=32.8ms
                    2,             // Run average over 4 periods
                    1,             // Remove glitches below 125ns
                    FALSE);        // Do not invert input
}

void ApplicationPoll()
{
  WORD wRxDataLen;
  BYTE bRxCarrierLow;
  BYTE bRxCarrierHigh;
  BYTE bRxStatus;
      :
  switch(mainState)
  {
    case START_IR_LEARN:
      ZW_IR_status_clear();      // Clear all IR status flags
      ZW_IR_learn_data();        // Start IR Receiver
      mainState=WAIT_IR_DONE;
      break;
    case WAIT_IR_DONE:
      if (ir_rx_flag==TRUE)  // Wait until IR RX flag is set
      {
        ZW_IR_rx_status_get( &wRxDataLen,
                             &bCarrierPrescaler,
                             &bRxCarrierLow,
                             &bRxCarrierHigh,
                             &bRxStatus);
```

```
          if (bRxStatus == 0x00 || bRxStatus == IRSTAT_CDONE)
          {
            if (bRxStatus == IRSTAT_CDONE)
            {
              /* report that carrier could not be detected */
                      :
            }
            /* decode received data */
                    :
          }
          else
          {
            /* error handling */
              :
          }
          mainState=IDLE;
        }
        else
        {
        /* Cancel the operation when boCancelIR is set
           TRUE by other part of the code */
          if (boCancelIR == TRUE) ZW_IR_disable();
          mainState=IDLE;
        }
        break;
    }
}
```

Figure 30. Code example on use of IR receiver

The application software can use the following IR API calls to control the 400 Series Z-Wave Single Chip IR Controller.

5.4.17.5 ZW_IR_tx_init

```
void ZW_IR_tx_init( BOOL boMSTimer,
                    BYTE bMSPrescaler,
                    BOOL boInvertOutput,
                    BOOL boHighDrive,
                    BOOL boIdleState,
                    BYTE bOutputEnable,
                    BYTE bCarrierPrescaler,
                    BYTE bCarrierLow,
                    BYTE bCarrierHigh)
```

ZW_IR_tx_init initializes the 400 Series Z-Wave Single Chip's integrated IR controller to Transmitter mode and sets the required TX options.

Defined in: ZW_infrared_api.h

Parameters:

boMSTimer IN	TX Mark/Space prescaler mode:	
	TRUE	Mark/space generator runs on carrier period timer. That is, the length of the Marks/Spaces is calculated as the carrier period multiplied by the value read in XRAM
	FALSE	Mark/space generator runs on a prescaled timer. That is, the length of the Marks/Spaces is calculated as the prescaled timer period multiplied by the value read in XRAM. Prescaler value is set by bMSPrescaler.
bMSPrescaler IN	Mark/Space timer prescaler	
	Valid values: 0-7 Resulting timer clock frequency: 0: 32MHz 1: 16MHz 2: 8MHz 3: 4MHz 4: 2MHz 5: 1MHz 6: 500kHz 7: 250kHz	Not applicable when boMSTimer is true
boInvertOutput IN	Invert IR output	
	TRUE	output is inverted
	FALSE	output is not inverted
boHighDrive	Invert IR output	
	TRUE	use 12mA drive strength of IR Tx IO output buffers
	FALSE	use 8mA drive strength of IR Tx IO out buffers
boIdleState IN	Idle State of IR output	
	TRUE	Idle state is high
	FALSE	Idle state is low
bOutputEnable IN	Outputs enabled	
	Valid values: 0-7 000: All outputs disabled xx1: P3.4 enabled x1x: P3.5 enabled 1xx: P3.6 enabled	Idle state is high

bCarrierPrescaler IN	Carrier generator prescaler Valid values: 0-7 Resulting timer clock frequency: 0: 32MHz 1: 32MHz/2 2: 32MHz/3 3: 32MHz/4 4: 32MHz/5 5: 32MHz/6 6: 32MHz/7 7: 32MHz/8
bCarrierLow IN	Carrier low time 0: 1 prescaled clock period 1: 2 prescaled clock periods : : 255: 256 prescaled clock periods
bCarrierHigh IN	Carrier High time 0: 1 prescaled clock period 1: 2 prescaled clock periods : : 255: 256 prescaled clock periods

Serial API (Not supported)

5.4.17.6 ZW_IR_tx_data

void ZW_IR_tx_data(WORD pBufferAddress,
WORD wBufferLength)

ZW_IR_tx_data sets the address and the length of the buffer containing the Mark/space data to be sent. The IR Controller will start to transmit immediately after these values have been set.

Defined in: ZW_infrared_api.h

Parameters:

pBufferAddress	IN	Address of Tx buffer in lower XRAM memory
wBufferLength	IN	Number of bytes in TX buffer. Valid values (1-511)

Serial API (Not supported)

5.4.17.7 ZW_IR_tx_status_get

BYTE ZW_IR_tx_status_get(void)

ZW_IR_tx_status_get is used to check to the status of the IR controller after an IR message has been transmitted.

Defined in: ZW_infrared_api.h

Return values:

IRSTAT_MSOVERFLOW	The format of the data in the IR buffer is invalid. The perceived Mark/Space value is greater than 2^{16}.
IRSTAT_PSSTARV	The IR controller's DMA engine was not able to read data from XRAM in time because the access to the XRAM was used by (an) other DMA engine(s) with higher priority. To get rid of this error, try to disable other DMA engines (USB, RF, etc.) and run the IR transmitter again.

Serial API (Not supported)

5.4.17.8 ZW_IR_learn_init

```
void ZW_IR_learn_init(  WORD pBufferAddress,
                        WORD wBufferLen,
                        BYTE bMSPrescaler,
                        BYTE bTrailSpace,
                        BYTE bCAverager,
                        BYTE bCGlitchRemover,
                        BOOL boInvertInput)
```

ZW_IR_learn_init initializes the 400 Series Z-Wave Single Chip's integrated IR controller to receive/learn mode and sets the required RX options.

Defined in: ZW_infrared_api.h

Parameters:

pBufferAddress IN	Address of Rx buffer in lower XRAM memory
wBufferLength IN	Size of RX buffer. Valid values (1-511)
bMSPrescaler IN	Mark/Space timer prescaler Valid values: 0-7 Resulting timer clock frequency: 0: 32MHz 1: 16MHz 2: 8MHz 3: 4MHz 4: 2MHz 5: 1MHz 6: 500kHz 7: 250kHz
bTrailSpace IN	Trailing space after last Mark. After the incoming IR signal has been low for this period of time the IR receiver stops. Valid values: 0-7 0: 512 prescaled clock periods 1: 1024 prescaled clock periods 2: 2048 prescaled clock periods 3: 4096 prescaled clock periods 4: 8192 prescaled clock periods 5: 16384 prescaled clock periods 6: 32768 prescaled clock periods 7: 65536 prescaled clock periods
bCAverager IN	Average Carrier high/low length measurement over multiple carrier periods. Valid values: 0-3 0: 1 carrier period 1: 2 carrier periods 2: 4 carrier periods 3: 8 carrier periods (Recommended value)
bCGlitchRemover IN	Remove glitches from incoming IR signal. Valid values: 0-3 0: disabled 1: < 125ns 2: < 250ns 3: < 500ns

boInvertInput IN	TRUE	IR input is inverted
	FALSE	IR input is not inverted

Serial API (Not supported)

5.4.17.9 ZW_IR_learn_data

void ZW_IR_learn_data(void)

ZW_IR_learn_data clears the `ir_rx_flag` variable and starts the IR Controller in Rx/learn mode. Use **ZW_IR_disable** to cancel on ongoing learn process.

Defined in:	ZW_infrared_api.h

Serial API (Not supported)

Refer to section 0 for a detailed description of the learn function.

5.4.17.10 ZW_IR_rx_status_get

```
void ZW_IR_rx_status_get(WORD  *wDataLength,
                         BYTE *bCarrierPrescaler,
                         BYTE *bCarrierLow,
                         BYTE *bCarrierHigh,
                         BYTE *bStatus)
```

ZW_IR_rx_status_get is used to check to the status of the IR controller, to get the length of the received data, the detected carrier characteristics, and the error state. Call this function after a learn operation is done, i.e. after the `ir_rx_flag` variable has been set.

Defined in: ZW_infrared_api.h

Parameters:

wDataLength OUT	Length of the received data	
bCarrierPrescaler OUT	Optimal Carrier prescaler value Valid values: 0-7 0: 32MHz 1: 32MHz/2 2: 32MHz/3 3: 32MHz/4 4: 32MHz/5 5: 32MHz/6 6: 32MHz/7 7: 32MHz/8	
bCarrierLow OUT	Length of the Low period of the Carrier (in prescaled system clocks)	
bCarrierHigh OUT	Length of the High period of the Carrier (in prescaled system clocks)	
bStatus OUT	IRSTAT_PSSTARV	The IR controller's DMA engine was not able to write data from XRAM in time because the access to the XRAM was used by (an) other DMA engine(s) with higher priority. To get rid of this error, try to disable other DMA engines (USB, RF, etc.) and run the IR transmitter again.
	IRSTAT_MSOVERFLOW	The duration of a mark/space exceeded 2^{16} prescaled clock periods.
	IRSTAT_COF	The Carrier detector failed because the perceived carrier low/high period was too long.
	IRSTAT_CDONE	The Carrier detector completed measuring the carrier without errors
	IRSTAT_RXBUFOVERFLOW	The RX buffer was too small to store the received IR data

Serial API (Not supported)

5.4.17.11 ZW_IR_status_clear

void ZW_IR_status_clear(void)

ZW_IR_status_clear clears the Tx and Rx IR Status flags. Call this function before rerunning the IR Controller.

Defined in:	ZW_infrared_api.h

Serial API (Not supported)

5.4.17.12 ZW_IR_disable

void ZW_IR_disable(void)

This function disables any ongoing IR operation and sets the IR Controller to its idle state. Use **ZW_IR_status_clear** to clear any status bit before starting the IR Transmitter or IR Receiver.

Defined in:	ZW_infrared_api.h

Serial API (Not supported)

5.4.18 Keypad Scanner Controller API

The built-in hardware keypad scanner is able to scan a matrix of up to 8 rows x 16 columns. When the Keypad Scanner is activated, the 8 row inputs (P1.0-P1.7) must either be connected to the hardware key matrix or kept open. The number of columns can be configured to the range 1-16. The actual IO's being used as column outputs are "KSCOL0" (P0.0) when the column count is set to one, "KSCOL0, KSCOL1" (P0.0, P0.1) when the column count is set to two, "KSCOL0, KSCOL1, KSCOL2" (P0.0, P0.1, P0.2) when the column count is set to three, etc. A column output can be left open, though.

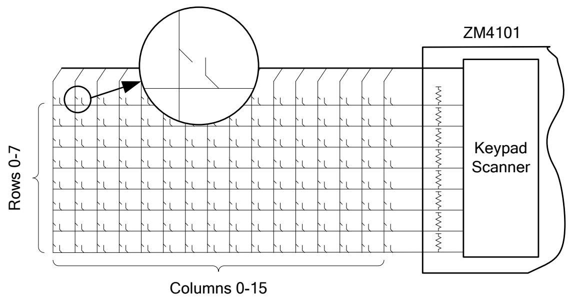

Figure 31. Keypad matix

Once the Keypad Scanner is enabled, it will scan each column for an amount of time (Scan Delay). If it, at a certain column detects a key press, it will wait for a period (Debounce delay) to get any eventual debounce noise to disappear. Then it will detect whether the input stays stable for another amount of time (Stable delay). The Keypad Scanner will issue an interrupt request to the CPU, if the row input is stable for the defined amount of "Stable delay" time. See figure below.

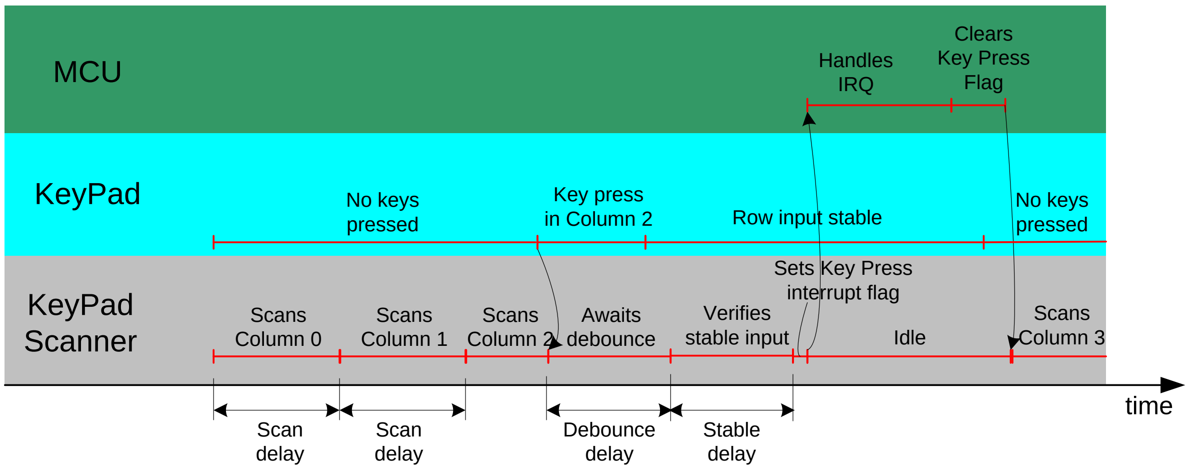

Figure 32. Scan flow

Each of these delays can be configured by using the function **ZW_KS_init,** which must be called in **ApplicationInitHW** as shown in figure below.

```
void KeyPadChanges(BYTE_P pbKeypadMatrix, BYTE bStatus)
{
  /* Call the user defined application function
     InterpretKeys with the keypad matrix as parameter */
  switch (bStatus)
  {
    case ZW_KS_KEYPRESS_VALID:
       InterpretKeys(keyPadMatrix);
       break;
     case ZW_KS_KEYPRESS_INVALID:
       beep();
       break;
     case ZW_KS_KEYPRESS_RELEASED:
       cleanup();
       break
     default:
  }
}


void ApplicationInitHW(BYTE bWakeupReason)
{
          :
     ZW_KS_init(7,      /* 8 Columns */
                4,      /* Column scan delay 10ms */
                15,     /* Debounce delay 32ms */
                5,      /* Row Stable delay 12ms */
                10,     /* Polling period of 100 ms*/
                KeyPadChanges   /* The callback function used to notify */
                                /* the application when changes occurs */
                                /* to the keypad matrix */
                )
     ZW_KS_enable(TRUE);
           :
}

void ApplicationPoll()
{
           :
     /* Go into power down mode */
     ZW_KS_pd_enable(TRUE);
     ZW_SetSleepMode(WUT_MODE,ZW_INT_MASK_EXT1,0);
           :
}
```

Figure 33. Example of the API calls for the KeyPad scanner

The Keypad ISR will detect any changes occurred to the keypad matrix. The changes to the keypad matrix array will be polled periodically. The polling period is defined by the application through **ZW_KS_init**.

Apart from setting the size of the key matrix and the delays, a callback function must be defined in **ZW_KS_init**. If any changes to the keypad matrix are detected the application will be notified by calling this user defined callback function. Figure above shows an example of how **ZW_KS_init** is used.

The parameter to the callback function is an array of the type BYTE. The array has 16 elements one for each column. It has 16 elements regardless of the number of actual configured columns in use. The element with index `n` holds the row-status for column number `n`. That is, bit 0 of an element hold the status of row 0, bit 1 of an element hold the status of row 1, etc. The array is defined in the Keypad API library.

Note: the Keypad Scanner IRQ signal is shared with "EXT1", external interrupt 1. Therefore, that interrupt routine must not be included in the application code, when using the Keypad Scanner.

When a key press must wake up the 400 Series Z-Wave Single Chip from powerdown mode, **ZW_KS_pd_enable(TRUE**) must be called just before the chip is put into powerdown mode. Doing so, will activate the external interrupt, if any key is pressed. When the 400 Series Z-Wave Single Chip is awake first the function **ZW_KS_init** and then **ZW_KS_enabled** must be called to initialize and enable the Key Scanner and thereby grab the actual key combination.

5.4.18.1 ZW_KS_init

```
void ZW_KS_init( BYTE bCols
                 BYTE bScanDelay
                 BYTE bDebounceDelay,
                 BYTE bStableDelay
                 BYTE bReportWaitTimeout,
                 VOID_CALLBACKFUNC(KeyPadCallBack)(BYTE_P keyMatrix, BYTE bStatus) )
```

ZW_KS_init initializes the 400 Series Z-Wave Single Chip's integrated Keypad Scanner.

Defined in: ZW_keypad_scanner_api.h

Parameters:

bCols IN	Sets the number of enabled columns. Valid values 0-15. 0: 1 Column 1: 2 Columns : 15: 16 Columns E.g. setting this to 7 will enable KSCOL0-KSCOL7 (P0.0-P0.7)
bScanDelay IN	Sets column “Scan delay” 0: 2ms 1: 4ms : 15: 32ms
bDebounceDelay IN	Sets “debounce delay” 0: 2ms 1: 4ms : 15: 32ms
bStableDelay IN	Sets “stable delay” 0: 2ms 1: 4ms : 15: 32ms
bReportWaitTimeout IN	Set the timeout delay before the main loop call the KeyPadCallBack function. 0: not valid 1: 10 ms 2: 20 ms : 255: 2550 ms

KeyPadCallBack IN	The call back function that the main loop will use to notify the application about the changes in the keypad matrix. The function will only be called when changes to the keypad matrix occurs. Parameters: pbKeyMatrix OUT: Pointer to the keypad matrix BYTE array. bStatus OUT: Returns the status of the contents of the key matrix as one of the follwing:	
	ZW_KS_KEYPRESS_VALID	The contents of the key matrix array is valid
	ZW_KS_KEYPRESS_INVALID	The contents of the key matrix array is invalid, i.e. more than 3 keys are pressed in an invalid manner
	ZW_KS_KEYPRESS_RELEASED	All keys have been released

Serial API (Not supported)

5.4.18.2 ZW_KS_enable

void ZW_KS_enable(BOOL boEnable)

ZW_KS_enable enables or disables the Keypad Scanner. Must be called in **ApplicationInitHW**.

Defined in: ZW_keypad_scanner_api.h

Parameters:

boEnable IN	TRUE	Enables Keypad Scanner
	FALSE	Disables Keypad Scanner

Serial API (Not supported)

5.4.18.3 ZW_KS_pd_enable

void ZW_KS_pd_enable(BOOL boEnable)

ZW_KS_ pd_enable(TRUE) must be called before putting the 400 Series Z-Wave Single Chip into powerdown mode, if the chip is to be woken by a key press.

Defined in: ZW_keypad_scanner_api.h

Parameters:

boEnable IN	TRUE	Enables Keypad Scanner Powerdown mode
	FALSE	Disables Keypad Scanner Powerdown mode

Serial API (Not supported)

5.4.19 USB Controller API

This section describes the 400 Series Z-Wave Single Chip built-in USB controller. The USB controller has 2 endpoints. Each endpoint have two directions OUT from the host to the device and IN from the device to the host. Endpoint 1 can sends/receives data up to 63 bytes per packet and endpoint 2 can sends/receives data up to 15 bytes per packet. The function **ZW_USB_ep1_write** / **ZW_USB_ep2_write** is used to send data to host. **ZW_USB_ep1_read/ZW_UAB_ep2_read** is used to read data from the host.

The USB controller can notify the application about the following events:

- Data from the host is available.
- Data to the host is sent.
- Soft reset
- Reset
- Suspend

The events can be enabled indevdually by using the **ZW_USB_int_src_enable**. Then to get them the function **ZW_USB_int_src_get** is used, the events should be cleard again by the SW by using the function **ZW_USB_int_src_clear.** Either the above events can be read by polling the USB controller or when an USB interrupt occur. Enable the USB controller by calling **ZW_USB_int_enable**.

5.4.19.1 ZW_USB_init

```
BYTE ZW_USB_init( WORD vendorID,
                  WORD productID,
                  WORD deviceBCD,
                  BYTE epNum )
```

This function initializes the USB controller.

Defined in: ZW_USB_api.h

Parameters:

vendorID IN	16-bit unique value
productID IN	16-bit unique value
deviceBCD IN	The device release number in BCD format
epNum IN	Number of the endpoints used (max 2)

Serial API (Not supported)

5.4.19.2 ZW_USB_disable

void ZW_USB_disable(void)

This function shuts down the internal USB controller and transceiver, to conserve power. The interface can be restarted with ZW_USB_init.

Defined in: ZW_USB_api.h

Parameters:

Serial API (Not supported)

5.4.19.3 ZW_USB_ep1_write

BYTE ZW_USB_ep1_write(BYTE * buffer,
BYTE len)

This function tries to copy data from buffer to the USB FIFO. The function will return the length and the data that was realy copied to the USB FIFO

Defined in: ZW_USB_api.h

Return value

The length of the data that was copied to the USB fifo

Parameters:

buffer IN	Address of the data to be sent to the host
Len IN	The length of the data to be sent to the host

Serial API (Not supported)

5.4.19.4 ZW_USB_ep2_write

BYTE ZW_USB_ep2_write(BYTE * buffer,
BYTE len)

This function tries to copy data from buffer to the USB FIFO. The function will return the length and the data that was realy copied to the USB FIFO.

Defined in: ZW_USB_api.h

Return value

The length of the data that was copied to the USB fifo

Parameters:

buffer IN	Address of the data to be sent to the host
Len IN	The length of the data to be sent to the host

Serial API (Not supported)

5.4.19.5 ZW_USB_ep1_read

```
BYTE ZW_USB_ep1_read( BYTE * buffer,
                      BYTE len)
```

This function tries to copy data from the USB FIFO to the buffer. The function will return the length and the data that was realy copied to the data buffer.

Defined in: ZW_USB_api.h

Return value

The length of the data that was copied to the data buffer

Parameters:

Buffer IN	Address of the data buffer to copy data in it.
Len IN	The length of the data to be copied from the USB FIFO

Serial API (Not supported)

5.4.19.6 ZW_USB_ep2_read

```
BYTE ZW_USB_ep2_read( BYTE * buffer,
                      BYTE len)
```

This function tries to copy data from the USB FIFO to the buffer. The function will return the length and the data that was realy copied to the data buffer

Defined in: ZW_USB_api.h

Return value

The length of the data that was copied to the data buffer

Parameters:

Buffer IN	Address of the data buffer to copy data in it.
Len IN	The length of the data to be copied from the USB FIFO

Serial API (Not supported)

5.4.19.7 ZW_USB_int_enable

void ZW_USB_int_enable(BOOL intEable)

This function enable/disable the USB interrupt.

Note: Declare ISR in application when using USB interrupt. The USB interrupt vector is **INUM_USB**

Defined in: ZW_USB_api.h

Parameters:

intEnbale IN	TRUE	Enbale the USB interrupt
	FALSE	Disable the USB interrupt

Serial API (Not supported)

5.4.19.8 ZW_USB_int_src_enable

void ZW_USB_int_src_enable(BYTE intSrcMask)

This function enable/disable one or more of the available USB events. An USB event can be enabled by setting the corresponding bit mask to 1 and is disaled by setting it to 0.

Note: Declare at least one event when using USB interrupt.

Defined in: ZW_USB_api.h

Parameters:

intSrcMask IN	USB_EP1_TX_INT	Data is sent from endpoint 1 fFIFO
	USB_EP2_TX_INT	Data is sent from endpoint 2 FIFO
	USB_EP1_RX_INT	Data is received in endpoint 1
	USB_EP2_RX_INT	Data is received in endpoint 2
	USB_SOFTRESET_INT	Soft rest request is sent from host to device
	USB_RESET_INT	Reset request is sent from host to device.
	USB_SUSPEND_INT	Devcie is suspended

Serial API (Not supported)

5.4.19.9 ZW_USB_int_src_get

BYTE void ZW_USB_int_src_get(void)

This function read the source of the USB controller events if any. The function will return a bit mask of the events sources. A return value of zero indicates that no event occurred.

Defined in: ZW_USB_api.h

Return value:

USB_EP1_TX_INT	Data is sent from endpoint 1 fFIFO
USB_EP2_TX_INT	Data is sent from endpoint 2 FIFO
USB_EP1_RX_INT	Data is received in endpoint 1
USB_EP2_RX_INT	Data is received in endpoint 2
USB_SOFTRESET_INT	Soft rest request is sent from host to device
USB_RESET_INT	Reset request is sent from host to device.
USB_SUSPEND_INT	Devcie is suspended

Serial API (Not supported)

5.4.19.10 ZW_USB_int_src_clear

void ZW_USB_int_src_clear(BYTE intSrcMask)

This function clear USB controller event source if any occurred. The parameter for the function is a bitmask of the USB controller events to be cleared.

Note: the function should be called if the **ZW_USB_int_scr_get** return non-zero value.

Defined in: ZW_USB_api.h

Parameters:

intSrcMask IN	USB_EP1_TX_INT	Data is sent from endpoint 1 fFIFO
	USB_EP2_TX_INT	Data is sent from endpoint 2 FIFO
	USB_EP1_RX_INT	Data is received in endpoint 1
	USB_EP2_RX_INT	Data is received in endpoint 2
	USB_SOFTRESET_INT	Soft rest request is sent from host to device
	USB_RESET_INT	Reset request is sent from host to device.
	USB_SUSPEND_INT	Devcie is suspended

Serial API (Not supported)

5.5 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…).

5.5.1 ZW_AddNodeToNetwork

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

Macro: ZW_ADD_NODE_TO_NETWORK(mode, func)

ZW_AddNodeToNetwork is used to add any nodes to the Z-Wave network.

The process of adding a node is started by calling ZW_AddNodeToNetwork() with the mode set to ADD_NODE_ANY, ADD_NODE_SLAVE or ADD_NODE_CONTROLLER. When the learn process is started the caller will get a number of status messages through the callback function completedFunc.

ADD_NODE_EXISTING is used when you do not want to learn new nodes but only get node info from nodes that are already known in the system.

Low power transmission mode is normally used during node inclusion. The option ADD_NODE_OPTION_HIGH_POWER can be added to the bMode parameter for Normal Power inclusion to extend inclusion range.

Network Wide Inclusion mode is by default disabled. For enabling of the Network Wide Inclusion feature the option bit ADD_NODE_OPTION_NETWORK_WIDE must be ORed to the bMode parameter.

The callback function will be called multiple times during the learn process to report the progress of the learn to the application. The LEARN_INFO will only contain a valid pointer to the Node Information Frame from the new node when the status of the callback is ADD_NODE_STATUS_ADDING_SLAVE or ADD_NODE_STATUS_ADDING_CONTROLLER.

WARNING: It is not allowed to call ZW_AddNodeToNetwork() between a ADD_NODE_STATUS_ADDING_* and a ADD_NODE_STATUS_PROTOCOL_DONE callback status, doing this can result in malfunction of the protocol.

NOTE: The learn state should ALWAYS be disabled after use to avoid adding other nodes than expected. It is recommended that ZW_AddNodeToNetwork() is called with ADD_NODE_STOP every time a ADD_NODE_STATUS_DONE callback is received, and that the controller also contains a timer that disables the learn state.

Defined in: ZW_controller_api.h

Parameters:

Mode IN	The learn node states are:	
	ADD_NODE_ANY	Add any type of node to the network
	ADD_NODE_SLAVE	Only add slave nodes to the network
	ADD_NODE_CONTROLLER	Only add controller nodes to the network
	ADD_NODE_EXISTING	Only get node info from nodes that are already included in the network
	ADD_NODE_STOP	Stop adding nodes to the network
	ADD_NODE_STOP_FAILED	Report a failure in the application part of the learn process
	ADD_NODE_OPTION_HIGH_POWER	Set this flag also in bMode for Normal Power inclusion
	ADD_NODE_OPTION_NETWORK_WIDE	Set this flag also in bMode for accepting Network Wide Inclusion Requests
completedFunc IN	Callback function pointer Should be NULL when learn state is turned off (ADD_NODE_STOP and ADD_NODE_STOP_FAILED)	

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_SLAVE	A new slave node has been added to the network
	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
	ADD_NODE_STATUS_NOT_PRIMARY	The call is not allowed because the controller is not a primary controller.
*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 | 0x4A | mode | funcID

ZW->HOST: REQ | 0x4A | funcID | bStatus | bSource | bLen | basic | generic | specific | cmdclasses[ ]

5.5.2 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

5.5.3 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 5.4.3.1
	TRANSMIT_COMPLETE_NOROUTE	No routes assigned because a route between source and destination node could not be found.

Serial API:

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

ZW->HOST: RES | 0x46 | retVal

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

5.5.4 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 IN	Transmit complete call back.

Callback function Parameters:

bStatus IN	(see **ZW_SendData**)

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

5.5.5 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).	

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.	

Serial API:

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

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

5.5.6 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

5.5.7 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 IN	Transmit complete call back.

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.

5.5.8 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 built-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

5.5.9 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

5.5.10 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 non-volatile memory 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 34. 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 initialization, 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 above)	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 above)

5.5.11 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 ≤ 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.
	ZW_GET_ROUTING_INFO_9600 or ZW_GET_ROUTING_INFO_40K or ZW_GET_ROUTING_INFO_100K or ZW_GET_ROUTING_INFO_ANY	Return only nodes supporting this speed. Only one option may be used at a time.

Serial API:

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

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

5.5.12 ZW_GetRoutingMAX

BYTE ZW_GetRoutingMAX(void)

Use this function to get the maximum maximum number of source routing attempts before the explorer frame mechanism kicks-in.

Defined in: ZW_controller_api.h

Return value:

BYTE	1...20	Maximum number of source routing attempts

Serial API:

Not implemented

5.5.13 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

5.5.14 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

5.5.15 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)

5.5.16 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 can't be removed. The call works on a primary controller and an inclusion controller.

A call back function should be provided otherwise the function will 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 IN	If TRUE then using Normal RF Power.
completedFunc IN	Remove process completed call back function

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

5.5.17 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 would 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.

Parameters:

nodeID IN	The node ID (1…232) of the failed node to be deleted.
bNormalPower IN	If TRUE then using Normal RF Power.
completedFunc IN	Replace process completed call back 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

5.5.18 ZW_RemoveNodeFromNetwork

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

Macro: ZW_REMOVE_NODE_FROM_NETWORK(mode, func)

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

The process of removing a node is started by calling ZW_RemoveNodeFromNetwork() with the mode set to REMOVE_NODE_ANY, REMOVE_NODE_SLAVE or REMOVE_NODE_CONTROLLER. When the delete process is started the caller will get a number of status messages through the callback function completedFunc.

The callback function will be called multiple times during the delete process to report the progress to the application. The LEARN_INFO will only contain a valid pointer to the Node Information Frame from the node that is deleted when the status of the callback is REMOVE_NODE_STATUS_REMOVING_SLAVE or REMOVE_NODE_STATUS_REMOVING_CONTROLLER.

The delete process is complete when the callback function is called with the status REMOVE_NODE_STATUS_DONE.

WARNING: It is not allowed to call ZW_RemoveNodeFromNetwork() between a REMOVE_NODE_STATUS_REMOVING_* and a REMOVE_NODE_STATUS_DONE callback status, doing this can result in malfunction of the protocol.

NOTE: The learn state should ALWAYS be disabled after use to avoid adding other nodes than expected. It is recommended that ZW_RemoveNodeFromNetwork() is called with REMOVE_NODE_STOP every time a REMOVE_NODE_STATUS_DONE callback is received, and that the controller also contains a timer that disables the learn state.

Defined in: ZW_controller_api.h

Parameters:

mode IN	The learn node states are:	
	REMOVE_NODE_ANY	Remove any type of node from the network
	REMOVE_NODE_SLAVE	Only remove slave nodes from the network
	REMOVE_NODE_CONTROLLER	Only remove controller nodes from the network
	REMOVE_NODE_STOP	Stop the delete process
completedFunc IN	Callback function pointer Should be NULL when learn state is turned off (REMOVE_NODE_STOP)	

Callback function Parameters (completedFunc):

*learnNodeInfo.bStatus IN	Status of learn mode:	
	REMOVE_NODE_STATUS_LEARN_READY	The controller is now ready to remove a node from the network.
	REMOVE_NODE_STATUS_NODE_FOUND	A node that wants to be deleted from the network has been found
	REMOVE_NODE_STATUS_REMOVING_*	A slave/controller node has been removed from the network. Remove node ID is returned.
	REMOVE_NODE_STATUS_DONE	The node has now been removed and the controller is ready to continue normal operation again.
	REMOVE_NODE_STATUS_FAILED	The remove process failed
*learnNodeInfo.bSource IN	Node id of the removed 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 | 0x4B | mode | funcID

ZW->HOST: REQ | 0x4B | funcID | bStatus | bSource | bLen | basic | generic | specific | cmdclasses[ ]

5.5.19 ZW_ReplicationReceiveComplete

void ZW_ReplicationReceiveComplete(void)

Macro: ZW_REPLICATION_COMMAND_COMPLETE

Sends command completed to sending controller. Called in replication mode when a command from the sender has been processed and indicates that the controller is ready for next packet.

Defined in: ZW_controller_api.h

Serial API:

HOST->ZW: REQ | 0x44

5.5.20 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 is in replication mode. It sends the payload and expects the receiver to respond with a command complete message (ZW_REPLICATION_COMMAND_COMPLETE).

Messages sent using this command should always be part of the Z-Wave controller replication command class.

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 IN	Transmit completed call back function

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

5.5.21 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. The **ZW_RequestNodeInfo** API 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 IN	Transmit complete call back.

Callback function Parameters:

txStatus IN	(see **ZW_SendData**)

Serial API:

HOST->ZW: REQ | 0x60 | NodeID

ZW->HOST: RES | 0x60 | retVal

5.5.22 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 IN	Transmit complete call back.

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

ZW->HOST: REQ | 0x48 | funcID | bStatus

5.5.23 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 IN	Transmit complete call back.

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

5.5.24 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 EEPROM memory. Finally write a new random home ID to the EEPROM memory.

NOTE: This function should not be used on a secondary controller, use ZW_SetLearnMode() instead and use the primary controller to remove it from the network.

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

5.5.25 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. The controller is assigned a valid Home ID and Node ID
1. The controller receives and stores the node table and routing table for the network
2. The application receives and stores application information transmitted as part of the replication

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.

NOTE: Learn mode should only be enabled when necessary, and it should always be disabled again as quickly as possible. However to ensure a successful synchronization of the inclusion process the device should be able to stay in learn mode in up to 5 seconds.

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 in direct range.
	ZW_SET_LEARN_MODE_NWI	Start the learn mode on the controller and accept routed inclusion.
	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).	

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[ ]

5.5.26 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 ≤ 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

5.5.27 ZW_SetRoutingMAX

BOOL ZW_SetRoutingMAX(BYTE maxRouteTries)

Use this function to set the maximum number of source routing attempts before the explorer frame mechanism kicks-in. Default value with respect to maximum number of source routing attempts is five. Remember to enable the explorer frame mechanism by setting the transmit option flag TRANSMIT_OPTION_EXPLORE in the send data calls.

A ZDK 4.5 controller uses the routing algorithm from 5.02 to address nodes from ZDK's not supporting explorer frame. The routing algorithm from 5.02 ignores the transmit option TRANSMIT_OPTION_EXPLORE flag and maximum number of source routing attempts value (maxRouteTries).

Defined in: ZW_controller_api.h

Parameters:

maxRouteTries	IN	1…20	Maximum number of source routing attempts

Serial API:

Not implemented

5.5.28 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 is 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.

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 IN	Transmit complete call back.	

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.

5.6 Z-Wave Static Controller API

The Static Controller application interface is an extended Controller application interface with added functionality specific for the Static Controller.

5.6.1 ZW_EnableSUC

BYTE ZW_EnableSUC (BYTE state, BYTE capabilities)

Macro: ZW_ENABLE_SUC (state)

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

5.6.2 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).	

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.	

Serial API:

HOST->ZW: REQ | 0x4C | mode | funcID

ZW->HOST: REQ | 0x4C | funcID | bStatus | bSource | bLen | basic | generic | specific | cmdclasses[ ]

5.7 Z-Wave Bridge Controller API

The Bridge Controller application interface is an extended Controller application interface with added functionality specific for the Bridge Controller.

5.7.1 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:

HOST->ZW: REQ | 0xA2 | srcNode | destNode | txOptions | funcID

ZW->HOST: RES | 0xA2 | retVal

ZW->HOST; REQ | 0xA2 | funcID | txStatus

5.7.2 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.

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	

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

5.7.3 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

5.7.4 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 ≤ 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]

5.8 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.

5.8.1 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)

5.8.2 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

5.8.3 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

5.9 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.

5.9.1 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. Setting the ID's to zero will remove the Slave node from the Z-Wave network, so that it can be moved to another network.

The Slave node must identify itself to the primary controller node by sending a Node Information Frame (see **ZW_SendNodeInformation**).

When learn mode is enabled, received "Assign ID's Command" are handled as follow:
1. If the current stored ID's are zero, the received ID's will be stored.
2. If the received ID's are zero the stored ID's will be set to zero.

The **learnFunc** is called as the "Assign" process progresses. The returned nodeID is the nodes new Node ID. If no "Assign" is received the callback function will not be called. It is then up to the main application code to switch of Learn mode. Once the assignment process has been started the Callback function may be called more than once. It is not until the callback function is called with ASSIGN_COMPLETE the learning process is done.

NOTE: Enable only learn mode when necessary and disabled again as quickly as possible. Recommend never enabling learn mode more than 1 second.

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 in direct range.
	ZW_SET_LEARN_MODE_NWI	Start the learn mode on the slave and accept routed inclusion.
	ZW_SET_LEARN_MODE_DISABLE	Stop learn mode on the slave
learnFunc IN	Node ID learn mode completed call back function	

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

5.9.2 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. 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

5.10 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.

5.10.1 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

5.10.2 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

Serial API:

HOST->ZW: REQ | 0x5D | bNodeID

ZW->HOST: RES | 0x5D | retVal

5.10.3 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 IN	Transmit completed call back function

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

5.10.4 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 IN	Number of destinations contained in pDestList.
completedFunc IN	Transmit completed call back function

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

5.10.5 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. The **ZW_RequestNodeInfo** API 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 IN	Transmit complete call back.

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.

5.11 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.

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
>
```

5.11.1 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)

5.11.2 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)

5.12 RF Settings in App_RFSetup.a51 file

RF normal and low power transmit levels is determined in the file App_RFSetup.a51.

Table 11. 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. If not valid default values are used.

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

4	FLASH_APPL_LOW_POWER_OFFS	0xFF		If 0xFF the default lib value is used:0x14

TXnormal Power need maybe adjustment to fulfil FCC compliance tests. 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.

The RF power transmit levels can be adjusted directly on the module by the Z-Wave Programmer [14].

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.

6.2.2 Updates from the Primary Controller

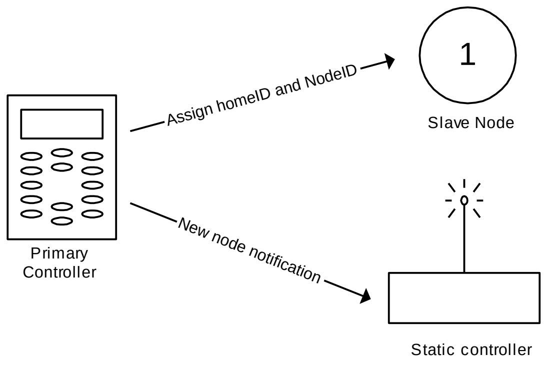

Figure 35. Inclusion of a node having a SUC in the network

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 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.

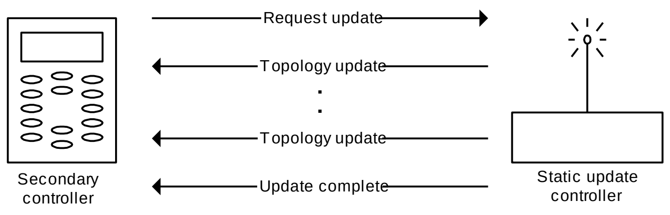

Figure 36. Requesting network updates from a SUC in the network

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.

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.

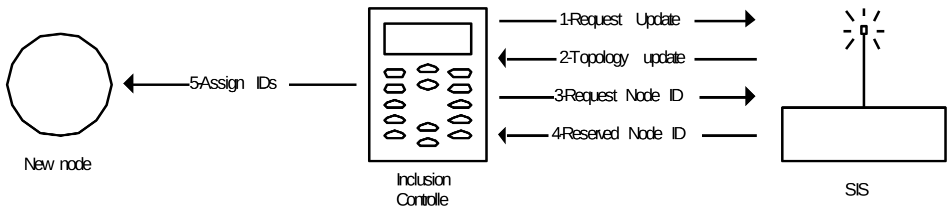

Figure 37. Inclusion of a node having a SIS in the network

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 continues 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.

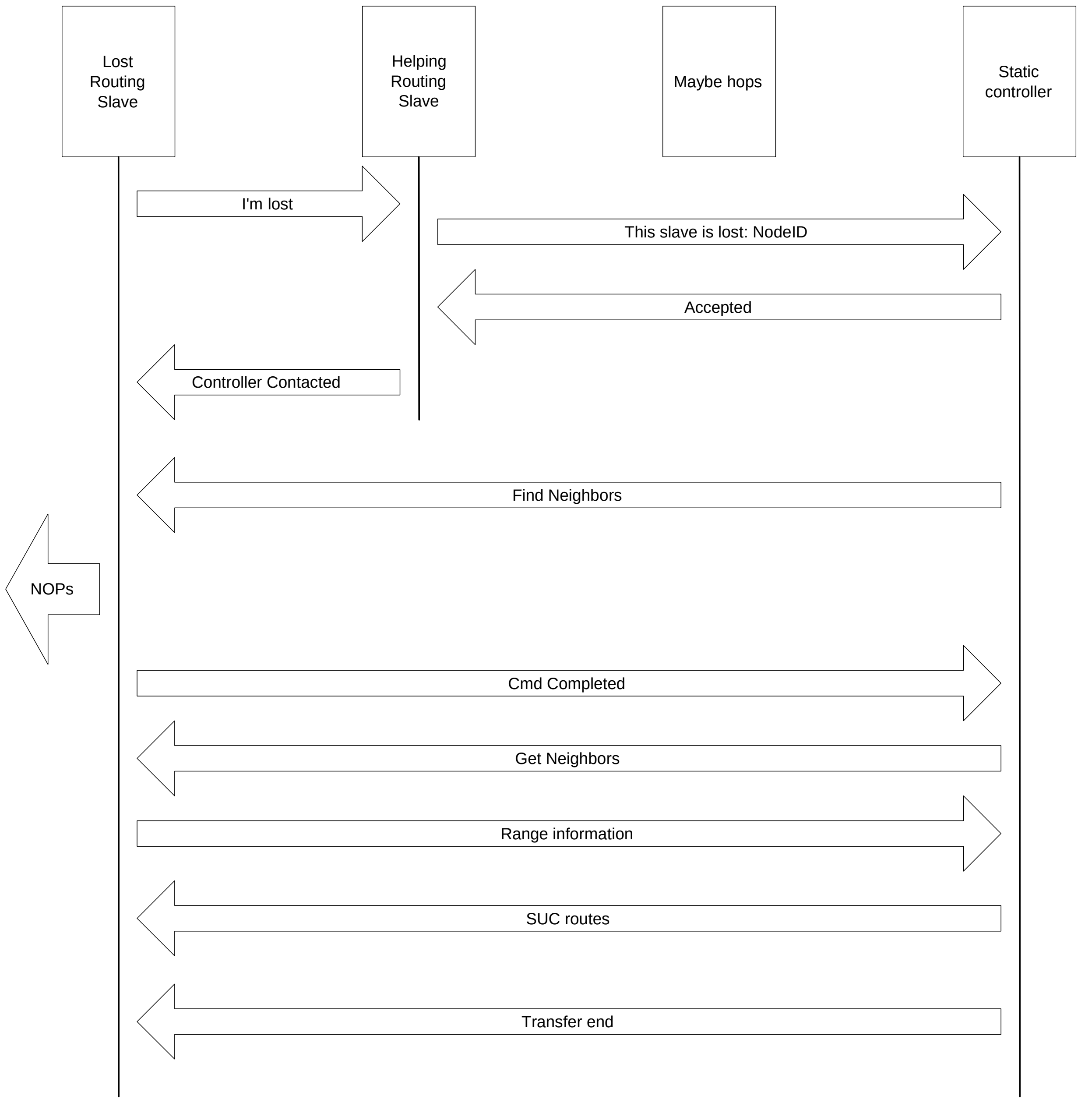

Figure 38. Lost routing slave frame flow

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.

7 APPLICATION NOTE: INCLUSION/EXLUCSION IMPLEMENTATION

This note describes the API calls the System 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.

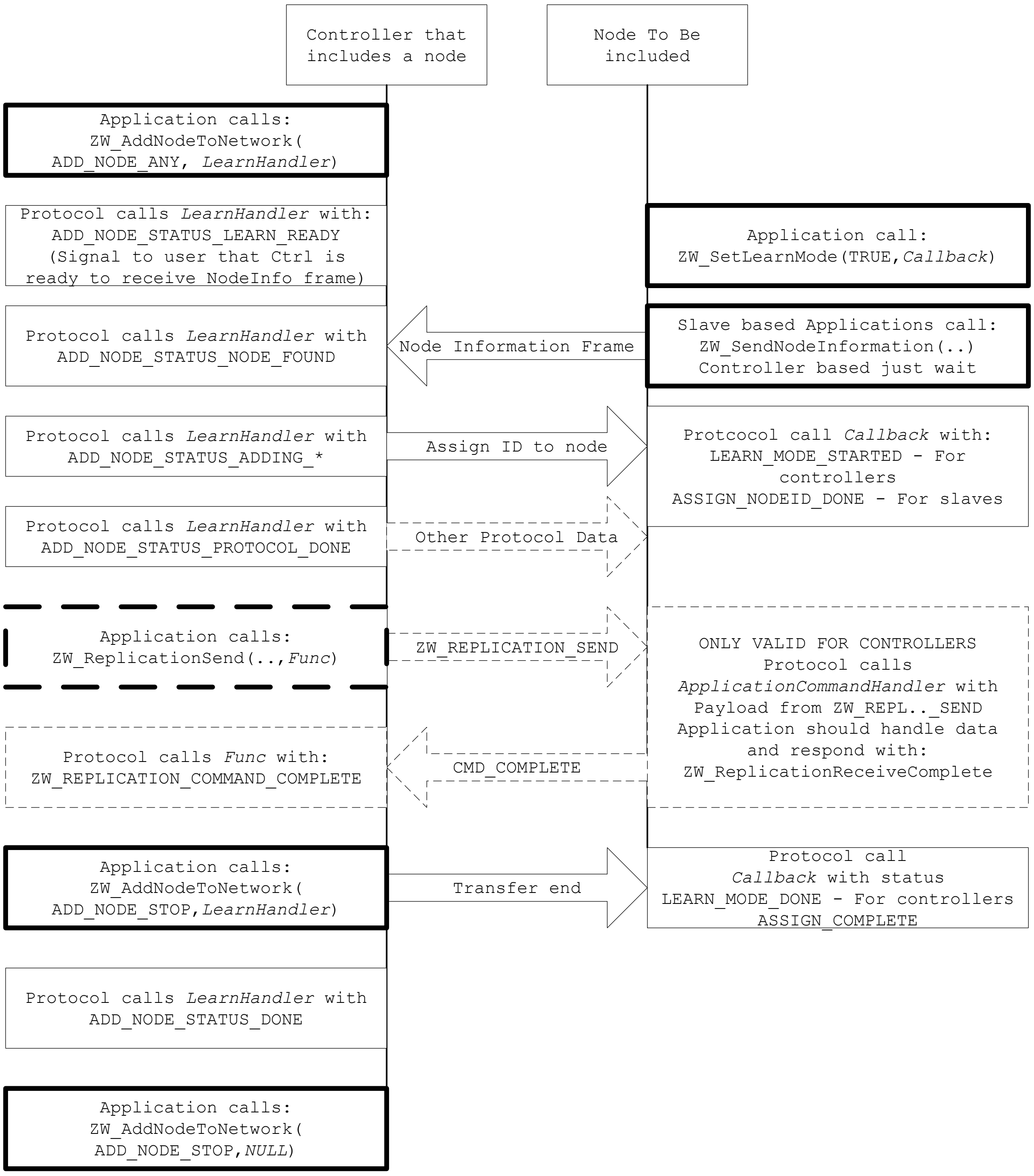
Controller that includes a node
Node To Be included
Application calls: ZW_AddNodeToNetwork(ADD_NODE_ANY, *LearnHandler*)
Protocol calls *LearnHandler* with: ADD_NODE_STATUS_LEARN_READY (Signal to user that Ctrl is ready to receive NodeInfo frame)
Application call: ZW_SetLearnMode(TRUE,*Callback*)
Protocol calls *LearnHandler* with ADD_NODE_STATUS_NODE_FOUND
Node Information Frame
Slave based Applications call: ZW_SendNodeInformation(..) Controller based just wait
Protocol calls *LearnHandler* with ADD_NODE_STATUS_ADDING_*
Assign ID to node
Protcocol call *Callback* with: LEARN_MODE_STARTED - For controllers ASSIGN_NODEID_DONE - For slaves
Protocol calls *LearnHandler* with ADD_NODE_STATUS_PROTOCOL_DONE
Other Protocol Data
Application calls: ZW_ReplicationSend(..,*Func*)
ZW_REPLICATION_SEND
ONLY VALID FOR CONTROLLERS Protocol calls *ApplicationCommandHandler* with Payload from ZW_REPL.._SEND Application should handle data and respond with: ZW_ReplicationReceiveComplete
Protocol calls *Func* with: ZW_REPLICATION_COMMAND_COMPLETE
CMD_COMPLETE
Application calls: ZW_AddNodeToNetwork(ADD_NODE_STOP,*LearnHandler*)
Transfer end
Protocol call *Callback* with status LEARN_MODE_DONE - For controllers ASSIGN_COMPLETE
Protocol calls *LearnHandler* with ADD_NODE_STATUS_DONE
Application calls: ZW_AddNodeToNetwork(ADD_NODE_STOP,*NULL*)

Figure 39. 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 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. 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 non-volatile memory 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 */
  }
}
```

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*/
  }
}
```

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

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.

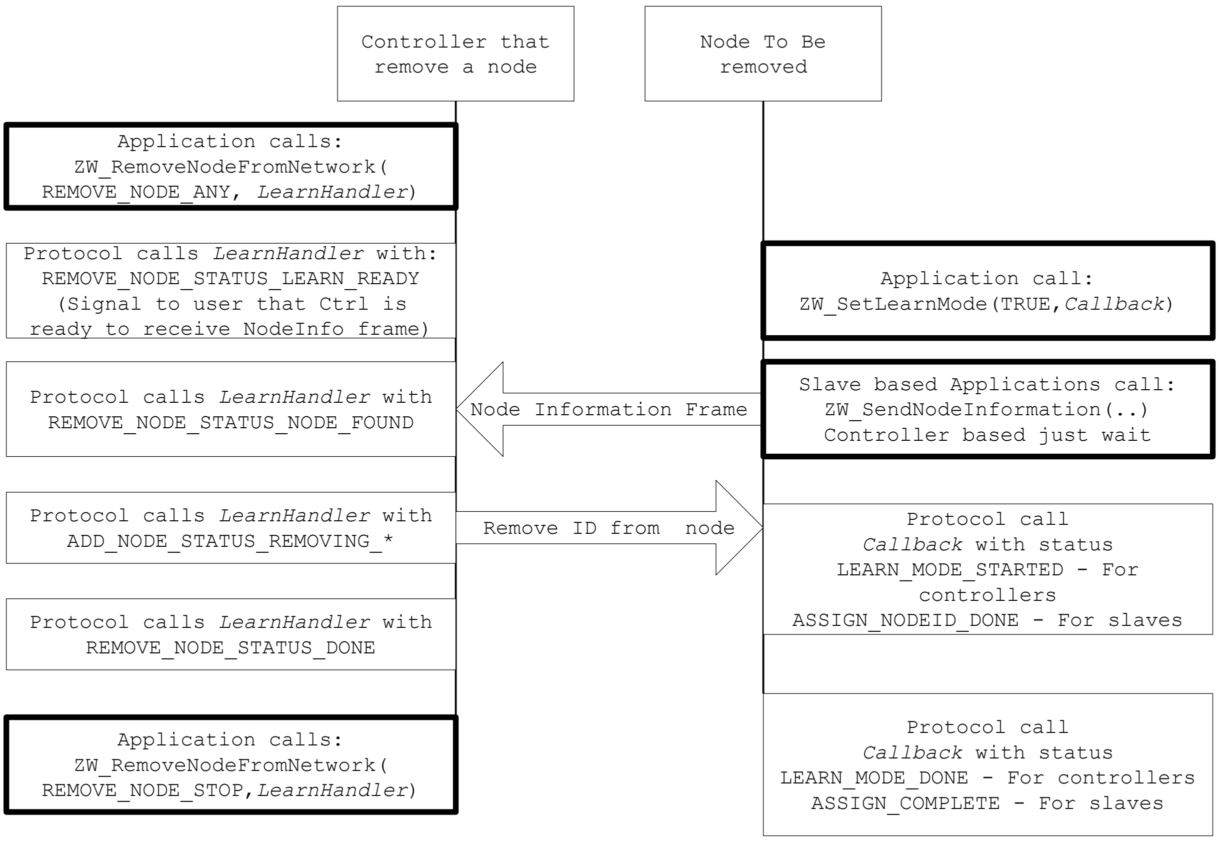
Controller that remove a node
Node To Be removed
Application calls: ZW_RemoveNodeFromNetwork(REMOVE_NODE_ANY, *LearnHandler*)
Protocol calls *LearnHandler* with: REMOVE_NODE_STATUS_LEARN_READY (Signal to user that Ctrl is ready to receive NodeInfo frame)
Application call: ZW_SetLearnMode(TRUE,*Callback*)
Protocol calls *LearnHandler* with REMOVE_NODE_STATUS_NODE_FOUND
Node Information Frame
Slave based Applications call: ZW_SendNodeInformation(..) Controller based just wait
Protocol calls *LearnHandler* with ADD_NODE_STATUS_REMOVING_*
Remove ID from node
Protocol call *Callback* with status LEARN_MODE_STARTED - For controllers ASSIGN_NODEID_DONE - For slaves
Protocol calls *LearnHandler* with REMOVE_NODE_STATUS_DONE
Application calls: ZW_RemoveNodeFromNetwork(REMOVE_NODE_STOP,*LearnHandler*)
Protocol call *Callback* with status LEARN_MODE_DONE - For controllers ASSIGN_COMPLETE - For slaves

Figure 40. Node exclusion frame flow

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.

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.

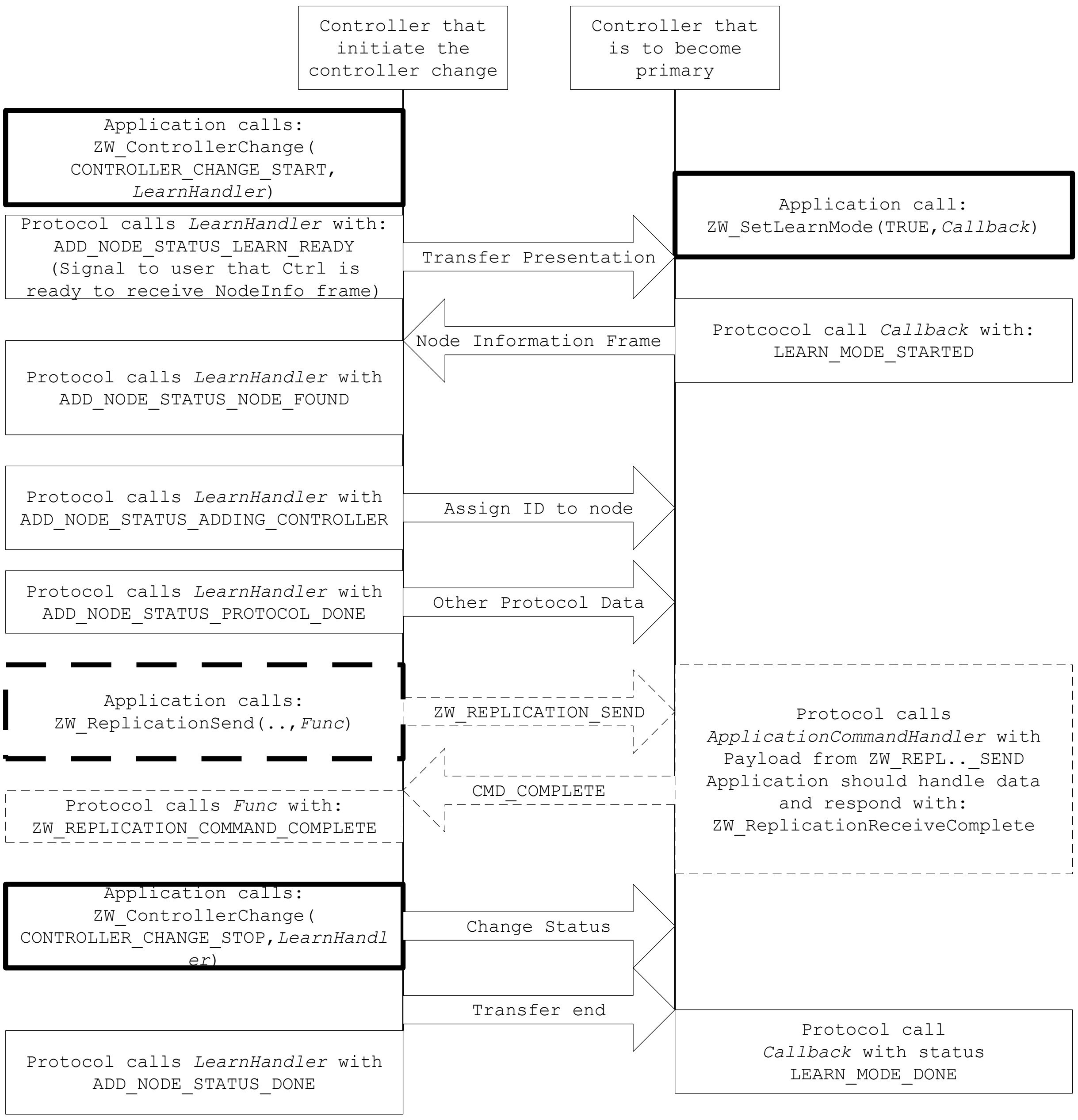
Controller that initiate the controller change
Controller that is to become primary
Application calls: ZW_ControllerChange(CONTROLLER_CHANGE_START, LearnHandler)
Application call: ZW_SetLearnMode(TRUE,Callback)
Protocol calls LearnHandler with: ADD_NODE_STATUS_LEARN_READY (Signal to user that Ctrl is ready to receive NodeInfo frame)
Transfer Presentation
Protcocol call Callback with: LEARN_MODE_STARTED
Node Information Frame
Protocol calls LearnHandler with ADD_NODE_STATUS_NODE_FOUND
Protocol calls LearnHandler with ADD_NODE_STATUS_ADDING_CONTROLLER
Assign ID to node
Protocol calls LearnHandler with ADD_NODE_STATUS_PROTOCOL_DONE
Other Protocol Data
Application calls: ZW_ReplicationSend(..,Func)
ZW_REPLICATION_SEND
Protocol calls ApplicationCommandHandler with Payload from ZW_REPL.._SEND Application should handle data and respond with: ZW_ReplicationReceiveComplete
CMD_COMPLETE
Protocol calls Func with: ZW_REPLICATION_COMMAND_COMPLETE
Application calls: ZW_ControllerChange(CONTROLLER_CHANGE_STOP,LearnHandler)
Change Status
Transfer end
Protocol call Callback with status LEARN_MODE_DONE
Protocol calls LearnHandler with ADD_NODE_STATUS_DONE

Figure 41. Controller shift frame flow

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, SRN11332 ZW0201/ZW0301 Developer’s Kit v4.50 (Beta1) Patch1.
[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, INS11442, Instruction, Z-Wave 400 Series Z-Wave Single Chip Developer’s Kit v6.00 Contents.
[35] SD, APL10742, Application Note, ZM3102N with External PA and Switch

INDEX